Retrofitting Language-oriented Design with SugarJ

Stefan Fehrenbach

Fachbereich Mathematik und Informatik
Philipps-Universität Marburg

A thesis submitted for the degree of
Bachelor of Science

Supervisors:
Prof. Dr. Klaus Ostermann
Sebastian Erdweg, MSc.

November 2011

Abstract

Language-oriented design embraces the use of language-based abstractions to improve understandability, extensibility, and maintainability of code. Most existing applications are not designed and implemented in a language-oriented way. The use of domain-specific language extensions could improve them in many regards but rewriting them from scratch is far too tedious. With SugarJ you can incrementally and independently introduce language-oriented design techniques to those parts of your legacy code that benefit the most from them. The result is a modernised code base that is less likely to contain bugs, and easier to extend and maintain.

Acknowledgements

I would like to express my deepest gratitude to all those who made it possible for me to complete this thesis. I am deeply indebted to my supervisors Prof. Ostermann who asked excellent questions that shaped my understanding of my work, and Sebastian Erdweg whose help, encouragement, and stimulating suggestions are reflected in every good part of this thesis.
I am heartily grateful to my friends and family for being patient and understanding and supporting me during the whole process of research and writing. I would like to give my special thanks to my friend M. S. for proof reading, suggestions and enduring rants, as well as to my mother and my friend U. W. for having been, and still being, wonderful persons and role models for both academia and real life.
Lastly, I offer my regards to every single person who is a source of knowledge, inspiration, and motivation to somebody else. You make the world a better place.

Contents

1 Introduction **3**

2 Incremental Introduction of Language-oriented Design **7**
2.1 Step one: Identify one particular problem or domain 7
2.2 Step two: Design a language-based abstraction 8
2.3 Step three: Write a sugar library 9
2.4 Step four: Apply 9
2.5 If not perfect, repeat 9

3 Background on SugarJ **11**
3.1 Using a sugar library 11
3.2 Writing a sugar library 12
3.2.1 Pair syntax 12
3.2.2 Pair desugaring 13
3.3 Editor services 15

4 Retrofitting Language-oriented Design: A Case Study **15**
4.1 JavaBeans style properties 15
4.1.1 Language support for properties 16
4.1.2 The accessors sugar library 17
4.1.3 The accessors sugar library applied to the Java Pet Store 17
4.2 XML 18
4.2.1 Language support for XML 18
4.2.2 The XML sugar library 19
4.3 Java Persistence Query Language 20
4.3.1 Problems with JPQL 21
4.3.2 Host language integration 21
4.3.3 Editor support 22
4.3.4 Fully parsed queries 24
4.3.5 Static checks 25

5 Technical realisation **27**
5.1 Accessors 27
5.1.1 Syntax 27

5.1.2 Desugaring 28
5.2 XML 30
5.2.1 Syntax 30
5.2.2 Desugaring 31
5.2.3 Editor support 32
5.3 BNF 32
5.3.1 Syntax 33
5.3.2 Desugaring 34
5.4 Java Persistence Query Language 35
5.4.1 Syntax 35
5.4.2 Desugaring 35
5.4.3 Editor support 36
5.4.4 Static checks 37

6 Discussion and future work **39**
6.1 The Java Pet Store as a model legacy application 39
6.2 Retrofitting language-oriented design and the role of SugarJ 40

7 Conclusion **42**

1. Introduction

For some years now, researchers and practitioners propose new ways to design programs with a focus on one fundamental constituent of every software, its programming language. Terminology and exact approaches differ, so we will use *language-oriented design* throughout this thesis to mean a union of programming techniques and design principles that focus on programming languages. Mainly, the use of domain-specific languages, language-oriented programming [17] as described by M. P. Ward, and the idea of growing a language [13] to meet our needs that has been brilliantly articulated by Guy L. Steele Jr.

Ward's approach, in broad strokes, is to write a program neither bottom-up nor top-down but middle-out by first designing a domain-specific programming language tailored to the problem at hand, implementing this language, and writing the actual application in this new language. The main idea behind growing a language is not to start from scratch but extend a given general-purpose language with domain-specific features.

Language-oriented design has many benefits. Appropriate use of languages increases developer productivity because languages raise the level of abstraction, separating domain concerns from their implementation. Well designed domain-specific languages encapsulate domain knowledge and are easy to use by presenting domain concepts in domain syntax. This increased clarity improves long-term maintainability and extensibility. Domain abstraction and a single implementation of domain concepts opens up opportunities for domain-specific analyses for error reporting, consistency and static safety, and optimisations [14].

Currently, for large parts of the software industry, the programming language of choice is Java and with it, almost invariably, comes object-oriented programming. Large and complex software was built in this style, which indisputably shows that object-oriented programming is practical. As the programs grow larger, however, we start to see scalability problems with object-oriented design. Sergey Dmitriev [6] makes the case that most of these problems can be traced back to a mismatch between domain concepts and their model in a general-purpose object-oriented programming language by means of classes, objects, and methods, and their runtime behaviour. Mapping domain concepts to the use of a class library is a considerable mental

effort that slows development. This affects maintenance even more, as there is an additional, reverse, mapping involved. To correct a bug, a programmer has to translate code from the general purpose language to her mental model of domain concepts, find a solution in terms of domain concepts, and translate the solution back to code. The use of domain-specific languages with concrete syntax narrows the representational gap and thereby improves understandability, extensibility, and maintainability.

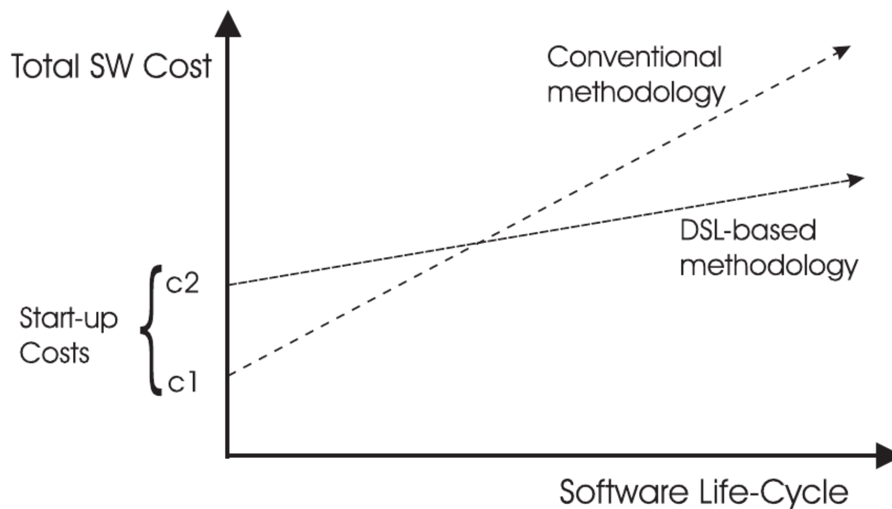

Figure 1.1: Total cost of ownership over the software life-cycle by design methodology. Illustration by Paul Hudak [10]

The most discussed disadvantage of language-oriented design is the start-up costs, which are usually higher compared to a more conventional approach. Paul Hudak illustrates this nicely using a graph that is replicated in Figure 1.1.

Since large applications benefit the most from language-oriented design, it is unfortunate that existing literature does not address large legacy code bases. Instead, the focus is new design patterns, illustrated using small code samples that are easily written from scratch. This precludes widespread adoption in the software industry because rewriting an application from scratch that, because of its size, reaches the bounds of what is reasonably maintainable using object-oriented programming, is a huge undertaking promising little immediate benefit.

With this thesis, we aim to provide a way to bring legacy code bases from conventional object-oriented Java to a more language-oriented design. We explicitly avoid rewriting large amounts of code or changing overall architecture. Instead, we propose to incrementally introduce language-

oriented design to an existing application on demand.

The foundation of our incremental introduction of language-oriented design is libraries. Libraries can be developed independently and are flexible in their application, multiple libraries can be used jointly or independently, and in large to small sections of code. Furthermore, libraries are already fully accepted by the industry.

Paul Hudak [10] showed that libraries can be used to embed domain-specific languages. These embedded domain-specific languages are, however, host language libraries, and by relying heavily on host language syntax and semantics, sacrifice the benefits of domain syntax and analyses.

We decided to use SugarJ [8], a language based on Java that is syntactically extensible through sugar libraries. Sugar libraries provide domain-specific language extensions by extending Java's syntax and defining the extension's semantics through a translation to Java. Additionally, SugarJ provides means to extend the editing environment and implement domain-specific semantic code analyses.

The process we propose for retrofitting a legacy code base with language-oriented design is a four-step procedure that can be iterated as often as the resulting benefits out-weight the costs:

1. Identify one particular problem or domain, a source of confusion, verbosity, or likely mistakes.
2. Design a language-based abstraction for this domain.
3. Write a sugar library that extends Java with this language-based abstraction.
4. Import the library and locally rewrite problematic code employing the new abstraction to avoid the problem and to embrace the domain.

To evaluate our process, we applied this retrofitting of language-oriented design to the Java Pet Store, which "is the reference application for building Ajax web applications on Java Enterprise Edition 5 platform"[1] with encouraging results:

- We implemented several immediately reusable sugar libraries and demonstrate (in Section 3) that their application improves several aspects of the Java Pet Store code, specifically:
  - We eliminate boilerplate code and improve code readability by capturing common code patterns in sugar libraries.
  - With SugarJ we bring more static safety to existing code. In particular, SugarJ provides syntactic safety for parts of programs in embedded languages and enables domain-specific static analyses.

[1]http://java.sun.com/developer/releases/petstore/, accessed 17 November 2011

- We improve the editing experience when using embedded languages, by providing IDE support like content completion, syntax colouring, and domain-specific edit-time error reporting.

- We compare our use of SugarJ with other approaches and come to the conclusion that, only SugarJ's particular feature set, independent but composable language libraries, offers the necessary flexibility.
- We demonstrate that self-applicability is not only theoretically desirable but indeed very useful in practice. In this case it enabled us to reuse semi-formal language specification as actual code.
- Overall, we show that one can indeed use SugarJ to incrementally bring the benefits of language-oriented design to an existing legacy code base while the application remains production ready over the entire process.

2. Incremental Introduction of Language-oriented Design

In this section we describe the process we propose for introducing language-oriented design techniques to a traditionally designed, object-oriented, legacy code base.

We use the Java Pet Store 2.0 Reference Application as our example legacy code base. The Java Pet Store – written by Sun Microsystems to demonstrate the Java Enterprise Edition 5 – is a web application for selling pets.

2.1 Step one: Identify one particular problem or domain

```java
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(" <cat-id>" + i.getProductID() + "</cat-id>\n");
    sb.append(" <name>" + i.getName() + "</name>\n");
    sb.append(" <description><![CDATA[" + i.getDescription() + "]]></description>\n");
    sb.append(" <image-url>" + i.getImageURL() + "</image-url>\n");
    sb.append(" <price>" + NumberFormat.getCurrencyInstance(Locale.US).format(i.getPrice())  + "</price>\n");
    sb.append("</item>\n");
    return sb.toString();
}
```

Figure 2.1: Java Pet Store XML code

Exemplary, we identify the domain of XML document creation in the Java Pet Store as worthy of improvement.

The Java Pet Store uses XML to interchange data between client and server. Figure 2.1 shows server-side code for serialising a pet to XML.

Encoding domain-specific languages as strings has several drawbacks. To the compiler all strings are arbitrary data so there is no syntactic safety. The Java Pet Store does not employ validation of XML documents before sending them to the client, so any errors manifest far away from their origin. In general, string encoding requires the escaping of quotes. A future

extension of the Java Pet Store's data interchange format might require XML attributes and expose this problem.

There is most likely more than one area of potential improvement in any legacy code base. Rewriting from scratch to address all problems at once is too expensive and time consuming and according to Fred Brooks' second-system effect [3] not likely to succeed. Because our approach is built around composable libraries, we can address problems in the code base one at a time. Sugar libraries are also largely independent so we are at liberty to chose the domain that promises the largest cost-benefit ratio first.

2.2 Step two: Design a language-based abstraction

In the previous step we identified two areas of improvement. We would like to avoid the syntactic overhead of Java strings and `StringBuffer.append`, and add static syntax checking that alerts us in case of improperly nested elements and similar syntactic mistakes.

The most natural syntax for dealing with XML data is XML's syntax. Ideally, a function that returns a hello world XML document should be as easy as the following:

```
public String greet() {
  return <hello>world</hello>;
}
```

We also need access to Java for generating dynamic documents, to greet a user with his name, for example. We address this requirement by treating code in between `${` and `}` as Java and include the result of its evaluation in the XML document.

```
public String greet(User user) {
  return <hello>${ user.name }</hello>;
}
```

The return types in the above demonstration code expresses a second design choice, the semantics. In SugarJ, embedded languages derive their semantics from their translation to Java code. In case of the XML sugar library, we decided to translate literal XML to string expressions. This choice reduces the amount of rewriting needed in Step four.

The existing Java Pet Store code relies on strings and strongly suggests the semantics of this XML embedding, and the language embedded itself already has a standardised syntax. In general, the design of domain-specific languages tends to be more involved.

2.3 Step three: Write a sugar library

The next step is implementing the language extension designed in the previous step as a sugar library. For now it is sufficient to know that a parser and a subsequent semantic analysis report syntactic errors, and the XML greeting code from before is translated to this Java code:

```
public String greet(User user) {
  return String.format("<hello>%s</hello>", user.name);
}
```

An introduction to SugarJ can be found in Section 3 and the XML sugar library's implementation is discussed in more detail in Section 5.

2.4 Step four: Apply

The last step is to use the sugar library developed in Step three and designed in Step two, to improve the code that is concerned with the domain chosen in Step one.

Figure 2.2 shows the same method as Figure 2.1 but adapted to use the XML sugar library and thereby avoiding all deficiencies of string encoded XML.

```
private String handleItem(String targetId){
    Item i = cf.getItem(targetId);
    return  <item>
                <id>${i.getItemID()}</id>
                <cat-id>${i.getProductID()}</cat-id>
                <name>${i.getName()}</name>
                <description>${"<![CDATA[" + i.getDescription() + "]]>"}</description>
                <image-url>${i.getImageURL()}</image-url>
                <price>${NumberFormat.getCurrencyInstance(Locale.US).format(i.getPrice())}</price>
            </item>;
}
```

Figure 2.2: Method from Figure 2.1 using the XML sugar library

The XML sugar library was carefully designed to minimise the amount of rewriting of existing code. Especially the choice of translating to string-typed expressions keeps rewriting effort localised because no callers need to be adapted.

One important point for application in practice is that newly adapted and legacy code can coexist. There is no need to immediately adapt all eligible code. Rewriting can be deferred problem-free to an opportune moment, e.g. a pending extension involving related functionality.

2.5 If not perfect, repeat

At this point, we have a production ready, modified Java Pet Store code base that is clearer, safer, and more aesthetically pleasing than the original code.

The XML sugar library does compile time syntax checking, and the concrete XML syntax used is clearer and more readable than the string encoding that has been used before.

As we mentioned in Step one, the XML domain was likely not the only potential beneficiary of language-oriented design. Because sugar libraries are composable, it is possible to repeat these four steps, incrementally approaching a language-oriented design, or to stop as soon as a cost-benefit ratio limit is reached.

3. Background on SugarJ

This section provides background information on SugarJ, SDF and Stratego, the means of implementing the language-based abstractions we design and use in the previously discussed process of retrofitting language-oriented design. Feel free to skip ahead if you are familiar with SugarJ already.

SugarJ is a syntactically extensible language with powerful meta programming facilities. Seen from a pragmatic implementation perspective, it is a preprocessor that allows programmers to augment the general-purpose language Java with syntactic extensions called sugar libraries, similar to how they augment the standard class library with third party class libraries.

To illustrate the use and development of sugar libraries, we demonstrate the use and design of one of the most basic language extensions. The pair sugar library [8] allows programmers to use the familiar syntax for pairs that is used in Mathematics, in Java.

3.1 Using a sugar library

The use of the pair sugar library might look like in Listing 3.1, below. The first line `imports` the sugar library, just like any other library. From this point on, the program text is parsed with a modified grammar, that allows pair types and pair values to be written in the familiar mathematical notation.

```
import pair.Sugar;

public class Test {
  private (String, Integer) p = ("Answer", 42);
}
```

Listing 3.1: Use of the pair sugar library in a file named `Test.sugj`

Using an existing sugar library is easy: (1) place the sugar library on the classpath, (2) `import` the sugar library and use its features to improve code, (3) rename the former `.java` file to a `.sugj` file, and (4) compile the file with `sugarjc` instead of `javac`.

3.2 Writing a sugar library

There are three aspects to every programming language: syntax, semantics and pragmatics. This naturally extends to sugar libraries.

3.2.1 Pair syntax

SugarJ uses the *SDF syntax definition formalism* [9] to describe the syntax that sugar libraries introduce.

In the case of the pair sugar library, we extend the Java syntax using the SDF grammar shown in Listing 3.2, below.

```
package pair;

import org.sugarj.languages.Java;

public sugar Sugar {
  context-free syntax
    "(" JavaType "," JavaType ")" -> JavaType {cons("PType")}
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr {cons("PExpr")}
}
```

Listing 3.2: Pair syntax

Like traditional class libraries, sugar libraries are organised in packages and the pair sugar library is in `package pair`. The `import` statement allows access to the Java syntax this sugar library extends. A sugar library is defined syntactically similar to a class, with `sugar` replacing `class`, so the name of this sugar library is `Sugar`, fully qualified: `pair.Sugar`.

`context-free syntax` marks the beginning of the syntax definition. Sugar libraries may use any context-free syntax. Productions are, confusingly, written "the wrong way 'round". The single non-terminal is written to the right-hand side of the arrow `->`, followed by annotations enclosed in curly braces. The left hand side denotes what sequence of terminals and non-terminals the non-terminal on the right hand side may be expanded to.

The first line of the above syntax definition would be written like the following in the probably more familiar Backus-Naur Form:
JavaType ::= "(" JavaType "," JavaType ")"

Conceptually, the pair sugar library's syntax allows its users to write two Java types, enclosed by parentheses and separated by a comma, in the place of one, the same goes for Java expressions.

The annotation {`cons("PType")`} causes the parser to introduce a node labeled "PType" to the abstract syntax tree when using this particular production.

3.2.2 Pair desugaring

Languages defined in sugar libraries derive their semantics from *desugarings*. Desugarings are transformations of the abstract syntax tree produced by the parser. As such, they are a mapping from the syntactic constructs of the embedded language to constructs in the host language, most often Java. Desugarings to more general SugarJ code, that is grammar definitions, transformations, and code that uses other sugar libraries, are also possible. In the end, the semantics of a language in a sugar library is the semantics of the desugared code.

We would like the code from Listing 3.1, to desugar to the following Java code:

```
public class Test {
  private pair.Pair<String, Integer> p =
      pair.Pair.create("Answer", 42);
}
```

Listing 3.3: Code from Listing 3.2, desugared

Speaking in concrete syntax, we want to transform the type declaration `(T1, T2)` to `pair.Pair<T1, T2>`, similarly we need to transform the pair expression `(E1, E2)` to `pair.Pair.create(E1, E2)`, for every `T1`, `T2`, `E1`, `E2`.

The `desugarings` block in a sugar library definition declares the names of the transformation rules the SugarJ compiler will apply exhaustively to the syntax tree to achieve the desugaring result. The pair sugar library declares two desugaring rules, one for types and one for expressions:

```
...
public sugar Sugar {
  ...

  desugarings
    desugar-pair-type
    desugar-pair-expr
}
```

SugarJ transformations are written in Stratego [15], a language designed for program transformation. Stratego uses two kinds of "functions" called rules and strategies. Rules use pattern matching over the abstract syntax tree and are usually used to transform one specific node of an abstract syntax tree to another. The rules that define the pair desugaring are shown below:

```
...
public sugar Sugar {
  ...

  rules
    desugar-pair-type :
      PType(t1, t2) -> |[ pair.Pair<~t1, ~t2> ]|

    desugar-pair-expr :
      PExpr(e1, e2) -> |[ pair.Pair.create(~e1, ~e2) ]|
}
```

On the left hand side of the arrow `->` is the pattern a rule matches on, the right hand side is a new abstract syntax tree. `desugar-pair-type` can only be applied to a node of type `PType` with two children, which are bound to variables `t1` and `t2`. The brackets `|[` and `]|` that surround the right hand side expressions in the above code allow us to use concrete Java syntax to describe the result of applying this rule. Without this, `desugar-pair-type` would be a little more verbose but it gives an impression of what Java abstract syntax trees look like:

```
desugar-pair-type:
  PType(t1, t2) ->
    ClassOrInterfaceType(
      TypeName(PackageOrTypeName(Id("pair")), Id("Pair")),
               Some(TypeArgs([t1, t2])))
```

We mentioned Stratego strategies. Strategies are more general than rules, and in fact rules are special-purpose strategies with domain-specific syntax. Strategies are mostly used like higher-order functions in functional programming languages. There are strategies with familiar functions like `map` and `fold` to map a rule over a list, or to aggregate a list. There are also strategies like `bottom-up` and `top-down` that define tree traversals in a generic way.

Stratego's more advanced features make it very powerful tool in its domain of program transformation. However, it can also be used to great effect in a very restricted form by anyone who ever programmed in any functional language using only pattern matching and simple, higher-order function-like strategies.

3.3 Editor services

Through editor libraries, SugarJ provides access to the editing environment [7]. We used this functionality only sparingly for the JPQL sugar library.[1]

Sugarclipse, the SugarJ Eclipse plugin that implements the editor services, is work in progress but its features already include syntax highlighting, code folding, cross references, edit-time error reporting, outline view, and semantic code completion.

Editor services are based on decorated abstract syntax trees and are therefore very robust. Especially in the presence of nested embedded languages this approach excels because there is no need for additional code to support different combinations of sugar libraries.

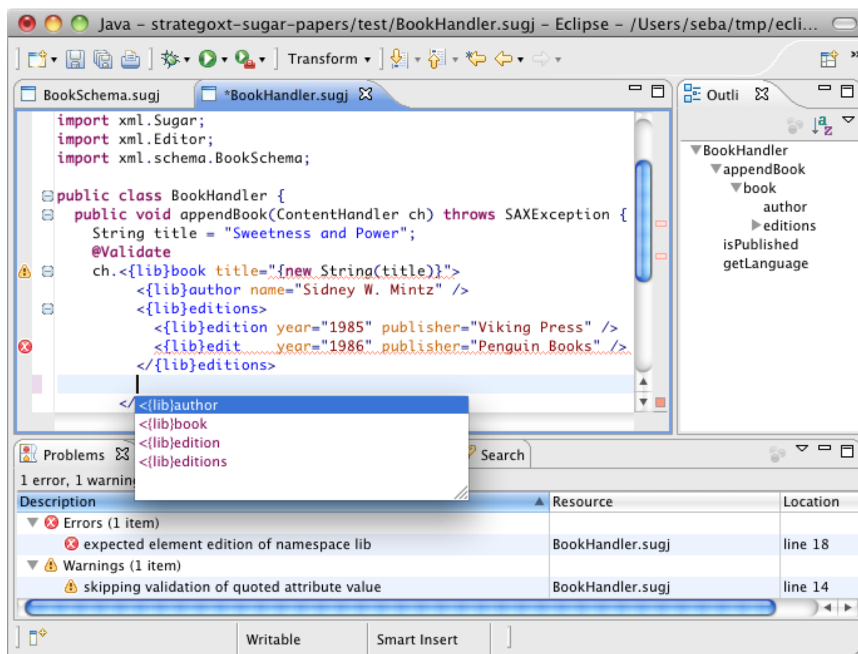

Figure 3.1: Sugarclipse, the SugarJ IDE (source: sugarj.org)

[1] see Section 5

4. Retrofitting Language-oriented Design: A Case Study

To evaluate the process for incrementally retrofitting a legacy code base with language-oriented design we proposed in Section 2, we applied it to Java Pet Store.

We identified several aspects of the Java Pet Store code that pose problems for maintenance and future extension and developed sugar libraries that address these concerns. In this section, we showcase the application of these libraries to the Java Pet Store and therby demonstrate, how sugar libraries improve conciseness, editing experience, and static safety. The sugar libraries' technical realisations are discussed in Section 5.

4.1 JavaBeans style properties

The *JavaBeans* standard is a coding convention that enables uniform object creation and access to fields, called properties in this context. JavaBeans are ordinary classes that are `Serializable`, provide a nullary constructor, and provide getters and setters for their fields that follow a common naming scheme.

The original goal of the JavaBeans standard was to build visual tools to create and modify GUI components, like buttons and labels. Naming accessors and mutators according to the JavaBeans standard, using the prefixes `get` or `is`, and `set`, respectively, is widely adopted in Java code, not only among GUI toolkits.

The Java Pet Store uses classes with JavaBean style accessors for its business logic entities, some of which have quite a few properties, resulting in correspondingly many lines of boilerplate code for accessors. The class `Item`, for example, represents a pet, and consists of 13 fields like ID and description. The resulting boilerplate fills many-a-page; the five business methods that are actually interesting, are well hidden in between.

4.1.1 Language support for properties

Many programming languages provide special support for properties, that reduces the amount of boilerplate code needed.

Java

```java
public class Person {
  private String name;
  private boolean adult;

  public Person() {}

  public Person(String name,
                boolean adult) {
    this.name = name;
    this.adult = adult;
  }

  public String getName() {
    return this.name;
  }

  public void
  setName(String name) {
    this.name = name;
  }

  public boolean isAdult() {
    return this.adult;
  }
}
```

C#

```csharp
public class Person {
  public string Name {get; set;}
  public bool Adult {get;
                 private set;}

  public Person() {}

  public Person(string name,
             bool adult) {
    set_Name(name);
    set_Adult(adult);
  }
}
```

Common Lisp

```lisp
(defclass person ()
  ((name :initarg :name
         :accessor person-name)
   (adult :initarg :adult
         :reader person-adult)))
```

Listing 4.1: A person modelled in different programming languages

Consider an exemplary person entity, that has two properties: a name, and whether the person is an adult; the latter property shall be read-only.

The code in Listing 4.1 shows, how we would model a person in Java, Common Lisp, and C#. Java has no support for properties, so we use ordinary methods for getters and setters. In both Java and C#, we also declare two constructors, one that takes no arguments, as required by the JavaBeans standard, and one that initialises the fields with its parameters. In C# we do not need to write getter and setter methods, the shorthand syntax `{get; set;}` makes the compiler generate those automatically. In Common Lisp we can even omit the constructor implementation, the `:initarg` option

makes the generated constructor accept an additional optional argument for initialisation.

4.1.2 The accessors sugar library

Since the Java Pet Store's business entities make heavy use of JavaBeans style properties, we would like our language to provide us with more support.

For retrofitting Java with accessors support through a sugar library, we took inspiration from both C# and Common Lisp. The syntax is adapted from C#'s, since it fits nicely with Java syntax. From Common Lisp, we adapted the idea of generating a constructor to initialise fields.

```
import sugar.Accessors;

public class Person {
  private boolean adult {con; get};
  private String name {con; get; set};

  public Person() {}
}
```

Listing 4.2: `Person` using the accessors sugar library

Listing 4.2 shows the example class `Person` from before, this time using the accessors sugar library. After importing the sugar library in the first line, a field declaration may be followed by a combination of one or more of `get`, `set` and `con`, indicating whether getter and setter should be created, and whether this field should be part of the initialising constructor. The code resulting from desugaring, is the same as the Java code in Listing 4.1.

4.1.3 The accessors sugar library applied to the Java Pet Store

The `Item` class is a typical example of a business logic entity in the Java Pet Store that follows the JavaBeans conventions. Most of its code is boilerplate; there are 9 getters and 13 setters, which just get or set without any validation or computation, as well as two constructors, one nullary and one 10-ary, which initialise an `Item` with `null`ed fields, and the constructor's arguments' values, respectively. In addition, there are four getters with special annotations for persistence, and five methods that actually do something interesting.

The five business methods fill one screen page, the overall size is about five pages. We used the accessors sugar library to generate the 22 accessors and the initialising constructor from the field declarations in Figure 4.3.

The result is only two pages of code, one of which is the untouched business methods, the other is comprised of required imports, field declarations, and getters specifically annotated for persistence purposes.

```
private String itemID {set};
private String productID, name, description,
               imageURL, imageThumbURL {con; get; set};
private BigDecimal price {con; get; set};
private Address address {con; set};
private SellerContactInfo contactInfo {con; set};
private int totalScore, numberOfVotes {con; get; set};
private int disabled {get; set};
private Collection<Tag> tags = new Vector<Tag>();
```

Listing 4.3: Field declarations with accessor annotations for the class `Item`

To summarise, the accessors sugar library captures a pattern commonly found in Java code. By reducing the amount of boilerplate code, it helps the programmer focus on code that is actually important.

4.2 XML

As a web application, the Java Pet Store naturally deals with XML data. The largest part is mostly static HTML, hidden in Java Server Pages. SugarJ support for Java Server Pages is an ongoing project, but it is not discussed here any further.

In addition to its graphical user interface that relies on HTML, the Java Pet Store also uses XML as one of its data interchange formats for communication between the user interface, running in the browser, and the back end, running atop an application server.

The code in Figure 4.1 for example, returns an XML representation of an item in the Java Pet Store.

```
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(" <cat-id>" + i.getProductID() + "</cat-id>\n");
    sb.append(" <name>" + i.getName() + "</name>\n");
    sb.append(" <description><![CDATA[" + i.getDescription() + "]]></description>\n");
    sb.append(" <image-url>" + i.getImageURL() + "</image-url>\n");
    sb.append(" <price>" + NumberFormat.getCurrencyInstance(Locale.US).format(i.getPrice())  + "</price>\n");
    sb.append("</item>\n");
    return sb.toString();
}
```

Figure 4.1: Use of XML strings in the Java Pet Store

4.2.1 Language support for XML

This style of language embedding, through strings, is prone to errors. This is especially ironic considering, that XML was designed for interoperability purposes, and with a focus on static validation.

A real world web application likely needs to interface with external sources, or consumers, of XML data. As features are added and interchange formats change, the application's XML emitting code has to be adapted. To avoid mistakes, we would like our language to do as much static checking as possible; to help programmers, editor support is desirable.

Most programming languages have libraries to deal with XML. XML libraries prevent syntactic mistakes, some even offer validation. However, traditional, class based libraries, like JDOM for Java, have one inherent flaw: their syntax is the syntax of objects, not XML.

Scala has syntactic support for XML built-in and checks, at least, for basic well-formedness like proper nesting. A sugar library to retrofit Java with XML support, should offer at least this much.

In the spirit of code reuse,[1] we adapted an existing XML sugar library [7, 8] to desugar literal XML to a string representation. This approach combines minimal involvement in sugar library development, with as little adaption of legacy code in the Java Pet Store as possible. In spite of this minimum in effort, it still offers many benefits, like static checking for well-formed tags, general syntactic safety, editor support in the form of syntax highlighting, code folding, and integration with Eclipse's overview feature.

4.2.2 The XML sugar library

After importing the sugar library, the `handleItem` method from Figure 4.1, can be rewritten as shown in Figure 4.2.

```
private String handleItem(String targetId){
    Item i = cf.getItem(targetId);
    return  <item>
                <id>${i.getItemID()}</id>
                <cat-id>${i.getProductID()}</cat-id>
                <name>${i.getName()}</name>
                <description>${"<![CDATA[" + i.getDescription() + "]]>"}</description>
                <image-url>${i.getImageURL()}</image-url>
                <price>${NumberFormat.getCurrencyInstance(Locale.US).format(i.getPrice())}</price>
            </item>;
}
```

Figure 4.2: Method from figure 4.1 using the XML sugar library with string desugaring

The Java Pet Store, as it is today, uses only a small subset of XML functionality. In the future, it will likely be required to be extended with more advanced use of XML.[2]

We see the employment of the XML sugar library, in this basic capacity, as a first step of a future extension of the Java Pet Store. The static safety it provides today, protects against malformed XML and basic syntactic

[1] read "avoiding work"

[2] The Java Pet Store will most likely never change again, but remember, it is a model for a real application.

errors, and thereby builds a foundation for easy and safe extension, like unit tests do. Validation against predefined schemata promises an even greater increase in static safety, when needed, in the future.

4.3 Java Persistence Query Language

The *Java Persistence API* is a framework for managing relational data in Java applications. It is comparable to the well known object-relational mapping library Hibernate,[3] but the mapping is more direct, as usually every persistence entity corresponds closely to a single row in a single table.

The Java Persistence API includes a query language, named *Java Persistence Query Language (JPQL)*,[4] that is syntactically and semantically closely related to SQL, but operates on persistence entities instead of relational database tables.

The syntax and semantics of JPQL are defined in JSR 220 [12], an informal overview is part of the Java 5 EE Tutorial [5]. To follow the code samples below, it suffices to know that to retrieve persisted entities, a programmer creates a `Query` object by passing a JPQL query string to the factory method `createQuery`, that is provided by an `EntityManager`. Also, queries may contain named placeholders, called query parameters, prefixed with a colon, that can be instantiated with the `setParameter` method.

The Java Pet Store uses the Java Persistence API to persist its business entities, like pets, tags, categories and addresses. The Java Persistence Query Language is used to retrieve entities, possibly filtered for some property, e.g., to present a user with a list of all panda bear guys. See Listing 4.4 for a typical, albeit short, query from the Java Pet Store code.

```
public List<Item>
getItemsVLH(String pID, int start, int chunkSize) {
    EntityManager em = emf.createEntityManager();
    Query query = em.createQuery(
      "SELECT i FROM Item i WHERE i.productID = :pID AND i.disabled
= 0");
    List<Item> items = query.setParameter("pID",pID)
      .setFirstResult(start).setMaxResults(chunkSize)
      .getResultList();
    em.close();
    return items;
}
```

Listing 4.4: Example JPQL query that retrieves enabled items of a certain kind. The query parameter `:pID` is set at runtime using `setParameter`

[3] hibernate.org – Hibernate implements the Java Persistence API since Version 3.2

[4] based on the Hibernate Query Language

4.3.1 Problems with JPQL

As briefly mentioned before, JPQL queries are embedded into Java as plain strings. This form of embedding has some disadvantages compared to a more direct embedding in the host language:

While editor support is not impossible, JPQL query strings could be detected and treated differently from other strings, none is provided by any IDE, as of today. This means, most visibly, no syntax highlighting but also no outline view, reference resolving, code completion and similar features programmers have come to expect from their editing environment.

String embedding is naturally constrained by the host languages strings. Escaping of special characters over multiple language levels can become hard to deal with.

Since Java provides no way to do arbitrary compile time computation, any static analysis and error checking for string embedded languages is impossible. For the Java Pet Store, we therefore need to do a full compile, deploy to the application server, and run-cycle, to find a missing comma or a misspelled query parameter. Further static analysis is conceivable, since the operands of JPQL queries are Java objects that belong to Java classes, elaborate static type checking would certainly be possible, yielding great benefits in static safety.

The JPQL sugar library developed in the context of this thesis addresses many of these problems and requires only a minimum of changes to the legacy code.

4.3.2 Host language integration

After importing the JPQL sugar library, JPQL queries are a syntactically first class language extension and appear to be messages[5] to an `EntityManager`. There is no need for elaborate means to wrap lines, or escape literal strings in queries.

With the JPQL sugar library, the query from Listing 4.4 can be written as seen in Listing 4.5. The code still follows the same overall structure.

The attentive reader might have noticed, that the call to `setParameter` is missing. In traditional JPQL query strings, the colon is used to mark query parameters, to be set later. With the JPQL sugar libraries queries, it can be thought of as an operator to change scopes; it allows access to variables in the Java lexical environment from within a query. Besides being more concise, this prevents premature query execution, where not all parameters have been set, a mistake previously only discovered at runtime.

```
public List<Item>
getItemsVLH(String pID, int start, int chunkSize) {
```

[5]as in Smalltalk-style object-oriented programming with message passing

```
  EntityManager em = emf.createEntityManager();
  Query query = em.SELECT i
                  FROM Item i
                  WHERE i.productID = :pID
                    AND i.disabled = 0;
  List<Item> items = query.setFirstResult(start)
    .setMaxResults(chunkSize).getResultList();
  em.close();
  return items;
}
```

Listing 4.5: Query from Figure 4.4 using the sugar library. The query is not a string anymore and can be freely wrapped, also note the missing `setParameter`

4.3.3 Editor support

Editor support, as detailed before, for string embedded domain-specific languages is hard and rarely done. A modern editing environment provides its users not only with keyword highlighting and automatic insertion of closing parenthesis. IDEs like Eclipse provide, for Java, generation of overviews, code folding, context sensitive code completion, indentation support, and reference resolving.

```
public List<Item> getItemsByCategoryByRadiusVLH(String catID, int start,
        int chunkSize,double fromLat,double toLat,double fromLong,
        double toLong){
    EntityManager em = emf.createEntityManager();
    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);
    List<Item>  items = query.setFirstResult(start).setMaxResults(chunkSize).getResultList();
    em.close();
    return items;
}
```

Figure 4.3: Eclipse default Java editor showing a JPQL query

As you can see in Figure 4.3, the JPQL query, in a section of original Java Pet Store code displayed in the Eclipse IDE, stands out as a long string literal coloured blue, like every other string literal independent of content. There is absolutely no editor support for queries. We do not have any evidence for this claim, but we are certain this query is not formatted as the author initially imagined it to be. One explanation for the curious distribution

of space characters at the beginning and end of lines, as well as the non-idiomatic line break and keyword positions might be that Java disallows string literals to be broken over multiple lines, which makes adjusting indentation after modification difficult.

The JPQL sugar library addresses this problem and allows queries to be laid out as their writers desire. As you can see in Figure 4.4, using the JPQL sugar library, queries are no longer bound by the shortcomings of Java string literals so programmers can wrap and indent as is natural for SQL-like code, which leads to more readable code.

```
public List<Item> getItemsByCategoryByRadiusVLH(String catID, int start,
        int chunkSize,double fromLat,double toLat,double fromLong,
        double toLong){
    EntityManager em = emf.createEntityManager();
    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;
    List<Item> items = query.setFirstResult(start).setMaxResults(chunkSize).getResultList();
    em.close();
    return items;
}
```

Figure 4.4: Same query as in Figure 4.3, using the JPQL sugar library

This screenshot also shows basic syntax highlighting. JPQL keywords, e.g., SELECT, are coloured like Java keywords and input variables are displayed in an unintrusive shade of green that intensifies the visual hint that is already provided by the colon prefix.

While syntax highlighting is certainly important, programmers have come to expect more sophisticated support from their editing environment. To this end, the JPQL editor library already provides basic code completion templates, see Figure 4.5. In the future we might see extensive semantic code completion, leveraging the Java type system to filter applicable JPQL functions and provide likely join attribute candidates. Reference resolving, that is linking query parameters back to their declaration, might also be a useful future extension.

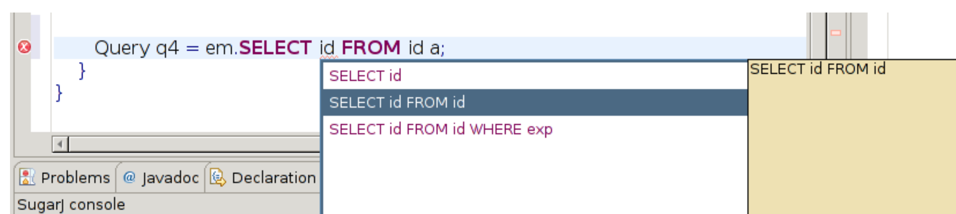

Figure 4.5: Code completion for JPQL queries

4.3.4 Fully parsed queries

JPQL queries are parsed using a grammar based on the documentation [4] provided by Oracle. For technical background on this see Section 5.3.

Having programs fully parsed is a vast improvement over string embedding. Syntactic errors are detected at compile-time[6] instead of at runtime, where they can easily be missed, especially without an extensive test suite. Furthermore, error reporting output from test suites, is unlikely to be as immediate as the errors reported directly in the IDE, as seen in Figures 4.6 and 4.7. SugarJ applies error recovery parsing [11] to provide error messages of high quality.

```
em.SELECT c FROM Category c WERE c.id = 5
```

Figure 4.6: Misspelt `WHERE`. Formerly a runtime error, now highlighted in the IDE

Figure 4.7: Missing comma between `Category c` and `Product p` with possible correction in a tooltip

While adapting the Java Pet Store to use the JPQL sugar library, we found what is probably a mistake in the documentation, an inconsistency between the implementation of the Java Persistence Query Language and its specification. According to Oracle's documentation the following query is syntactically incorrect, because the pattern, that follows the keyword `LIKE`, is supposed to be a literal string:

```
SELECT p FROM Product p
WHERE p.categoryID LIKE :categoryID
```

The relevant production in the grammar is this one:

```
like_expression ::=
    string_expression [NOT] LIKE pattern_value
    [ESCAPE escape_character]
```

The documentation elaborates: "The pattern value is a string literal that can contain wildcard characters",[7] but `:categoryID` is not a string literal, it is an input variable.

[6] or type-time when using Sugarclipse, the SugarJ plugin for Eclipse

[7] http://download.oracle.com/javaee/5/tutorial/doc/bnbuf.html#bnbvg

This particular pattern, `LIKE` followed by an input variable, occurs quite often throughout the Java Pet Store. It works just fine, and there is no reason it should not work, other than that it is undocumented behaviour.

We assume it is a mistake in the documentation, one that is easily fixed by introducing a production to the grammar that describes a pattern value as either a string literal or an input variable. We do not mention this to criticise the Java Pet Store's developers or Sun's documentation writers. The point is that using SugarJ, this use of undocumented behaviour was very easy to spot and could have been avoided from the start.

4.3.5 Static checks

Most static checks for SQL queries are nearly impossible, because the database is an external service and its layout can change from compile-time to runtime and even in between queries. In an ideal world, we would like the query compiler do every static check possible so that the database management system only ever needs to report runtime errors.

As a proof of concept for static analysis, we implemented name resolution for identification variables.[8] Identification variables are far more important in JPQL, than their counterparts are in SQL, because fields, or columns, are not implicitly resolved even if they are uniquely named.

In the following query, `i` is an identification variable:

```
SELECT i FROM Item i WHERE i.id = 42
```

Identification variables are declared in the `FROM`-clause and this is already enforced by the parser because it is required in the grammar. Their proper use, however, is a context-sensitive property which is impossible to enforce by a context-free grammar. Nevertheless, Java programmers are used to being alerted by their IDE, as seen in Figure 4.8, when they accidentally use undeclared variables.

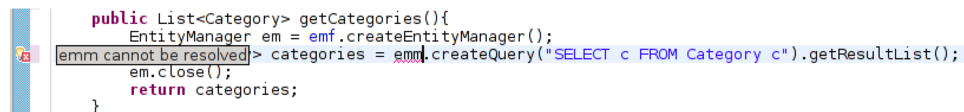

Figure 4.8: Eclipse highlighting the use of an undeclared variable

The JPQL sugar library's editor services provide undeclared variable detection and reporting for JPQL queries. For the sake of consistency the visual presentation is just the same for JPQL as it is for Java, as you can see in Figure 4.9.

In this section we demonstrated what problems can be avoided by using sugar libraries. We captured a pattern commonly found in Java code in a

[8]known as tuple variables in the SQL world

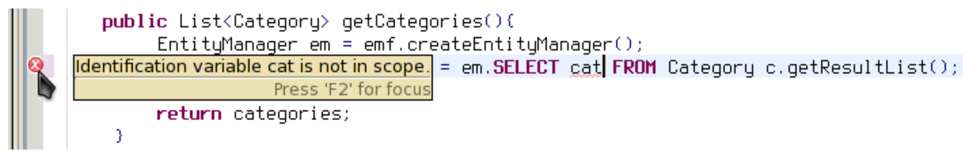

Figure 4.9: Sugarclipse highlighting the use of an undeclared identification variable

sugar library and thereby reduced the amount of boilerplate code. For both XML and JPQL we eliminated the shortcomings of string embedding and provide syntactic safety as well as an improved editing experience.

5. Technical realisation

In the previous section, we demonstrated the effects of various sugar libraries on the Java Pet Store, leading to leaner, clearer and safer code. In this section we discuss the technical realisation of the sugar libraries used in the previous section.

We describe these language extensions' syntax definitions, highlight interesting aspects of their desugarings, and, where applicable, discuss pragmatic aspects, such as editor support.

5.1 Accessors

The accessors sugar library extends Java with annotations to field declarations that desugar to traditional JavaBean style getters and setters. A constructor to initialise selected fields will also be generated when requested.

```
class Foo {
  private int bar, baz {con; get; set};
  private Boolean quux {con; get};
}
```

Listing 5.1: Example of a class using the accessors sugar library

5.1.1 Syntax

After having decided what syntax to use, the implementation of the syntax in SugarJ is not much more than just writing it down.

From the minimal example in Listing 5.1 we gather that we need to extend the way fields are declared in Java with a semicolon-separated list of annotations. The annotations being one of `con`, `get`, and `set`, for a field marked to be initialised by the constructor, or to generate a getter, or setter.

Figure 5.2 shows the syntax definition of the accessors sugar library. Line 5 is essentially a Java field declaration plus `AccDecs`, which are the accessors annotations. Lines 6 to 9 define the annotations' format. Lines 2 to 4 are used to introduce a custom node in the syntax tree at class declaration level.

We need this later, to get at the class's name, which is also the name of the constructor we need to generate.

```
1 context-free syntax
2   JavaClassDecHead AccClassBody -> AccClassDec
      {cons("AccClassDec")}
3   "{" AccClassBodyDec* "}" -> AccClassBody
      {cons("AccClassBody")}
4   AccFieldDec | JavaClassBodyDec -> AccClassBodyDec
5   (JavaAnno | JavaFieldMod)* JavaType {JavaVarDec ","}+
      AccDecs ";" -> AccFieldDec {cons("AccFieldDec")}
6   "{" {AccDec ";"}+ "}" -> AccDecs
7   "con" -> AccDec {cons("AccCon")}
8   "get" -> AccDec {cons("AccGet")}
9   "set" -> AccDec {cons("AccSet")}
```

Listing 5.2: Syntax definition of the accessors sugar library

The syntax definition is, overall, rather compact and straightforward. This has two reasons: the amount of syntax newly introduced is quite small, and SugarJ allows us to reuse the productions, prefixed with "Java", defining the original Java syntax.

5.1.2 Desugaring

Desugaring in this case, as in many cases, is a straightforward transformation from one abstract syntax tree into another.

The implementation of desugarings, especially for these macro-like extensions, can often be divided into three steps:

1. extract relevant parts of the original abstract syntax tree
2. normalise the representation
3. generate code using the normalised data.

The strategy `genGetters` is the entry point for getter generation, and receives a list of method, constructor, field, and annotated field declarations as its implicit argument. It follows the aforementioned design pattern closely.

```
genGetters = filter(?AccFieldDec(_, _, _, _)) ;
             mapconcat(getterTriples) ; map(getter)
```

Listing 5.3: Three steps to generating getters

According to the first item in the above pattern, `genGetters` extracts the relevant declarations by applying a filter that matches only field declarations with accessor annotations.

To understand normalisation it is helpful to discuss code generation first. The typical getter looks like this: `public Type getField() { return field; }`, so for code generation we need the field's type, name and the prefix, which is usually "get", but is "is" for boolean types. Let us assume that normalisation provides us with exactly this data for every getter that is to be generated, a triple consisting of type, name, and prefix. Code generation just maps the rule seen in Listing 5.4 over a list of triples and thereby returns a list of getter ASTs.

```
getter : (type, name, prefix) ->
  MethodDec(MethodDecHead([Public()], None(), type,
    Id(<conc-strings> (prefix, <upcaseFirstChar> name)),
    [], None()), Block([Return(Some(ExprName(Id(name))))]))
```

Listing 5.4: Code generation basically returns the AST of `public type prefixName() {return name;}` for given `type`, `name`, and `prefix`

The second step is normalisation, the glue between the filter step, which is syntax driven, and the code generation, which is data driven. Normalisation is usually a bit more involved because it, naturally, needs to deal with non-normalised data. The Java syntax also allows for more than one field to be declared with the same declaration by seperating multiple field names by commas. Those fields then share their type, modifiers, and, when using this sugar library, their accessor annotations. Normalisation basically needs to do three things: filter out fields that shall not receive a getter, flatten the list of lists of fields that is a result of the multiple field syntax, and return the appropriate getter prefix for boolean versus non boolean types.

```
getterTriples = ?AccFieldDec(_, type, vars, accs);
  if <elem> (AccGet(), accs)
  then !vars; map(\VarDec(Id(name)) ->
                 (type, name, <defaultPrefix> type)\)
  else ![]
  end
```

Listing 5.5: Normalisation code returns lists of (type, name, prefix) triples that are concatenated later on

The first line of the normalisation code in Listing 5.5 binds parts of the field declaration with accessors, namely the type, the names of the variables, and their declared accessors. It proceeds to check whether a getter is to be generated at all and if so, returns a list of triples containing all the data relevant for code generation.

The appropriate prefix is determined by pattern matching on the field's type as seen in the code below, where <+ is ordered choice to prefer the boolean rules over the generic rule.

```
defaultPrefixBool : Boolean() -> "is"
defaultPrefixBool : ClassOrInterfaceType(
   TypeName(Id("Boolean"))) -> "is"
defaultPrefixOther : _ -> "get"
defaultPrefix = defaultPrefixBool <+ defaultPrefixOther
```

Desugaring of actual field declarations, setters and the constructor follows the same overall pattern of extracting, normalising, and then using the normalised data for straight forward code generation.

The accessors sugar library also features custom prefixes. In retrospect, this appears to be a case of premature generalisation, as they are never used in the Java Pet Store and it is not clear that we would ever want to use them, especially since "get" is exchanged for "is" for boolean types automatically. Custom prefixes complicate normalisation and have been omitted from this discussion for clarity.

The entire accessors sugar library definition fits narrowly on one screen page and could be made even more compact by making better use of Stratego's standard library. More importantly, it is very easy to understand. Every programmer who ever used one of Erlang, F#, Haskell, some variant of ML, Prolog, Scala or Scheme is familiar with pattern matching and the basic higher order functions used in the implementation.

5.2 XML

The XML sugar library we used on the Java Pet Store is largely based one written by Sebastian Erdweg[8, 7]. This original XML sugar library is combined from several smaller independent parts: the XML syntax, an integration of XML in Java statements, a desugaring to SAX calls, and editor services.

The Java Pet Store does not use SAX and as one of our goals is to remain true to the original design to minimise rewriting effort, we replace the SAX desugaring by one that is closer to the Java Pet Store's XML string embedding.

5.2.1 Syntax

The fact that we desugar to string expressions, not SAX statements as in the original, implies we change the Java integration to allow XML expressions, instead of statements, as is the case in the original XML sugar library. SugarJ enables us to reuse the entire Java syntax and the XML syntax defined in the original XML sugar library by just `import`ing them. The syntax definition that allows XML in place of Java expressions and introduces an unquote operation is thereby reduced to the following four lines:

```
context-free syntax
  Document -> JavaExpr {cons("XMLDocument")}
  "${" JavaExpr "}" -> JavaEscape {cons("JavaEscape"), prefer}
  JavaEscape -> Element
```

Listing 5.6: Integration of the XML syntax with Java

This makes the following code fragment syntactically legal:

```
return <item>
        <id>${i.getItemID()}</id>
        <cat-id>${i.getProductID()}</cat-id>
       </item>;
```

Listing 5.7: XML document using the sugar library

5.2.2 Desugaring

The Java Pet Store in its original form uses string encoded XML documents to respond to requests from the browser. It uses a `StringBuffer` to gradually compose an XML document from interleaved XML fragments and string-typed Java expressions.

```
StringBuffer sb = new StringBuffer();
sb.append("<item>\n");
sb.append(" <id>" + i.getItemID() + "</id>\n");
sb.append(" <cat-id>" + i.getProductID() + "</cat-id>\n");
sb.append("</item>");
return sb.toString();
```

Listing 5.8: Typical Java Pet Store code for generating XML documents

Listing 5.8 shows how XML is generated throughout the Java Pet Store. For the XML sugar library, we chose to desugar to `String.format` calls, as seen in Listing 5.9 below.

```
return String.format(
  "<item><id>%s</id><cat-id>%s</cat-id></item>",
   i.getItemID(), i.getProductID());
```

Listing 5.9: XML code from Listing 5.7 desugared

This desugaring is semantically very close to the original Java Pet Store code we would like to replace, minimising the amount of local rewriting needed to benefit from the sugar library.

The desugaring itself is a straightforward pretty printing of the XML AST to a string representation with every unquote replaced by the `%s` format

directive, which is filled out with the arguments to `format`. An excerpt of the pretty printing code is shown in Listing 5.10. This is very similar to the desugaring for the Java Persistence Query Language.

```
xml-to-string = pprint <+ arglist; map(xml-to-string); concat-
    strings <+ !""
pprint : Element(ElemName(QName(None(), open_name)),
    attributes, body,
ElemName(QName(None(), close_name)))
    -> <concat-strings> ["<", open_name, <xml-to-string>
        attributes, ">", <xml-to-string> body, "</",
        close_name, ">"]
pprint : Attribute(AttrName(QName(None(), aname)), content)
    -> <concat-strings> [" ", aname, "=", <xml-to-string>
    content]
pprint : DoubleQuoted(any)
    -> <concat-strings>["\\\"", <xml-to-string> any, "\\\""]
pprint : CharDataPart(s) -> s
pprint : JavaEscape(_) -> "%s"
```

Listing 5.10: Excerpt of the XML pretty printing code

5.2.3 Editor support

Beneficially, all editor services are provided by the original XML sugar library. This includes syntax colouring, error reporting, content completion, code folding and outline view. No adaption was necessary because SugarJ editor services are based on decorated abstract syntax trees and we reuse the original XML sugar library's XML syntax and parsed representation.

5.3 BNF

Backus-Naur Form (BNF) was first used by John Backus to describe the syntax of ALGOL in 1959. It has since become the most widely used notation (with many flavours and dialects) to describe the context-free part of programming languages' syntax.

The most complex part of the JPQL sugar library is parsing the language. The context free grammar that Oracle provides [4] for documentation, is over 200 lines for about 70 productions. Unfortunately, this grammar is in Backus-Naur Form and SugarJ uses the Scannerless GLR parser, which in turn uses SDF, not BNF, for its syntax descriptions.

Fortunately, SugarJ is self-applicable, meaning it is possible to extend SugarJ through sugar libraries. Domain-specific languages that aim to improve on writing other domain-specific languages are called meta-DSLs.

In our course of implementing the JPQL sugar library, we decided against translating the JPQL grammar from BNF to SDF by hand. Instead, we chose to take advantage of SugarJ's self-applicability by writing a meta-DSL that allows us to reuse the JPQL grammar without translation or modification.[1]

From this effort springs a new sugar library that allows SugarJ developers to use BNF for describing their language's syntax, instead of, or in combination with, the usual way of language description in SugarJ, SDF.

5.3.1 Syntax

There are several dialects of BNF, among them *Extended BNF* and *Augmented BNF*, that are somewhat well specified or even standardised. In addition, there are countless special-purpose dialects that are only informally defined, if at all, and are only used for single grammars.

Oracle defined their own BNF dialect and describe it in the documentation containing the JPQL grammar. The BNF sugar library uses Oracles BNF dialect, it is however easily adaptable for other dialects.

```
context-free syntax
  BnfSort "::=" BnfTNT -> BnfProduction {cons("BnfProduction")}
  "\"" BnfLit "\"" -> BnfTNT {cons("BnfLiteral")}
  BnfSort -> BnfTNT {cons("BnfSort")}
  BnfTNT "*" -> BnfTNT {cons("BnfStar")}
  "{" BnfTNT "}" -> BnfTNT {cons("BnfGroup")}
  BnfTNT BnfTNT -> BnfTNT {right, cons("BnfSeq")}
  "[" BnfTNT "]" -> BnfTNT {cons("BnfOptional")}
  BnfTNT "|" BnfTNT -> BnfTNT {right, cons("BnfAlternative")}

context-free priorities
    BnfTNT "*" -> BnfTNT {cons("BnfStar")}
  > BnfTNT BnfTNT -> BnfTNT {cons("BnfSeq")}
  > BnfTNT "|" BnfTNT -> BnfTNT {right, cons("BnfAlternative")}
```

Listing 5.11: SDF syntax definition for Oracles flavour of Backus-Naur Form

The specification of BNF's syntax is, for bootstrapping reasons, written in SDF, an excerpt can be seen in Listing 5.11. SDF's disambiguation features, like declaring left or right associativity and ordering productions to show their priorities, make the definition very concise.

A new SugarJ block header (`bnf syntax`) was introduced to prelude a syntax description in BNF, much like SDF syntax blocks are preceded by `context-free syntax`. This allows us to define new syntax with BNF directly in SugarJ files, as you can see in Listing 5.12, which shows the `IN` expression as defined in the JPQL grammar.

[1] some modification was still necessary, to account for ambiguities, see the next section

```
bnf syntax
  in_expression ::= path_expression ["NOT"] "IN"
     "(" {in_item {"," in_item}* | subquery} ")"
```

Listing 5.12: The BNF meta-DSL allows programmers to use BNF to describe their sugar library's syntax

5.3.2 Desugaring

The mapping from BNF to SDF is quite direct. All operations on context-free languages like the Kleene star, alternatives or sequences are present in either formalism. This direct mapping translates to the desugaring definition in Stratego:

```
bnf2sdf : BnfSort(s) -> sort(<camel-case> s)
bnf2sdf : BnfStar(s) -> iter-star(s)
bnf2sdf : BnfGroup(s) -> s
bnf2sdf : BnfOptional(s) -> opt(s)
bnf2sdf : BnfAlternative(a, b) -> alt(a, b)
bnfseq1 : BnfSeq(a, seq(b, c)) -> seq(a, [b|c])
bnfseq2 : BnfSeq(a, b) -> seq(a, [b])
```

Listing 5.13: Excerpt from the BNF desugaring code reflects the direct translation of most context-free language constructs from BNF (on the left) to SDF (on the right)

The generation of SDF productions, however, is not enough for practical use in SugarJ. An abstracty syntax tree returned by the SugarJ parser, contains only nodes declared with the `cons("name")` meta-data. We decided to generate a node for every production and name it the same as the production itself.

We further decided to let the grammar generate a concrete syntax tree, retaining all terminals[2] because of its greater flexibility. It is easy to go from concrete to abstract by filtering out nodes, whereas the opposite direction, which is frequently needed for (pretty) printing, is a little more difficult.

The desugared form of the `in_expression` production from Listing 5.12 with terminal symbols inlined is shown in Listing 5.14 below. As you can see the bracketing is different, this is due to different precedence rules in SDF and BNF. Fortunately, because we only deal with abstract syntax trees in desugaring, all of this complexity is handled by the BNF parser and the SDF pretty printer.

[2]but no whitespace

```
PathExpression "NOT"? "IN" "(" ( InItem ( "," InItem )* )
 | Subquery ")" -> InExpression {cons("InExpression")}
```

Listing 5.14: Desugared production from Listing 5.12

5.4 Java Persistence Query Language

5.4.1 Syntax

Thanks to the BNF meta-DSL developed in the previous section, we were able to mostly just copy and paste the JPQL grammar provided by Oracle for documentation.

Unfortunately, this grammar as it is, is ambiguous, and removing these ambiguities required some manual intervention. Presumably, for documentation purposes, the part that deals with arithmetic, functions on strings and dates, and boolean values and expressions tries to encode a type system in the grammar.

Because it is provided for documentation only, the context free grammar omits dealing with lexical issues like number formats and string syntax, but the prose following the grammar describes them in detail. We took the liberty to ignore some of those descriptions. Strings, for example, are delimited not by double quotes (`"java string"`) like in Java, but by single quotes (`'jpql string'`). The reason is likely to avoid escaping every double quote, as the usual means of embedding JPQL code in Java is as a string. Because the JPQL sugar library removes this problem altogether, we chose to reuse Java string syntax for JPQL string syntax by defining `string_literal ::= JavaStringLiteral`. We reuse the Java syntax description in a similar way for several other lexical constructs like numbers and what would be table and column names in a relational database, but are actually Java class names and identifiers in JPQL.

5.4.2 Desugaring

Because JPQL already has an embedding in the host language, as strings and `setParameter` calls on `Query` objects, the desugaring is trivial.

The desugaring is reprinted in Listing 5.15 in its entirety. It is this compact because pretty printing in this case is flattening of a concrete syntax tree (`qls2str`) and because it makes heavy use of higher-order strategies.

The query, represented as an abstract syntax tree after parsing, is flattened and pretty printed to a string. A factory method in the `EntityManager` creates a `Query` object from a query string and this `Query` is then augmented by as many `setParameter` calls as necessary.

```
strategies
  arglist = ?p1#(p2); !p2
  /** Takes a concrete syntax tree
      and returns a pretty printed string */
  qls2str = bottomup(is-string <+ arglist); flatten-list;
      separate-by(|" "); concat-strings
  named-params = collect(\NamedInputParameter(name) -> name\)
  strip-colon = trim-chars(':')
  set-parameter = ?(param, subtree);
      !Invoke(Method(subtree, None(), Id("setParameter")),
        [Lit(String([Chars(<strip-colon> param)])),
         ExprName(Id(<strip-colon> param))])
rules
  compileQuery : QueryCreation(ExprName(em-id), qls)
    -> <foldr(!Invoke(Method(MethodName(AmbName(em-id),
                Id("createQuery"))),
                [Lit(String([Chars(<qls2str> qls)]))]),
          set-parameter)> <named-params> qls
```

Listing 5.15: The complete JPQL desugaring

5.4.3 Editor support

Anecdotal evidence suggest that basic syntax highlighting is helpful to quickly scan code. Fortunately, with SugarJ, it is also very easily implemented and still robust. In Listing 5.16 below, you see all the code needed to make the editor highlight "SELECT" and "FROM" in JPQL queries. For many sugar libraries there is even less code needed as SugarJ tries to recognise keywords automatically, as the accessors sugar library demonstrates.

```
colorer
  _.SelectClause : 127 0 85 bold
  _.FromClause : 127 0 85 bold
```

Listing 5.16: `SELECT` and `FROM` will be displayed in bold, purple font

SugarJ highlighting operates on the parsed representation, `SelectClause` and `FromClause` are the names of the corresponding nodes in the abstract syntax tree. This is more robust than, for example, regular expressions that are used in many editors for colouring purposes. The Sugarclipse editing environment would not highlight a field named "from", even if it was allowed by the grammar, which forbids keywords as identifiers. Mistakes of this kind are identified and brought to the programmer's attention by the JPQL Sugar library.

Another piece of support the JPQL Editor library provides is code folding, implemented as follows:

```
folding
  SelectClause
  FromClause
  WhereClause
```

Automatic code completion is provided through this kind of declaration:

```
completions
  completion template :
    "SELECT " <id>
  completion template :
    "SELECT " <id> " FROM " <id>
```

The editor services for JPQL are only 32 lines, but already offer the same features as the SQL modes of Emacs, vim and many other editors. It is also more robust through decoration of abstract syntax trees.

5.4.4 Static checks

We implemented one static check as a proof of concept.

The JPQL Sugar library alerts the programmer of the use of undeclared identification variables like in the following query:

```
SELECT item.name FROM Item item WHERE ietm.id = 42
```

Listing 5.17: `ietm` is undefined. Figure 4.9 shows Eclipse highlighting a similar error

SugarJ provides a generic way to report errors in Eclipse. It runs the `constraint-error` strategy provided by the sugar library on the parse tree before desugaring and marks the erroneous AST nodes returned by it.

The general idea of finding undeclared identifiers, is to traverse the syntax tree, collect all bindings, and return failure when we find the use of an identifier that is not declared.

The implementation of this idea for the JPQL Sugar library is a little more convoluted. There are three reasons this check is 30 lines of code and not only one or two: (1) the grammar allows several distinct ways to declare and reference identification variables, not just one each, (2) the scope of a binding is not restricted to its children in the abstract syntax tree, and (3) we want to return all unbound identifiers at once, so it is easier to find and fix mistakes in the declaration.

Reason (1) poses no conceptional problem; we just need to introduce more patterns to match declaration and use sites. Reason (2) prevents us from using a general `top-down` and `collect-all` strategy, instead we

need to implement scoping explicitly and implement the traversal more explicitly. For example, we match on the `SELECT` clause node and first collect bindings in the `FROM` clause and only then proceed to check the actual `SELECT` statement and its subtree. This makes (3) more of an effort, because we need to pass the list of errors-so-far around explicitly.

```
rules
  idvdecs = idvdec1 <+ idvdec2
  idvdec1 : FromClause(_, idvd, idvdlist) -> <conc>(<idvdecs>
       idvd, <concat> <map(idvdecs)> idvdlist)
  idvdec1 : IdentificationVariableDeclaration(rvd, joinlist)
      -> <conc>(<idvdecs> rvd, <concat> <map(idvdecs)>
      joinlist)
  idvdec1 : RangeVariableDeclaration(_, _,
      IdentificationVariable(name)) -> [name]
  idvdec2 = arglist; map(idvdecs); concat
strategies
  idv-part = string-tokenize(|['.']); first
  check-sql(|names) = ss(|names)
  idv(|names) = ?iv@IdentificationVariable(name);
    if <elem> (name, names)
      then ![]
      else ![(iv, <concat-strings>["Identification variable ",
           name, " is not in scope."])]
    end

    pe(|names) = ?pe@PathExpression(path);
      if <elem>(<idv-part> path, names)
        then ![]
        else ![(pe, <concat-strings>["Identification varible ",
             <idv-part> path, " is not in scope."])]
      end

      ss(|names) = ?SelectStatement(sc, fc, wc, gc, hc, oc);
        !fc; idvdecs; ?addnames; <conc>(names, addnames); ?
            newnames;
        ![sc, wc, gc, hc, oc]; map(collect-all(idv(|newnames)
            <+ pe(|newnames))); flatten-list
```

Listing 5.18: Excerpt from the static check for undeclared identification variables

6. Discussion and future work

In this section, we discuss the suitability of the Java Pet Store as a model for a real world legacy application, the proposed process for retrofitting language-oriented design and its limitations, as well as the choice of SugarJ.

6.1 The Java Pet Store as a model legacy application

In our humble opinion, much of the recent work in software engineering is very interesting but not necessarily immediately practical. Work on test-driven, model-driven, and agile development as well as the general area of language-oriented design focuses mostly on the development of new applications but neglects the fact that it is not practical for most of the industry to rewrite their software from scratch with every advancement in software engineering processes.

We would like for the Java Pet Store to become the *Drosophila melanogaster*[1] of software engineering. We think, that a common reference application would help researchers trying to find a way to apply their new ideas on legacy code bases, too. The use of models in other disciplines and other areas in computer science, suggests that it encourages collaboration and improves comparability of results. The focus on one model also helps to avoid work that repeats known concepts in only a slightly different stetting. The state of the art in functional programming languages, e.g., would not be where it is today if not for Haskell, a language specifically designed to be the common research model for a strongly and statically typed, pure, lazy, functional programming language.

Several of its properties make the Java Pet Store a good model for a legacy application. It has a common client-server architecture with user interaction through a web interface and a database backend, which is abstracted by a object-relational persistence layer common in the industry. The code itself is object-oriented Java with some use of XML and a SQL-like query language, and most of the user interface is written in Java Server Pages. Like most applications in the wild, its documentation, besides JavaDoc strings, is

[1] Common fruit fly – a model organism in genetics, physiology, and life history evolution

rather poor. Last but not least, it is very stable, the latest Subversion commit is from June 2007, and it is distributed under a liberal license.

Unfortunately, the test coverage of the Java Pet Store code is rather incomplete, but we expect that a few theses on tests engineering, focusing on the Java Pet Store as a model application, would make it one of the most comprehensively tested applications. We suspect that Java might not be the best target for language-oriented research, but the work on the Java Syntactic Extender [1], the Meta Programming System [16], and SugarJ [8] suggest otherwise.

6.2 Retrofitting language-oriented design and the role of SugarJ

The interaction of code that uses sugar libraries with tools written for classic Java code is of special importance for the applicability and acceptance of our proposed retrofitting process by the industry.

SugarJ is text-based, as opposed to projectional workbenches like MPS [6, 16], so interaction with text-oriented software that does not need to analyse code, e.g., version control, poses no difficulties.

For Eclipse, there is a plugin called Sugarclipse that provides a complete editing environment for SugarJ code. Moreover, the editing environment is extensible through editor libraries [7] and is therefore capable of growing together with the language. Integration with other development environments is not inhibited by design, but currently Eclipse is the only choice.

The semantics of languages embedded through sugar libraries are defined through translations to Java, so the last resort to tooling is first running the SugarJ compiler to generate Java code and then using any other tools on the generated Java code. There is a command line compiler that mimics `javac` and can be used from `make` or `ant` to compile SugarJ code, but integration with build tools has not yet been tested exhaustively.

For now, debugging is restricted to generated Java source code, and subsequently generated JVM byte code. Unfortunately, debugging on the level of SugarJ code is currently impossible because all source level information is lost during desugaring. Retaining source level information across program transformations is an interesting field providing many future research opportunities.

One criticism of M. P. Ward's language-oriented programming approach [17] is that creating programming languages is difficult. This criticism naturally transfers to our proposed process of retrofitting language-oriented design. The design and implementation of a sugar library indeed requires a wide area of skills; some theoretical background on formal languages and parsing techniques, experience with Java as the host and target language,

experience with meta-programming, compilers or program transformation, and last but not least, familiarity with the domain in question.

Paul Hudak [10] addressed this difficulty and proposed embedded domain-specific languages that reuse most of the host language's syntax and semantics. This eliminates much of the design and implementation work, but also sacrifices the benefits of domain-specific concrete syntax and flexible semantics. By basing our approach on composable, reusable, independent libraries, we reduce the complexity of embedded language design. Small, independent language extensions are easier to design and can be used jointly to simplify a complex code base. Whether the incremental addition of rather small domain-specific libraries can improve the overall architecture of a large application is something we would like to explore more comprehensively in the future.

Writing sugar libraries can also be made more accessible by exploiting SugarJ's self-applicability. It is already possible to use concrete Java syntax for code that is to be generated instead of writing down the corresponding abstract syntax tree nodes manually. The BNF sugar library written in the course of this work frees sugar library developers from learning SDF.

The use of existing sugar libraries requires no expertise in language design, but an understanding of the desugaring is still helpful to recognise code that might be improved by use of a sugar library. In the future, we hope to work on automatic recognition of code improvable by existing sugar libraries. Research in this area might even result in an automatic conversion tool that would eliminate the fourth step, adaption of existing code, from our process of retrofitting language-oriented design.

Perhaps the most interesting direction of future work is to study fundamentally the interactions of static analyses in general and type systems [2] in particular. Even a complete Java type system only, will enable the JPQL sugar library to guarantee type safety of JPQL queries and semantic code completion in the editing environment. We do not claim to foresee the applications of an extensible type system.

7. Conclusion

We proposed a process for retrofitting Java-based object-oriented legacy applications with language-oriented design. We are able to reduce initial investment, a hurdle to industry adoption, by relying on reusable, composable, and independent sugar libraries and keeping the overall architecture of an application intact. The incremental applicability of our retrofitting process ensures that the software is production ready at all times because partial local rewriting can be deferred as necessary.

To demonstrate the positive effects of language-oriented design on legacy applications in general and the application of our retrofitting process in particular, we made a case study of applying our process to the Java Pet Store. The resulting improvements in code quality are encouraging and the by-products, several sugar libraries, are immediately reusable for other code improvement efforts.

With this work we aim to narrow the gap between advanced processes proposed by the research community and the reality in the software industry.

Bibliography

[1] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–42. ACM, 2001.

[2] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. Available at http://bracha.org/pluggableTypesPosition.pdf.

[3] Frederick P. Brooks, Jr. *The mythical man-month*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Oracle Corporation. BNF Grammar of the Java Persistence Query Language. Available at http://download.oracle.com/javaee/5/tutorial/doc/bnbuf.html.

[5] Oracle Corporation. The Java 5 EE Tutorial. Available at http://download.oracle.com/javaee/5/tutorial/doc/bnbtg.html.

[6] Sergey Dmitriev. Language oriented programming: The next programming paradigm. Available at http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf, 2004.

[7] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[8] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[9] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[10] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[11] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 445–464. ACM, 2009.

[12] The Java Community Process. JSR 220: Enterprise JavaBeans™ 3.0. Available at http://jcp.org/en/jsr/detail?id=220, 2006.

[13] Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[14] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[15] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 2051 of *LNCS*, pages 357–362. Springer, 2001.

[16] Markus Voelter. Embedded software development with projectional language workbenches. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6395 of *LNCS*, pages 32–46. Springer, 2010.

[17] M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.

Appendix. Find all code and future revisions of this thesis here:
`http://www.mathematik.uni-marburg.de/~fehrenbach/bsc`