Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Beyond Annotations: A Proposal for Extensible Java (XJ)

Tony Clark, Paul Sammut, James Willans - Ceteva Ltd.

Abstract

Annotations provide a limited way of extending Java in order to tailor the language for specific tasks. This paper describes a proposal for a Java extension which generalises Annotations to allow Java to be a platform for developing Domain Specific Languages.

1 Introduction

In his 1994 paper on Language Oriented Programming (LOP) [1], Martin Ward proposes that the problems of complexity, conformity, change and invisibility that occur with large software systems can be addressed by designing a *formally specified, domain-oriented, very high level programming language* as the basis for system design. Recently, Guy Steele [2] has emphasised this point:

> *a good programmer in these times does not just write programs. [...] a good programmer does language design, though not from scratch, but building on the frame of a base language*

The term Domain Specific Language (DSL) has been used to refer to languages that have been designed for a restricted class of applications. Martin Fowler [3] makes the distinction between *external* DSLs and *internal* DSLs. An external DSL is written outside the main language of an application whereas an internal DSL is written in and uses the main application language.

Medium to large scale enterprise information systems are implemented using a collection of different technologies. These include JavaScript, JUnit, Ant, (lots of) XML, Drools and various extensions to Java implemented using annotations. In addition to fairly standard technologies such as those mentioned above, there are a large number of extensions being proposed to standard programming languages such as Java. Many of these extensions are implemented using different pre-processor technologies. Each of these different technologies addresses a different aspect of the overall system. Therefore the system is implemented using a collection of loosely integrated DSLs.

The arguments made by Ward in favour of LOP and DSLs are even more relevant today given the proliferation of technologies and the scale of software systems. Developers are using these techniques all the time in various different ways.

The danger with home-grown LOP technology is that it can make the problem worse, not better. If developers use a variety of methods and tools to construct a language then they may get some immediate benefit, but issues will arise if the underlying technology is not stable. In addition, to be useful DSLs must be portable - perhaps not as widely distributed as general purpose languages, but transferrable nonetheless. For large scale benefit, it should not be necessary to supply the complete language support system along with a system module.

This paper is a proposal for a language extension to Java called XJ that supports LOP and therefore allows DSLs to be constructed to be standard, portable and easily understood. The extension is conservative in the sense that it will not conflict with any existing language features and will preserve backward compatibility. The proposal extends classes with syntax definitions to produce new modular language constructs, called syntax-classes, that can easily be distributed along with an application in the usual way.

The LOP extensions to Java have been implemented in an existing language called XMF [4]. XMF is a high-level object-oriented language designed to support DSLs. XMF has been used in a commercial product and has a Java-like sub-language. Therefore we are confident that the XJ proposal has been adequately validated.

This paper is structured as follows: section 2 reviews the state of the art in DSLs; section 3 describes the XJ language extensions; section 4 describes some LOP defined examples using XJ; finally, section 5 analyses the proposal and compares it to related proposals for Java-based LOP technologies.

2 Domain Specific Languages

Consider an idealised development process. You are given an application to develop. You start by trying to understand the kind of executions you will need to perform - just fragments at first. Gradually, you join the executions

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.34

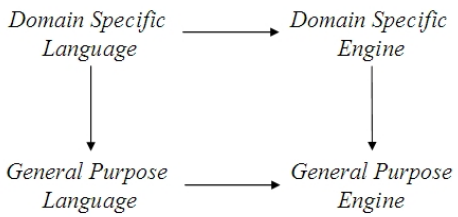

Figure 1. Abstract Model of Development

together to form ever greater descriptions of what will need to happen. The executions, are described in terms of entities from the application domain: financial transactions, customers, interest rates and so on.

You start to notice common themes in the executions and design execution rules that cover them. Gradually, you wind up with a fairly comprehensive collection of execution rules; each rule expects to be given some information that causes it to perform some calculation. The rules define an idealised *domain specific execution engine* and the data that drives the engine defines a *domain specific program*. The range of variability in the programs you can given the engine is a *domain specific language*.

Once you have arrived at an engine and language then you choose a *general purpose language*. You understand the language very well and your implementation task becomes one of working out how to translate the domain specific program into a real-world program so that it faithfully executes on your *general purpose engine*. See figure 1.

Of course, the above process is idealised, however it is representative of how a developer gets from an application specification to a working implementation. How closely the process is followed will depend on how expert the programmer is (there are lots of translations going on). An expert programmer will perform many of the steps instinctively and perhaps often not even be aware of the various representations.

The difference between the DSL and the implementation language is the *representation chasm*. Once the process has jumped from one side of the chasm to the other, it is very difficult to get back. This one-way process is a source of many problems since it loses the meaning of the original representation. Once lost, it is difficult for humans to interpret the code (maintenance becomes an issue), and it is difficult for machines to extract meaning from the code (for example building reuse libraries or generating efficient code).

Current mainstream languages do not provide any support for LOP, however there is increasing interest in LOP and DSLs because of the perceived benefits outlined above. A number of language extensions to Java have been proposed to support LOP and these are analysed in section 5.

The structure of this section is as follows: section 2.1 describes a number of architectural styles for developing DSLs; section 2.2 describes a number of technologies used to implement DSLs; section 2.3 analyses the styles and technologies and makes a proposal for a standard extension to Java that supports LOP and DSLs; finally, section 2.4 lists a number of example DSLs that could be added to Java using the proposals.

2.1 DSL Architectural Styles

There are a number of approaches to architecting a DSL using LOP technologies. This section briefly describes the main styles.

Fowler makes the distinction between *internal* and *external* DSLs. An internal DSL makes use of an existing language and possibly integrates with it whereas an external DSL is completely separate. It should be noted that, if technology supports a fully integrated internal approach then that can be used to implement an external approach.

DSLs may be *fully* or *partially integrated* with their host language. A full integration allows the DSL constructs to be interleaved, where appropriate, with the host constructs. Effectively the host language is extended by the DSL. Partial integration involves levels of restriction on the potential for interleaving. A very restricted integration allows the DSL constructs to occur at pre-defined points in the host language. A fully restricted integration is effectively an external DSL.

DSLs may introduce *new syntax constructs*, may reinterpret the syntax constructs of the host language, or may simply use the existing constructs of a language in a systematic way. New syntax constructs are perhaps the purest form of DSL but will require some parsing technology. Reinterpretation often involves observing patterns of existing language constructs and treating them specially in some way, perhaps involving pattern matching technology. A typical example is to treat certain procedure calls, method calls or field references in a special way.

DSLs may be *translational* or may involve their own *execution engines*. A translational DSL transforms source code from the DSL constructs to existing constructs in a target language. This may involve a pre-processor or be integrated within the target language compiler. A translational approach requires technology for manipulating syntax - concrete, abstract or both.

DSLs that have their own execution engines tend to be external DSLs (but need not be). In this case the DSL program is a data structure that is processed by an engine written specifically for the purpose. Such an approach is related to software frameworks as discussed below.

2.2 Existing Technologies for DSLs

2.2.1 XML

XML has been proposed as a technology for expressing DSLs. This is possible and achieves a standard representation, however XML is not integrated with a programming language in any way and cannot take advantage of the feature of the language in the DSL constructs, therefore it does not meet the criteria set out in [2] for language definition. In certain limited highly declarative domains (for example expressing data structure relationships, declaring project dependencies or menu items) XML is fine, but otherwise it requires a complete language engine to be written from scratch using a representation that is not particularly human-friendly.

2.2.2 Pre-processors

A pre-processor can be used to add DSL features to an existing language. Some pre-processors also allow macros to be defined so that the user can define new language features. In both cases this achieves the aims of LOP but introduces new technology. The pre-processor technology must be distributed with the language definition in order for anyone to use it. Most pre-processing technologies have restricted knowledge of the target language and cannot provide new language features with information about their static context (variables in scope, types etc). This makes it difficult to use pre-processors to extend an existing language in order to achieve [2].

Pre-processors may also be limited to the existing syntax of the target language, i.e. they can interpret patterns of existing constructs, but not introduce new constructs. This feature limits the ability of pre-processors to achieve the goals set out in [1]. Finally, pre-processors may not work with existing language tools. For example, Java IDEs can only process standard Java; language constructs introduced via a pre-processor will not be legal in such an IDE and the code produced by the pre-processor will not be understood by the developer.

2.2.3 Chained Calls

A good place to start when defining a DSL is to make a list of the nouns and verbs associated with the domain. These are good candidates for the data and language features that the DSL will need to support. A style of OOP uses chains of method calls to implement these features. Proponents of this style, claim that the approach achieves the benefits of a DSL without needing any new language features. Whilst chained-calls can lead to readable code and no doubt the approach can be good practice since it focusses design attention on the domain, there is limited scope for proper encapsulation of new concepts and no scope for conveying semantic intent to the language tools.

2.2.4 Processing Concrete Syntax

Systems that support an LOP approach often provide mechanisms for working with concrete syntax. These include patterns for matching structure and templates for generating code. It is important that processing is not limited to concrete syntax since there are often constructs in a language that have no corresponding concrete syntax language feature. For example, in Java there is no concrete syntax for defining a package containing classes. However, in order to achieve [1] it is necessary to be able to produce new features that abstract commonly occurring language patterns.

Systems that are limited to just pattern matching over concrete syntax cannot introduce new language features whose structure differs from that of existing features. In order to deal with new concrete syntax, some form of parsing mechanism is required. One way of achieveing this is to allow grammars to be defined within the language. Each new grammar corresponds to a new language construct. In order for the new language constructs to be embedded within the existing language, the grammar rules should be able to extend those of existing constructs.

2.2.5 Scripting and External DSLs

An external DSL is one that does not extend or use the main implementation language. Such a language is often used to implement a module in a larger system. There are a number of scripting languages, such as JRuby, that can be used in conjunction with general purpose programming languages, such as Java. In these cases the GPL is used to implement the core system and the scripting languages provides a DSL for specific aspects, in the case of JRuby this could be the user interface.

For specific types of application this can work well, but it does not really address the key motivations behind [2, 1]. The scripting language is often just another GPL that happens to be used for a specific aspect of the system. The main GPL (Java) knows little about the semantics of the constructs it is calling in the scripting language. In many cases the scripting language is quite broad in scope and therefore is not really achieving the aim of providing a DSL.

2.2.6 Java Annotations

Java has recently been extended with annotations which can be used to introduce properties to specific places in the source code. The motivation for the annotations has been to provide a mechanism for extending the Java language with new constructs, by allowing existing constructs to have

properties. A new class of construct can be defined by distinguishing between existing constructs based on their property annotations. In addition, the annotations are a standard part of the language and therefore can be processd by Java language tools such as compilers and IDEs.

This approach has been used in a number of extensions to Java and, if used lightly, can be successful. However the aim of achieving new program constructs that reflect the domain or capture programming idioms is not achieved using annotations since they cannot introduce any new syntax. In addition, if annotations are overused then the source code becomes difficult to read.

2.2.7 Frameworks and Libraries

A DSL is related to frameworks in a number of ways. A DSL that extends the syntax of a language typically uses a parsing framework to deal with the new concrete syntax. If the new syntax is fully integrated with an existing language then extension points can be provided by the parser for the existing language to support the extensions. If the compiler or interpreter for an existing language is a framework then it can provide extension points for new language constructs to be expanded into existing language constructs. Finally, if a DSL is to be implemented as an external language then the execution engine for that language can be viewed as a framework where the extension points are implemented in data that is supplied as the DSL program.

2.3 Analysis

We have described DSLs, their architectural styles and some technologies that can be used to implement them. This section reviews the benefits and drawbacks of DSLs and then makes some recommendations regarding an approach to DSL architecture and implementation technology. XJ supports all of these recommendations.

2.3.1 Benefits of DSLs

LOP is important because it allows languages to grow in line with [2]. It allows new programming idioms and patterns to be captured in a standard way, can significantly reduce the number of technologies required to produce a large application, and it supports DSL development. The amount of investment necessary to implement a new language from scratch or even to make significant modifications to an existing language is huge. Given that LOP and DSLs are generally accepted to have benefits, this raises a significant problem regarding DSL development.

DSLs improve the readability and maintainability of code since they close the representation chasm. In addition DSLs improve the scope for a developer to convey the intent of program code to system tools such as IDEs and compilers. This is important for tool support of DSLs and to ensure efficient execution of DSL programs. DSLs can be designed so that system component properties are made explicit which provides scope for reuse.

2.3.2 Drawbacks of DSLs

Current approaches to DSLs suffer from having no standard technology that supports language definition. DSL developers are forced to use a variety of technologies or write their own. This has lead to the objection that DSLs are hish-risk since they require specialised skills for development and are difficult to maintain. Lack of standards for DSLs also mean that tool support for development is weak.

2.3.3 Recommendations

Standardisation is key to achieving the aims of LOP and DSLs. Without standardisation, each LOP developer uses different approaches and technologies which makes the development highly specialized and the maintenance difficult to control. If LOP technology can be standardized then languages become easier to develop since tools can support the standard and problems of maintenance are greatly reduced. Standardization can also significantly reduce the number of technologies necessary to support LOP and develop DSLs.

Should such a standard support fully integrated DSLs or external DSLs? Given that technology for fully integrated DSLs subsumes that of external, the the standard should support a the definition of language constructs that can be fully integrated with a host language. In addition, the standard should allow full concrete syntax extension and the ability to process abstract syntax in arbitrary ways since this is perhaps the essence of DSLs.

Should the standard support a translational approach or an execution engine approach? As noted above, the execution engine approach is universal, but quite specialized. It is more suited to external DSLs than internal DSLs. Whilst there are benefits from an execution engine approach (notably control and debugging), a translational approach is more accessible and supports a more lightweight approach for non-language specialists.

Our proposal is to incorporate the above recommendations as a standard language extension to Java. The extension produces a language called eXtensible Java (XJ) which is introduced in the rest of this paper.

2.4 Example DSLs

The following is a list of some of the DSLs that could be supported by XJ: *SQL* traditional methods of embedding SQL in Java have used strings, a DSL for SQL would embed the language within Java and allow Java expressions to

be references within SQL, see [5]; *Testing* there are a number of proposals for languages to support testing including JUnit and jMock [6] allowing the details of testing to be hidden and for tests to be expressed declaratively; *XML* and *HTML* to produce languages similar to JSP; *Architecture* Marcus Voelter [7] shows how system architecture can be expressed using a DSL and describes how code can be generated from the DSL; *Closures* there is current discussion regarding how to add closures to Java; *SPLA* feature models and product line architectures allow single systems to describe a number of products; *Hardware* many hardware systems are controlled by small scripting languages that could be embedded in embedded within Java; *Rules* rules systems such as Drools [8] already class themselves as DSLs; *GUI* Libraries for constructing GUIs tend to require a large amount of code to create, link and configure the various graphical components. DSLs would allow the detail of this code to be hidden and constructed in a standard way.

3 XJ

XJ is a proposed extension to Java that supports LOP. It introduces the idea of a *syntax-class* into Java. A syntax-class is a normal Java class that defines a language grammar. When the Java parser encounters an occurrence of a language feature delimited by a syntax-class, the class's grammar is used to process the input. If the parse succeeds then the grammar synthesizes an Java abstract syntax tree (AST). An object of type AST has a standard interface that is used by the Java compiler when it processes the syntax. New types of AST can be defined providing that they implement the appropriate interface.

The rest of this section introduces XJ using examples and analyses the main features: section 3.1 specifies a simple language construct; section 3.2 shows how Java is extended with concrete syntax for the new construct; section 3.3 describes the features in XJ for specifying concrete syntax; section 3.4 describes how the new construct is translated into standard Java; section 3.5 describes how abstract syntax is manipulated in XJ; section 3.6 describes some further simple language examples in XJ; finally, 3.7 reviews the main features of XJ.

3.1 The Select Language Construct

Consider a simple language construct in Java that selects an element from a collection based on some predicate. An example of the new construct is shown below:

```
import language mylang.Select;

public Person getChild(Vector<Person> people) {
  @Select Person p from people where p.age < 18 {
    return p;
  } else { return null; }
}
```

The new language construct is called Select. A LOP-defined construct in XJ is used by prefixing a reference to the syntax-class with the @-character. The value `p` is selected from the vector `people` providing that the age of the person is less than 18. If a value can be selected then it is returned otherwise null is returned. The use of Select is equivalent to the following definition:

```
public Person getChild(Vector<Person> people) {
  for(int i = 0; i < people.size(); i++) {
    Person p = people.elementAt(i);
    if(p.age < 18) return p;
  }
  return null;
}
```

3.2 Concrete Syntax for Select

A new language construct is defined in XJ by defining a syntax-class. A syntax-class contains a grammar which is used by the Java parser to process the concrete program syntax and to return an abstract syntax tree (AST). Once a syntax-class has been defined, it can be used in program code by referencing the class after the syntax escape character '@'.

Figure 2 shows the concrete syntax part of the syntax-class for Select. The grammar definition describes how to recognize a select statement. A grammar consists of named parse rules. The grammar for Select extends that for Statement which allows Select to be used wherever a statement is expected and also allows Select to reference the parse rules defined for statements (in this case Type, Exp and Block).

The Select rule specifies that a well-formed statement is a type followed by a name, the keyword `from` followed by an expression, the keyword `when` followed by an expression and then a block which is the body of the select. After the body there may be an optional `else` keyword preceding a block.

In each case within the Select rule, the parse elements produce a value that may optionally be associated with names. For example, the type is associated with the name T. In addition, a parse rule can contain Java statements that return a value. These are enclosed in { and }, and may reference any of the names that have been defined to the left of the statement. The final value returned by the Select rule is an instance of the class Select.

3.3 XJ Grammars and Parsing

XJ requires Java classes to be extended with an optional grammar. Classes that contain a grammar definition

```
package mylang;

import language java.syntax.Grammar;

import java.syntax.*;

public class Select extends Sugar {
  private Type type;
  private Var var;
  private AST collection, test;
  private Block body, otherwise;
  public Select(Type T,String n,AST c,AST t,Block b,Block o) {
    // Initialize fields from arguments...
  }
  @Grammar extends Statement {
    Select ::=
      T = Type
      n = Name
      'from' c = Exp
      'when' t = Exp
      b = Block
      o = ('else' Block | { return new Block(); })
      { return new Select(T,n,c,t,b,o); }.
  }
```

Figure 2. A Select Command (Part 1 of 2)

are referred to as syntax-classes. XJ also requires that the Java parser is extended to allow the current grammar to be changed during a parse.

Referencing a syntax-class after the syntax escape character '@' causes the Java parser to temporarily switch grammars. When the parser encounters `@C ...`, it finds the class `C` (loading it if necessary), extracts its grammar and continues with the parse using the grammar defined by the class. The starting non-terminal for the class is always the parse rule named `C`.

The XJ parser maintains a stack of grammars. When an `@C ...` construct is encountered, the grammar for `C` is pushed onto the stack. If the current grammar succeeds, then the stack is popped, the current value is returned and the parse continues with the grammar at the top of the stack. XJ grammars can be associated with their own tokenizers, but get the standard Java tokenizer by default.

XJ grammars have been implemented and used for a number of years in the XMF system [15]. XMF is open-source so the parsing algorithms and language for expressing grammars are available for inspection.

3.4 Abstract Syntax for Select

The value synthesized and returned by a grammar must be an instance of java.syntax.AST. If the return value is an instance of one of the standard Java AST classes then no special action needs to be taken by the syntax-class. If the return value is an instance of a user-defined syntax-class then that class must implement the AST interface which is used by the compiler to translate the source code into Java VM code. To make this process easier, a user defined syntax-class can extend java.syntax.Sugar which implements the AST interface through a method called desugar. The desugar method is responsible for translating the receiver into an AST for which the interface is defined (typically desugaring into standard Java code).

The syntax-class Select extends the class java.syntax.Sugar and defines the desugar operation in figure 3. The Java compiler needs to process an AST instance in various ways. It achieves this via the AST interface. The class java.syntax.Sugar implements the AST interface by calling desugar and then performing the appropriate AST operation on the result. The desugar operation is supplied with the current compilation context which contains the variables in scope, types etc.

The Select syntax-class uses desugar to produce the selection code. The particular code depends on the type of the collection so the first thing that desugar does it to work out the type of the collection and dispatch to an appropriate desugaring method. Figure 3 shows how vectors are desugared, other types such as arrays are very similar. The desugarVector method returns code using quasi-quotes which are explained in the next section.

3.5 Quasi-Quotes

To define the desugar method, a syntax-class will generally construct new AST instances. This process is made easy in XJ through the use of quasi-quotes. A quasi-quoted AST is shown below:

```
  public AST desugar(Context context) {
    Class<T> cType = context.getType(collection);
    if(isVector(cType)) return desugarVector(cType,contect);
    else // More cases...
  }
  public AST desugarVector(Class<T> cType,Context context) {
    Var done = context.newVar();
    Var coll = context.newVar();
    return [| boolean <done> = false;
              <cType> coll = <collection>;
              for(int i = 0; i < <coll>.size(); i++) {
                <underlyingType(cType)> <var> = <coll>.elementAt(i);
                if(<test>) {
                  <done> = true;
                  <body>;
                }
              }
              if(!<done>) <otherwise>;
            |];
  }
}
```

Figure 3. A Select Command (Part 2 of 2)

```
[| x + <new java.syntax.Int(1)> |]
```

which is equivalent to the Java expression:

```
new java.syntax.BinExp(
  new java.syntax.Var("x"),
  "+",
  new java.syntax.Int(1))
```

The delimiters [| and |] transform the enclosing concrete syntax into the corresponding abstract syntax. Within [| and |] the delimiters < and > can be used to 'unquote' the syntax in order to drop in some abstract syntax. The two forms of delimiters can be arbitrarily nested. Quasi-quotes are an easy way to create code templates in XJ.

The drop-quotes < and > allow a single AST valued expression to be 'dropped' into program code. Sometimes, it is necessary to splice a collection of program elements into a single location. For example, this can occur when a single DSL construct expands into a collection of method definitions. XJ provides splice-quotes <$ and $> to support this. For example the following is a method that constructs a class definition with a fixed field and a variable number of methods:

```
public AST mkClass(String n,Vector<Method> M) {
  return
    [| public class <n> {
         private int storage;
         <$ M $>
       }
    |];
}
```

3.6 Other Examples

Syntax expansions can make use of syntax-classes (similar to macros-calling-macros) in XJ. Some of the examples in the rest of this paper use a couple of syntax-classes in this way. These are specified in this section without their implementations.

3.6.1 Iterate

The UML Object Constraint Language provides a useful language construct called 'iterate' which is a restricted form of collection-folding. The following is a simple example of the use of a syntax-class called Iterate:

```
@Iterate int i in nums with int sum = 0 {
  return sum + i;
}
```

The example adds up a collection of integers in the collection `nums`. This code is equivalent to a call of a method called `addUp`:

```
public int addUp(Vector<Integer> nums) {
  int sum = 0;
  for(int i : sums) sum = sum + i;
  return sum;
}
```

3.6.2 Comprehensions

A set comprehension is a standard construct in mathematics for processing set-elements and constructing new sets. This is a useful feature for processing Java collections:

```
@Cmp(x + 1) { int x <- nums }
```

which creates a new collection by adding 1 to the integers in the collection `nums`. This is equivalent to a call of `add1`:

```
public Vector<Integer> add1(Vector<Integer> nums) {
  Vector<Integer> vec = new Vector<Integer>();
  for(int x : nums) vec.addElement(x + 1);
  return vec;
}
```

3.7 Review

The main features of XJ are: *syntax-classes* XJ extends Java classes with grammars; a new type of syntax construct `@`; *import* a new import mode for importing references to syntax classes; *AST* access to, and standardization of, Java abstract syntax and the compiler context; *quasi-quotes* that allow abstract syntax to be constructed as though it were concrete syntax; *lifting* any Java value can be transformed into an AST object whose evaluation will produce the original value.

4 A DSL in XJ

This example is due to Martin Fowler [16]. Suppose we have data that contains information about customer events. A customer event might be a service call including the name of the customer, the type of the call (failed hardware, billing query etc) and the date. This information may be provided in real-time or in a log file as text:

```
SVCLFOWLER    10101MS0120050313
SVCLHOHPE    10201DX0320050315
SVCLTWO     x10301MRP220050329
USGE10301TWO     x50214..7050329
```

A Java program is to process the information. Obviously the first task of the Java application is to split the input strings up in terms of their fields. If there are a very small number of types of call then it would be OK to just write the appropriate string manipulation calls on the input. However, it would be better to define a string processing framework and use that. Fowler gives a framework-based implementation something like that shown in figure 4.

A DSL could be defined that makes the code much easier to read:

```
@Reader CallReader
  map(SVCL,ServiceCall)
    4-18:CustomerName
    19-23:CustomerID
    24-27:CallTypeCode
    28-35:DataOfCallString
  end
  map(USGE,Usage)
    4-8:CustomerID
    9-22:CustomerName
    30-30:Cycle
    31-36:ReadDate
  end
  do
    ServiceCall
    Usage
end
```

which is equivalent to a class definition:

```
public class Callreader {
  public void ConfigureCallReader(Framework f) { ... }
  private ReaderStrategy ConfigureServiceCall() { ... }
  // More configurations...
}
```

The syntax-class for the Reader construct is defined in figure 5.

5 Analysis and Related Work

This paper has reviewed the motivation for language oriented programming (LOP) and domain specific languages (DSLs). It has described and analysed a number of approaches and technologies for LOP and DSLs and has made some recommendations regarding how LOP and DSLs can become part of mainstream system development. The key recommendation is that the technology for LOP should become a standard part of a mainstream language and we have shown how this can be achieved by defining eXtensible Java (XJ). XJ supports LOP through the introduction of syntax-classes.

A syntax-class is a standard Java class extended with a grammar. Once defined, a syntax-class can be used as a construct within Java programs through the use of the '@' language escape character. LOP is also supported within XJ through standard access to Java abstract syntax and the use of quasi-quotes. Quasi-quotes originate in the semantic brackets or Quine-Quotes used in programming language theory (notably the work of Scott and others on denotational semantics). Recently they have been used for meta-programming systems [18] and [19]. The latter gives a good overview of the field.

There are other systems that support LOP within Java. OpenJava (OJ) [9] supports LOP using a meta object protocol (MOP) that supports syntax expansion. Each class may specify a meta-class that is responsible for expanding its definition. The meta-class uses a standard definition interface to process its instances. This is a very flexible approach which supports complete access to abstract syntax. However, OJ does not include a mechanism for extending the concrete syntax of the language.

Maya [10] uses a pattern matching mechanism to define macro expansion rules on Java abstract syntax trees. The macros are defined using a new language concept called a *mayan* which defines a grammar production rule in terms of patterns. Maya allows macros to be overloaded based on the type of the arguments, thereby allowing the macros to expand differently. Maya offers similar features to XJ, however the changes to the language are global and not modularized by being attached to classes. The use of the syntax escape @-character, means that new syntax constructs are encapsulated, attached to classes and do not have global effect on the Java grammar. This is in contrast to Mayans that are independent of classes and have global scope much like macros in Lisp and Scheme.

The Java Syntactic Expander (JSE) [11] is a macro system that supports the definition of macros in Java and uses a quasi-quote mechanism to process the abstract syntax. JSE

```
public void ConfigureCallReader(Framework framework) {
  framework.registerStrategy(ConfigureServiceCall());
  framework.registerStrategy(ConfigureUsage());
}
private ReaderStrategy ConfigureServiceCall() {
  ReaderStrategy result = new ReaderStrategy("SVCL",typeof(ServiceCall));
  result.addFieldExtractor(4,18,"CustomerName"));
  result.addFieldExtractor(19,23,"CustomerID"));
  result.addFieldExtractor(24,27,"CalltypeCode"));
  result.addFieldExtractor(28,35,"DataOfCallString"));
}
private ReaderStrategy ConfigureUsage() {
  ReaderStrategy result = new ReaderStrategy("USGE",typeof(Usage));
  result.addFieldExtractor(4,8,"CustomerID"));
  result.addFieldExtractor(9,22,"CustomerName"));
  result.addFieldExtractor(30,30,"Cycle"));
  result.addFieldExtractor(31,36,"ReadDate"));
}
```

Figure 4. Configuring a Framework for Service Calls

```
import language java.syntax.Grammar;
public class Reader extends Sugar {
  private String name;
  private Vector<Mapping> mappings;
  private Strategy strategy;
  public Reader(String n,Vector<Mapping> m,Strategy s) {
    // Initialization from args...
  }
  @Grammar {
    Reader ::= n = Name '{' M = Mapping*  s = Strategy '}' { new Reader(n,M,s) }.
    Mapping ::= 'map' '(' t = Name ',' n = Name ')' '{' F = Field* '}' { new Mapping(t,n,F) }.
    Field ::= s = Int '-' e = Int ':' n = Name { new Field(s,e,n) }.
    Strategy ::= 'do' N = Name* { new Strategy(@Cmp new MappingRef(n) | n <- N end) }.
  }
  public AST desugar(Context context) {
    return
      [| public class <name> {
           public void <"Configure" + name>(Framework framework) {
             <@Iterate Strategy s in strategies with e = [| |] {
                String name = "Configure" + s.getName();
                return [| <e>; framework.registerStrategy(<name>()); |]
             }>
             <$ @Cmp([| private ReaderStrategy <"Configure" + m.getName() {
                          ReaderStrategy result = new ReaderStrategy(t,typeof(<v>));
                          <@Iterate Field f in m.getFields() with e = [| |] {
                             AST start = f.getStartPos().lift();
                             AST end = f.getEndPos().lift();
                             AST name = f.getName().lift();
                             return [| <e>; result.addFieldExtractor(<start>,<end>,<name>) |];
                           }>
                        } |]) {
                AST t = m.getTag().lift();
                AST v = Var(m.getName());
                Mapping m <- mappings;
             } $>
         }
       }
      |];
   }
}
```

Figure 5. Reader Syntax Class

does not support new syntax rules and operate on the concrete syntax of the language. As we have seen in XJ it is sometimes necessary for syntax-classes to have raw access to AST structures.

Meta-AspectJ (MAJ) [12] is an annotation-based language extension mechanism that generates AspectJ. MAJ does not support concrete language extension.

The Jakarta Tool Suite (JTS) [13] provides a macro facility and uses a quasi-quote like mechanism for dealing with abstract syntax. The expansion mechanism provides access to environments that are similar to the compiler contexts of XJ. JTS allows the concrete syntax of Java to be extended via the Bali parser. JTS is aimed at writing GenVoca generators and as such, although it provides a number of key LOP features, it is a pre-processor and therefore not as integrated with Java as the proposal for XJ.

The Fortress language [17] is an example of a new language that has included features that support LOP. The support is fairly simple: delimiters can be specified and the compiler supplies the raw string between the delimiters to user code for processing. However, this is an indication that LOP and DSLs are starting to emerge in languages simed at the mainstream.

XJ is a proposal for a language extension to Java. In order for XJ to achieve its potential as a LOP system to support mainstream DSL development it must become part of the Java standard. A requirement for any proposal for mainstream language extension is that it should undergo a rigorous validation process. Although XJ has not been implemented in Java, it is one of the key features of the XMF language [15, 14] that has been used in commercial tools (XMF-Mosaic) and has been made open-source in 2008. All of the language mechanisms and algorithms necesary to implement XJ have been validated through the XMF system.

References

[1] Language Oriented Programming. Martin Ward. Software - Concepts and Tools, Vol.15, No.4, pp 147-161, 1994

[2] Growing a language. G. Steele. OOPSLA '98 Addendum. ACM Press, New York.

[3] Martin Fowler's blog on Domain Specific Languages `http://www.martinfowler.com/articles/languageWorkbench.html`

[4] XMF. Available from `http://www.ceteva.com`.

[5] JEQUEL A Java embedded Query language (SQL in Java). `http://jequel.de/index.php?n=Main.JEQUEL`

[6] Evolving an Embedded Domain Specific Language in Java. S. Freeman, N. Price. OOPSLA 2006.

[7] Architecture as Language: A Story. Marcus Voelter. `http://www.infoq.com/articles/architecture-as-language-a-story`.

[8] Drools. `http://www.jboss.org/drools/`

[9] OpenJava: A Class-based Macro System for Java. M. Tatsubori, et al. LNCS 1826, pp.117-133. Reflection and Software Engineering. Springer-Verlag, 2000.

[10] Maya: Multiple-Dispatch Syntax Extension in Java. Jason Baker, Wilson C. Hsieh. The 2002 Conference on Programming Language Design and Implementation.

[11] The Java Syntactic Extender (JSE). Jonathan Bachrach, Keith Playford. OOPSLA 2001.

[12] Easy Language Extension with Meta-AspectJ. S. Huang, Y. Smaragdakis. ICSE 2006.

[13] JTS: tools for implementing domain-specific languages. Batory, D. Lofaso, B. Smaragdakis, Y. Proceedings. Fifth International Conference on Software Reuse, 1998. Publication Date: 2-5 Jun 1998 On page(s): 143-153.

[14] Applied MetaModelling: A Foundation for Language Driven Development. Second Edition, 2008. T. Clark, P. Sammut, J. Willans. E-book available from `http://www.ceteva.com`.

[15] Superlanguages: Developing Languages and Applications with XMF. First Edition 2008. T. Clark, P. Sammut, J. Willans. E-book available from `http://www.ceteva.com`.

[16] DSLs. Martin Fower presentation at JA00 2006. `http://www.infoq.com/presentations/domain-specific-languages`.

[17] The Fortress Language Specification. Allen et al. Version 1.0 2008. Available from `http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf`

[18] Template Meta-Programming for Haskell. T. Sheard, S. Peyton-Jones. ACM Sigplan Notices, 37(12) 2002.

[19] Domain Specific Language Implementation via Compile-Time Meta-Programming Preprint PDF L. Tratt. To appear, TOPLAS, 2009.