Easy Language Extension with Meta-AspectJ

Shan Shan Huang, Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{ssh,yannis}@cc.gatech.edu

ABSTRACT

Domain-specific languages hold the potential of automating the software development process. Nevertheless, the adoption of a domain-specific language is hindered by the difficulty of transitioning to different language syntax and employing a separate translator in the software build process. We present a methodology that simplifies the development and deployment of small language extensions, in the context of Java. The main language design principle is that of language extension through unobtrusive annotations. The main language implementation idea is to express the language as a generator of customized AspectJ aspects, using our Meta-AspectJ tool. The advantages of the approach are twofold. First, the tool integrates into an existing software application much as a regular API or library, instead of as a language extension. This means that the programmer can remove the language extension at any point and choose to implement the required functionality by hand without needing to rewrite the client code. Second, a mature language implementation is easy to achieve with little effort since AspectJ takes care of the low-level issues of interfacing with the base Java language*.

1. INTRODUCTION AND MOTIVATION

The idea of extensible languages has fascinated programmers for many decades, as evident by the extensibility features in languages as old as Lisp. From early on in the history of software development, programmers dreamed of expressing complex software modules as reusable language extensions instead of as plain libraries. From a Software Engineering standpoint, the main advantages of expressing a concept as a language feature, as opposed to a library API, are in terms of conciseness, safety, and performance. A language feature can allow expressing the programmer's intent much more concisely—in contrast, libraries are limited to a function- or method-call syntax. A language feature can also enable better static error checking—a library can only check the static types of arguments of a function call against the declared types of the formals. Finally, a language feature can take advantage of context information and employ an optimized implementation, while a library routine cannot be customized according to its uses.

Despite these advantages, there are excellent reasons why full language extensibility is undesirable. Changing the syntax and semantics of a programming language is confusing and can lead to incomprehensible code. Furthermore, programming languages are complex entities, designed to provide a small number of features but allow them to be combined as generally as possible. A new feature can either increase the complexity of the language implementation significantly (because of interactions with all existing features), or will need to be limited in its interactions, which is a bad language design principle that leads to single-use features and design bloat.

In past work [3], we have advocated expressing small language extensions purely through unobtrusive annotations. Indeed, the introduction of user-defined annotations in mainstream programming languages, such as C# and Java, has allowed specialized language extensions (e.g., for distributed computing, persistence, or real-time programming) to be added without changing the base syntax.

We believe that the approach of limited language extension through annotations meshes very well with an implementation technique that uses our Meta-AspectJ (MAJ) tool [4] to express the semantics of the language extension. Specifically, MAJ is a language that allows writing programs that *generate* aspects in the AspectJ language [1]. In this way, the programmer can easily express an extension to the Java language as a program that: a) reads annotations and type information from an existing program using Java reflection; b) outputs a customized AspectJ aspect that is responsible for transforming the original program according to the information in the annotation; c) executes the generated aspect by using the standard AspectJ compiler.

In other words, our approach uses the AspectJ language as a compiler back-end. AspectJ code is not written by the application programmer but gets generated by the language extension, for the sole purpose of expressing program transformations easily and generally. This is appropriate, as AspectJ embodies the Aspect-Oriented Programming [2] philosophy of expressing program enhancements orthogonally and independently of the original source code.

Our approach has the advantage of simplifying the imple-

*This material is based upon work supported by the National Science Foundation under Grants No. CCR-0220248 and CCR-0238289.

mentation of the language extension significantly, yet without encouraging undisciplined language extension (since the only extensions allowed are through annotations). Specifically, the approach leverages the engineering sophistication of the AspectJ compiler implementation and its provisions for dealing correctly with different Java language features. If a programmer were to replicate the same effort by hand, she would be likely to need to reproduce much of the AspectJ compiler complexity.

The purpose of this paper is to support the idea of implementing small language extensions as programs that produce aspects. We have recently implemented a number of such small extensions to Java and they all exhibit a striking simplicity. Specifically, we did not have to implement (or extend) a Java parser, we did not need to deal with syntax tree pattern matching and transformation, and we did not need to provide special handling for many Java complexities. The combined annotations-MAJ approach ensured that our small language extensions were implementable in a few hundreds of lines of code, without sacrificing generality in their conditions of use. We discuss two such extensions in detail, after first introducing the MAJ language.

2. BACKGROUND: MAJ SYNTAX

MAJ is an extension of Java that allows writing programs that generate AspectJ source code. MAJ offers two operators for creating AspectJ code fragments: `‘[...]` ("quote") and `#[...]` ("unquote"). The quote operator creates representations of AspectJ code fragments. Parts of these representations can be variable and are designated by the unquote operator (instances of unquote can only occur inside a quoted code fragment). For example, the value of the MAJ expression `‘[call(* *(..))]` is a data structure that represents the abstract syntax tree for the fragment of AspectJ code `call(* *(..))`. Similarly, the MAJ expression `‘[!within(#className)]` is a quoted pattern with an unquoted part. Its value depends on the value of the variable `className`. If, for instance, `className` holds the identifier "`SomeClass`", the value of `‘[!within(#className)]` is the abstract syntax tree for the expression `!within(SomeClass)`.

MAJ also introduces a new keyword `infer` that can be used in place of a type name when a new variable is being declared and initialized to a quoted expression. For example, we can write:

```
infer pct1 = ‘[call(* *(..))];
```

This declares a variable `pct1` that can be used just like any other program variable. For instance, we can unquote it:

```
infer adv1 = ‘[void around() : #pct1 { }];
```

This creates the abstract syntax tree for a piece of AspectJ code defining (empty) advice for a pointcut. Of course, since AspectJ is an extension of Java, any regular Java program fragment can be generated using MAJ.

We can now see a full MAJ method that generates a trivial but complete AspectJ file:

```
void generateTrivialLogging(String classNm) {
  infer aspectCode =
   ‘[   package MyPackage;
        aspect #[classNm + "Aspect"] {
          before : call(* #classNm.*(..))
            { System.out.println("Method called"); }
        }
    ];
  System.out.println(aspectCode.unparse());
}
```

The generated aspect causes a message to be printed before every call of a method in a class. The name of the affected class is a parameter passed to the MAJ routine. This code also shows the `unparse` method that MAJ supports for creating a text representation of their code.

3. EXAMPLE 1: FILLING INTERFACE METHODS

Our first language extension is simple but represents a good exposition example to our approach, since it can be defined very quickly and it is hard to implement with alternate means.

The Java language ensures that a class cannot declare to "implement" an interface unless it provides implementations for all of its methods. Nevertheless, this often results in very tedious code. For instance, it is very common in code dealing with the Swing graphics library to implement an event-listener interface with many methods, yet provide empty implementations for most of them because the application does not care about the corresponding events. The example code below is representative:

```
private class SomeListener
    implements MouseListener, MouseMotionListener
{
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }

  // the rest are not needed. Provide empty bodies.
  public void mouseClicked (MouseEvent event) {}
  public void mouseReleased (MouseEvent event) {}
  public void mouseEntered (MouseEvent event) {}
  public void mouseExited (MouseEvent event) {}
  public void mouseMoved (MouseEvent event) {}
}
```

Of course, the programmer could avoid providing the empty method bodies on a per-interface basis, by associating each interface with a class that by default provides empty implementations of all interface methods. Then a client class can inherit the empty implementations and only provide implementations for the methods it needs. This pattern is indeed supported in Swing code (through library classes called *adapters*), but it is usually not possible to employ since the listener class may already have another superclass. Instead, it would be nice to provide a simple Java language extension implemented as an annotation. The implementation of the extension would be responsible for finding the unimplemented methods and supplying empty implementations by default (or implementations that just return a default primitive or null value, in the case of methods that have a return type). In this case, the above class could be written more simply as:

```
@Implements ({"MouseListener","MouseMotionListener"})
public class SomeListener {
  public void mousePressed (MouseEvent event) {
    ... // do something
  }
  public void mouseDragged (MouseEvent event) {
    ... // do something
  }
}
```

Of course, this extension should be used carefully since it weakens the tests of interface conformance performed by the Java compiler.

We implemented the above Java extension using MAJ. The code for the implementation was less than 200 lines long, with most of the complexity in the traversal of Java classes, interfaces, and their methods using reflection. The code processes a set of given Java classes and retrieves the ones with an `Implements` annotation. It then finds all methods that are in any of the interfaces passed as arguments to the `Implements` annotation and are not implemented by the current class. For each such method, code is generated in an AspectJ aspect to add an appropriate method implementation to the class. For instance, the code to add the method to the class in the case of a `void` return type is:

```
infer newMethod =
    `[ public void #methodName (#formals) {} ];
aspectMembers.add(newMethod);
```

Finally, the class needs to be declared to implement the interfaces specified by the annotation. This is easily done by emitting the appropriate AspectJ code:

```
infer dec = `[declare parents:
  #[c.getName()] implements #[iface.getName()]; ];
```

The final aspect (slightly simplified for formatting reasons) generated for our earlier listener class example is:

```
public aspect SomeListenerImplementsAspect1 {
  void SomeListener.mouseClicked(MouseEvent e) {}
  void SomeListener.mouseEntered(MouseEvent e) {}
  void SomeListener.mouseExited(MouseEvent e) {}
  void SomeListener.mouseMoved(MouseEvent e) {}
  void SomeListener.mouseReleased(MouseEvent e) {}

  declare parents:
     SomeListener implements MouseListener;
  declare parents:
     SomeListener implements MouseMotionListener;
}
```

This aspect performs exactly the modifications required to the original class so that it correctly implements the `MouseListener` and `MouseMotionListener` interfaces.

We invite the reader to consider how else this language extension might be implemented. Our approach of using annotations in combination with MAJ yielded a very simple implementation by letting AspectJ deal with most of the complexities of Java. Specifically, we did not have to deal with the low-level complexities of either Java source syntax or Java bytecode. For instance, we did not have to do any code parsing to find the class body or declaration that needs to be modified. Dealing with Java syntactic sugar, such as inner classes, was automatic. We did not need to do a program transformation to add the `implements` clauses or the extra methods to the class. Similarly, we did not need to worry about the valid syntax for adding an implemented interface if the class already implements another.

4. EXAMPLE 2: LANGUAGE SUPPORT FOR OBJECT POOLING

Our second example language extension addresses a common programming need, especially in server-side programming. Software applications often find the need for pooling frequently-used objects with high instantiation costs. We use the following database connection class as a running example:

```
public class DBConnection {
    public DBConnection(String dbURI,
                        String userName,
                        String password ) { ... }
    public void open() { ... }
    public void close() { ... }
}
```

The cost of an `open()` call is very high for a database connection. In applications concerned with performance, such as high-volume websites with lots of database requests, one often finds the need to pool database connections and keep them open, instead of repeatedly creating new ones and opening them.

Making a class such as `DBConnection` into a "pooled" class involves at the very least creating a pooling manager class that knows how to manage instances of the class being pooled. A different pooling manager class needs to be developed for each class being pooled, since the manager needs to have class-specific information such as how to instantiate a new instance of the class when the pool is running low (e.g., `DBConnection` objects are created by a constructor call, followed by an `open()` call), and how to uniquely identify objects of the same class that belong to different pools (e.g., `DBConnection` objects of different `dbURI`, `userName`, and `password` combinations need to be in different pools, and the pooling manager needs to understand which pool to fetch objects from when a request arrives).

We expressed the pooling concept as a language feature that can used transparently with any Java class, as long as some broad properties hold regarding its construction and instantiation interface. The rest of the application will be completely oblivious to the change. This facilitates the application of pooling after a large code base which uses the class in its non-pooled form has been developed. Using our extension, converting a class to a pooled class involves only 4 annotations: `@pooled`, `@constructor`, `@request`, and `@release`. For example, to convert the `DBConnection` class into a "pooled" class, and to adapt an existing code base to using the pooled functionality, the user only has the add the following annotations to the code:

```
@pooled(mgr=pooled.PoolTypes.BASIC, max=10, min=2)
public class DBConnection {
  @constructor
  public DBConnection(String dbURI,String userName,
                      String password ) { ... }
  @request
  public void open() { ... }
```

```
  @release
  public void close() { ... }
}
```

The `@pooled` annotation indicates that the class `DBConnection` should be pooled. It accepts parameters that can be used to customize the pooling policy. `@constructor` annotates the constructor call whose parameters serve as unique identifiers for different kinds of `DBConnection` objects. In this example, `DBConnection` objects with different `dbURI`, `userName`, and `password` combinations should be maintained separately. `@request` annotates the method that signals for the request of a pooled object, and `@release` annotates the method call that signals for the return of the pooled object back to the pooling manager.

The implementation of this language extension using MAJ is less than 400 lines of code. The MAJ program searches for classes annotated with `@pooled`, and generates two Java classes and one aspect to facilitate converting this class to be pooled. We next describe the generated code in more detail. The reader may want to consider in parallel how the same task could be accomplished through other means. Neither conventional Java facilities (i.e., classes and generics) nor AspectJ alone would be sufficient for expressing the functionality we describe below in a general way, so that it can be applied with little effort to arbitrary unomdified classes. For instance, none of these facilities can be used to create a proxy class with methods with identical signatures as those of an arbitrary Java class.

First, a pooling manager class, `PoolMgrForDBConnection`, is generated for `DBConnection`. The pooling manager class contains methods for requesting and releasing pooled `DBConnection` objects, as well as code to manage the expansion of the pool based on the `min` and `max` parameters.

In order to retrofit an existing code base to use `DBConnection` as a pooled class, we need to introduce proxy objects that will be used wherever an object of the original class would exist in the application code. This is necessary as different objects from the perspective of the client code will correspond to the same pooled object. We generate a proxy class as a subclass of the pooled class. In our example: `DBConnection_Proxy extends DBConnection`. All instances of the proxy class share a static reference to an instance of `PoolMgrForDBConnection`. Each proxy instance holds (non-static) references to the parameters to the `@constructor` constructor call, and the `DBConnection` object obtained from the pooling manager. The proxy class rewrites the `@request` and `@release` methods: the `@request` method is rewritten to obtain an object of `DBConnection` type from the pooling manager, using the unique identifiers kept from the constructor call, and the `@release` method returns the `DBConnection` method back to the pool, while setting the reference to this object to null. The MAJ code in the proxy takes care to exactly replicate the signature of the original methods, including modifiers and `throws` clauses. For instance, the `@release` method in the proxy is generated as:

```
infer meth =
 `[ #mods #ret #[m.getName()] (#formals) #throwStmt
    {
      m_poolMgr.release(m_uniqueId, m_proxiedObj);
      m_proxiedObj = null;
    }];
```

All other methods simply delegate to the same method in the superclass.

The idea is to have variables declared to hold a `DBConnection` object, now hold a `DBConnection_Proxy` object. Therefore, to complete the "proxy" pattern, we need to change all the calls of `new DBConnection(...)` to `new DBConnection_Proxy(...)`. This is the role of our generated aspect: tedious recoding effort is easily replaced by an aspect: the aspect intercepts all the constructor calls of `DBConnection`, and returns an object instantiated by calling `new DBConnection_Proxy(...)`.

In summary, a user can easily turn a class into a pooled class, *and* retrofit any existing code base to use this class in its new, pooled form. The client code does not need to be hand-edited at all, other than with the introduction of our 4 annotations.

5. FUTURE WORK

We believe that the years to come will see the emergence of a healthy ecology of small language extensions based on the annotation features of Java and C#. There are already major examples of such extensions, especially with distribution- and persistence-related annotations, implemented in the context of J2EE Application Servers. Such extensions can be implemented with heavyweight support—e.g., parsing files, or recognizing annotations in a class loader and performing bytecode manipulation. In fact, the JBoss AOP mechanism (in whose early design and implementation we have played an active role) is the foremost example of infrastructure used to implement annotation-based language extensions. Nevertheless, experience from compilers in general-purpose languages has shown that it is beneficial to develop a mature back-end language and implement high-level features by translating to that back-end. Our approach proposes that AspectJ is well-suited as such a back-end language for small Java language extensions and that generating AspectJ code offers significant simplicity benefits. In the future we plan to support this claim with more examples and perform a thorough comparison with competing mechanisms.

6. REFERENCES

[1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[3] Y. Smaragdakis. A personal outlook on generator research. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*. Springer-Verlag, 2004. LNCS 3016.

[4] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with meta-AspectJ. In *Generative Programming and Component Engineering (GPCE)*, pages 1–18. Springer-Verlag, October 2004.