Reusable Enterprise Metadata with Pattern-Based Structural Expressions

Eli Tilevich and Myoungkyu Song
Department of Computer Science
Virginia Tech, Blacksburg, VA 24060, USA
{tilevich,mksong}@cs.vt.edu

ABSTRACT

An essential part of modern enterprise software development is metadata. Mainstream metadata formats, including XML deployment descriptors and Java 5 annotations, suffer from a number of limitations that complicate the development and maintenance of enterprise applications. Their key problem is that they make it impossible to *reuse* metadata specifications not only across different applications but even across smaller program constructs such as classes or methods.

To provide better enterprise metadata, we present *pattern-based structural expressions (PBSE)*, a novel metadata representation that offers conciseness and maintainability advantages and is reusable. To apply PBSE to enterprise applications, we translate PBSE specifications to Java annotations, with annotating classes automatically as an intermediate build step. We demonstrate the advantages of the new metadata format by assessing its conciseness and reusability, as compared to XML and annotations, in the task of expressing metadata of J2EE reference applications and a mid-size, commercial, enterprise application.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.6 [**Programming Environments**]: Integrated environments; D.3.3 [**Language Constructs and Features**]: Frameworks, Patterns

General Terms

Design, Languages, Experimentation

Keywords

Frameworks, Metadata, Annotations, Configuration, Eclipse

1. INTRODUCTION

Automatic programming is the Holy Grail of software engineering. While generating a complete program automatically from a high-level description is still a futuristic vision, the declarative programming models of modern enterprise frameworks have taken the first steps toward this vision. A programmer expresses non-functional concerns, including persistence, transactions, distributions, and security using metadata, and frameworks provide the requested functionality. In a typical enterprise application, programmers implement the core functionality (i.e., business logic) by writing source code and non-functional concerns by declaring metadata.

Metadata identifies program constructs (e.g., classes, methods, fields, etc.) as interacting with framework services, which add the requested functionality by transparently injecting it into the identified program constructs. As an example, persistence frameworks automatically persist the fields of a class annotated with the `@Column` JPA [31] annotation. Programmers only need to declare which object fields correspond to relational database columns, and persistence frameworks supply all the required functionality to render the annotated fields persistent.

Enterprise metadata—a medium for expressing how program constructs interact with a framework—comes in different formats. One such format is XML, used for writing stand-alone XML configuration files, sometimes called deployment descriptors in J2EE [33]. More recently, metadata tags have been introduced into mainstream programming languages, including Java and C#. Java 5 annotations are a part of the source being placed near program constructs. Although in their latest releases, some enterprise frameworks have been changing their metadata format from XML to annotations, one can find examples of both metadata formats being used in modern enterprise applications.

Despite being the foundation of the declarative programming model, enterprise metadata has its share of maintenance and evolution burdens. Since XML configuration files refer to program construct names of the main source code written in Java or C#, such XML files must be kept consistent as the main program is maintained and evolved. If, for example, a name of a program construct changes, the change must be propagated to the XML configuration file, lest the mismatch causes some framework functionality to fail.

Though annotations are part of the source code, they also hinder program maintenance. Located right next to program constructs, annotations are not affected by the changes

AOSD '10 Rennes, St. Malo, France

to the constructs' names (e.g., renaming an annotated field does not affect its annotation). This tight coupling, however, is a double-edged sword—it hinders the transitioning between framework vendors that use different annotations to express equivalent functionality. For example, unit testing frameworks, such as JUnit 4 [29] and TestNG [4], use different annotations for test methods and other domain-specific information. Changing annotations scattered across an entire codebase is a tedious and error-prone undertaking, whose prohibitively high costs can cause the *Vendor Lock-in* anti-pattern [5].

Additionally, annotations cannot convey any structural information between programmer written code and framework functionality. For instance, the `@Column(name="someName")` annotation added to field "someName" only expresses that this one field and the database column to which it is persisted share the same name. Annotations cannot express that this invariant, for example, holds true for all the private fields of a class.

Finally, reuse—a major vehicle for increasing programmer productivity—is impossible with enterprise metadata, including both XML and annotations. XML descriptors are crafted individually for different classes. Annotations must be added separately not only to individual programs, but also to all the classes within the same program. Framework services are often used according to certain patterns that tend to be repetitive. For example, in many programs, all private fields may have to be made persistent. Such patterns cannot be encoded once and applied to multiple programs or even classes—each program or class must be annotated anew and its annotations maintained separately.

To address a lack of general reusability and to improve on other properties of enterprise metadata, this paper presents a new metadata format—Pattern-Based Structural Expressions (PBSE). By matching the structure of a program with pattern-based declarations, PBSE captures the relationship between metadata and the program's source code, making the metadata easier to author, understand, reuse, and maintain. PBSE is not only more concise than either XML or annotations, but PBSE specifications can be easily reused across different applications.

By addressing limitations of enterprise metadata, this paper presents the following novel contributions:

- A clear exposition of the advantages and shortcomings of mainstream enterprise metadata formats—XML and annotations.
- Pattern-Based Structural Expressions (PBSE)—a new metadata format that offers usability, reuse, and ease-of-evolution advantages, as compared to both XML and annotations.
- An automated translation approach that, given pattern-based structural expressions and their corresponding source files, can annotate the source with equivalent Java 5 annotations.

The remainder of this paper is structured as follows. Section 2 demonstrates the disadvantages of the mainstream enterprise metadata formats through an example from the transparent persistence domain. Section 3 introduces pattern-based structural expressions. Section 4 describes the PBSE language design and our automated PBSE-to-annotations translator. Section 5 quantifies the advantages of PBSE through case studies. Section 6 discusses the advantages of PBSE in comparison to XML and annotations. Section 7 compares this work to the existing state of the art. Section 8 discusses future work directions and presents concluding remarks.

2. MOTIVATION

Figure 1 demonstrates how modern enterprise frameworks use metadata, with the domain of transparent persistence as an example. On the left part of the figure, class `ManagerBean` is persisted using EJB 2, a persistence framework that uses XML configuration files to specify how instances of Enterprise Java Beans are mapped to relational database tables. On the right part of the figure, class `ManagerEJB`, with equivalent functionality to that of `ManagerBean`, is persisted using EJB 3, another persistence framework that uses Java 5 annotations as its metadata format. As we argue next, each of the two enterprise metadata formats has limitations that complicate the development and maintenance of enterprise applications that use frameworks.

2.1 Programmability

Using either metadata format to create correct specifications can be challenging. Authoring XML files with a text editor is cumbersome: the programmer must ensure not only the correctness of XML tags and grammar, but also the correspondence between the XML data and the program constructs of the persisted class. For example, each persisted class must be specified within `<entity> </entity>` tags, which is the root of an XML subtree with the descendant tags `<ejb--class>`, `<abstract-schema-name>`, `<field-name>`, `<primkey--field>`, etc. The immediate descendants of the `<entity>` tag may also have other descendants, making the authoring of such a tree-shaped structure quite error-prone. Omitting some tags may result in confusing compile and runtime errors. For example, forgetting to specify a `<primkey-field>` would render the entire specification invalid. Mistyping any field name would result in runtime errors, in response to the framework trying to access a non-existing field.

Annotations are more straightforward than XML files, because annotations are part of the Java language. Nevertheless, adding annotations according to a convention set by a particular framework may quickly become challenging. Even though a code completion facility of a modern Integrated Development Environment (IDE) could help programmers enter well-formed annotations with correct attributes, code completion cannot help them determine the value of string attributes (e.g., name in the `@Column` annotation) or identify where annotations should be added. Programmers must ascertain these requirements based on implicit framework conventions. For example, the `@Column` annotation can be added either to persistent field or to their getter methods according to the JavaBean naming convention, and these two approaches cannot be mixed.

One could argue that more sophisticated IDE support and better programming tools in general could simplify the au-

```java
public abstract class ManagerBean
  extends javax.ejb.EntityBean {
  public abstract String getOrderId ();
  public abstract String getStatus ();
  public abstract void setOrderId (String param);
  public abstract void setStatus (String param);
  ...
}
```

```xml
<entity>
  <ejb-class>ManagerBean</ejb-class>
  <abstract-schema-name>Manager
  </abstract-schema-name>
  <cmp-field><field-name>orderId
    </field-name></cmp-field>
  <cmp-field><field-name>status
    </field-name></cmp-field>
  <primkey-field>orderId
  </primkey-field>
  ...
</entity>
```

(1) Metadata as an XML file.

```java
@Entity
@Table(name="Manager")
public class ManagerEJB {
  private String orderId;
  private String status;

  @Id
  @Column(name="orderId", primaryKey=true)
  public String getOrderId(){
    return orderId;
  }
  @Column(name="status", primaryKey=false)
  public String getStatus(){
    return status;
  }
  public void setOrderId(String parm){
    orderId = parm;
  }
  public void setStatus(String parm){
    status = parm;
  }
}
```

(2) Metadata annotations.

Figure 1: Transparent Persistence Framework Example.

thoring of both XML files and annotations, but the very fact that such advanced support is required is a testament to the inherent complexity of expressing metadata using these formats.

2.2 Understandability

Consider a programmer assigned to take over an existing codebase written using an enterprise framework dependent on metadata. Examining XML files by hand is quite tedious. XML is a computer format optimized for automated processing rather than exposing information intuitively to the programmer. To understand how a framework implements some functionality, XML files must be examined with their corresponding source files. For example, to understand how instances of class `ManagerBean` are persisted, both its source and XML deployment descriptor must be examined.

While annotations ease program understanding in the small, programmers must examine the entire framework-dependent codebase. Annotations scattered around the codebase, provide no structural or summary information. For example, the invariant that all private fields of class `ManagerEJB` are persistent is not explicitly expressed. To determine this invariant, programmers must examine each getter method for the presence of the `@Column` annotation.

Sophisticated code analysis tools can certainly help programmers understand framework-dependent source code, but a more expressive metadata representation can render such analysis tools unnecessary.

2.3 Maintainability

Both metadata formats complicate maintenance. XML files are separate from the main source code, and their correspondences cannot be enforced by the compiler. If a source code change is not properly synchronized with the corresponding XML file, the problem will only be discovered at runtime. For example, if the `status` field in class `ManagerBean` is renamed to `orderStatus`, one must upgrade the corresponding entry in the XML file. This requirement, however, is implicit and depends entirely on the maintenance programmer.

Being a part of the language, annotations are easier to maintain. In the presence of structural changes, however, annotations must be added or removed accordingly. For example, when a field `time` added both to the database and the `ManagerEJB` class, this field or its getter method must be properly annotated, lest it will not be persisted. The invariant stating that all fields that share names with database columns must be persisted is implicit and cannot be enforced through annotations.

Maintenance of framework-based applications is a formidable challenge that has been the target of several recent research efforts [26, 35, 24]. These efforts, however, could be simplified if more expressive metadata could explicitly encode dependencies that must be preserved during program evolution.

2.4 Reusability

Both XML and annotation-based metadata representations are not reusable. The persistence information must be explicitly encoded for each class. In Hibernate 2 [2], another transparent persistence framework, a separate XML file must be used for each persistent class. In EJB 2, the same XML file is used for all the classes. Nevertheless, in both frameworks the persistence information is specified individually for each class. For example, the XML file for class `ManagerEJB` does not work with any other class. As a consequence, the knowledge about persisting `ManagerEJB` cannot be reused even for the classes in the same application.

Annotations do not improve reusability—each individual class must be annotated anew. The effort expended on annotating

```
public class ManagerEJB {
  private String orderId;
  private String status;

  public String getOrderId(){
    return orderId;
  }
  public String getStatus(){
    return status;
  }
  public void setOrderId(String parm){
    orderId = parm;
  }
  public void setStatus(String parm){
    status = parm;
  }
}
```

(1) Java code for PBSE metadata in (2).

```
 1 Metadata MyJPA<Package p>
 2   Class c in p
 3     Where (public *EJB)
 4       c += @Table
 5       @Table.name = (c.name =~ s/EJB$//)
 6       Column<c>
 7
 8 Metadata Column<Class c>
 9   Method m in c
10     Where (public * get* ())
11       m += @Column
12       @Column.name = (m.name =~ s/^get//^[A-Z]/[a-z]/)
13       Where (public * get*Id ())
14         @Column.primaryKey = true
15         m += @Id
16
17 MyJPA <"package1">
```

(2) PBSE metadata.

Figure 2: Transparent Persistence Framework PBSE Example.

a class cannot be leveraged in annotating other classes, due to annotations not being able to express any structural information. Each annotation provides information only about the annotated program construct. Annotations cannot express the relationships between the annotated elements (e.g., all the fields annotated with the same annotation). Further, annotations cannot even express a naming equivalence between its attributes and the annotated program constructs. For example, the attribute `name` of annotation `Column` directly corresponds to the name of the field of the getter method it is annotating. Because this invariant is only implicit, it cannot be reused in other contexts, such as different classes or another application.

Without sophisticated automated programming tools that utilize machine learning to extract such implicit invariants from metadata, enterprise metadata cannot be reused systematically. Since the precision of such programming tools tends to differ widely, existing enterprise metadata remains not reusable. A new metadata format designed with reusability in mind could improve programmer productivity.

3. PATTERN-BASED STRUCTURAL EXPRESSIONS

By examining metadata specifications of enterprise frameworks from different domains across multiple applications, we have observed that, in most cases, metadata is not added to a program randomly, but tends to follow well-defined patterns. It is this observation that led us to a new metadata format that is not only more concise and expressive, but also reusable. Our new metadata format, called Pattern-Based Structural Expressions (PBSE), is introduced by example next.

Consider the original motivating example from the transparent persistence domain. An equivalent PBSE specification in the right part of Figure 2 is applied to the Java code in the left part. PBSE reuses Java 5 annotations, declared as special Java interfaces, to define its own metadata specifications. A key difference is that PBSE metadata is declared in standalone specification files that are kept separate from the Java source files.

PBSE specifications are organized into modules, each representing metadata for a particular program construct. A PBSE module starts with the keyword `Metadata` followed by the module's name and its parameter declaration. Figure 2 (2) contains two PBSE modules. The first module is called `MyJPA`, and it defines Java Persistence API metadata for a package `p`. Line 2 iterates over all the classes in the package. To designate that only the classes that are `public` and whose suffix is "EJB" are to be persisted, a `Where` clause is used with the pattern parameter `public *EJB`.

The indented `Where` clause specifies the metadata information that should be applied to the classes that match the clause's pattern. Line 4 attaches `@Table` metadata to the matched class. PBSE uses the += operator to express the attaching of metadata to program constructs. Line 5 uses regular expressions[1] to assign the value of the class name without its "EJB" suffix to the `name` property of the `@Table` metadata. The =~ operator applies the regular expressions of its right operand to its left string operand. Line 6 invokes another PBSE module `Column` passing it the matched class as a parameter. All the invocations in PBSE are statically bound and are resolved by matching their names.

The `Column` PBSE module accepts a `Class` parameter and iterates over its methods (line 9). The `Where` clause on line 10 matches the getter methods following the JavaBean naming convention.[2] Line 11 attaches `@Column` metadata to the matched methods, and line 12 sets the `name` property of this metadata to the name of the method, having removed the "get" prefix and then changed the first letter to lowercase. Regular expressions are applied one after another from left

[1]We use the Perl language regular expression style due to its wide adoption.

[2]A more elaborate pattern could have filtered out the methods staring with "get" but returning `void`, e.g., `void getUpset()`.

to right, using the result of applying one expression as input to the next expression. Line 13 encodes the naming strategy for the getter methods which return persistent values, corresponding to the primary key of the underlying database table. The strategy in place assumes that the names of such getter methods will end with “id.” The methods matched by this `Where` clause have the `primaryKey` property of their `@Column` metadata set to `true`, and another metadata item, `@Id`, is attached.

Finally, line 17 applies the `MyJPA` metadata module to package “package1.” Thus, the persistence metadata will be attached to all the classes in this package. Further, since all persistent classes in an application usually share the same structure and naming conventions, `MyJPA` can be effortlessly reattached to other packages by adding another line of code (e.g., `MyJPA<“package2”>`).

3.1 Examples from Different Domains

The applicability of PBSE is not confined to persistence frameworks. We have discovered that enterprise frameworks commonly use structural patterns in their metadata. Not all of these frameworks use both XML and annotations as their metadata formats. Therefore, in the following we compare our PBSE specifications to whichever metadata format is used by a given example framework.

3.1.1 JUnit

Unit testing frameworks exercise test methods designated as such using metadata. A well-known and widely-used unit testing framework is JUnit [18], which uses `@Test` annotations to designate test methods, and `@Before` and `@After` annotations to designate the setup and tear down methods for each test class. The `@Test` annotation, in particular, has to be added to each and every test method, which can be wasteful, particularly if the number of test methods is large. Furthermore, neglecting to annotate a newly-added test method will result in missing tests.

```
Metadata MyJUnitSuite <Package p>
  Class c in p
    Where (public class Test*)
      MyJUnitTest <c>

Metadata MyJUnitTest <Class c>
  Method m in c
    Where (public void before* ())
      m += @Before
    Where (public void after* ())
      m += @After
    Where (public void test* ())
      m += @Test

MyJUnitSuite <"package1">
```

Figure 3: PBSE metadata for JUnit framework .

Figure 3 shows how JUnit metadata can be attached to all the test classes in a package, defined as all the public classes whose name starts with prefix “Test.” The `@Test`, `@Before`, and `@After` metadata are attached to the public methods returning `void` in the test methods based on their respective prefixes. A more general pattern could match the test methods whose name does not start with the “test” prefix.

```
1  Metadata MyTestNGSuite <Package p>
2    Class c in p
3      Where (public class Test*)
4        MyTestNG <c>
5
6  Metadata MyTestNG <Class c>
7    Method m in c
8      Where (public void before* ())
9        m += @BeforeMethod
10     Where (public void after* ())
11       m += @AfterMethod
12     Where (public void test* ())
13       m += @Test
14     Where (public void set* ( * ))
15       m += @Parameter
16       @Parameter.value = (m.name=~s/^set//)
17
18 MyTestingNG <"package1">
```

Figure 4: PBSE metadata for TestNG framework .

3.1.2 TestNG

TestNG [4] is another annotation-based unit testing framework, whose annotation set differs from that of JUnit. One could imagine how the necessity to change the annotations throughout an entire application from JUnit to TestNG or vice versa could preclude switching to another unit testing framework, even if such a switch is beneficial for technical reasons. Not changing vendors solely due to a prohibitive upgrade challenge is described by the Vendor Lock-in anti-pattern [5].

PBSE, being external to the main source code, removes this anti-pattern. Figure 4 shows the PBSE metadata specification for TestNG applied to the same set of test classes as in the JUnit example above. As a more recent unit testing framework, TestNG offers additional capabilities, among which is the ability to set custom parameters for test classes. The additional rule starts on line 14, which attaches the metadata `@Parameter` to setter methods (line 15), and sets its `value` property to the field name designated by the getter method (line 16), according to the JavaBean naming convention.

Thus, the same application can be tested with JUnit or TestNG simply by using a different PBSE specification.

3.1.3 The Security Annotation Framework

Another framework domain that can benefit from our pattern-based approach to expressing metadata is security. The security functionality of a typical enterprise application is divided into access control and encryption. An example of a security framework for enterprise applications is the Security Annotation Framework (SAF)[20]. SAF provides access control and encryption functionality, both of which are configured using Java 5 annotations. Methods can be granted read, update, create, and delete access. When the code to be secured with SAF follows a naming convention, the access can be granted based on patterns over method names rather than for each individual method.

Figure 5 shows the PBSE metadata security specification for a package in which classes are written according to the

```
Metadata MySecurity <Package p>
  Class c in p
    Where (public class *)
      c += @SecureObject
      MySecureObject <c>

Metadata MySecureObject <Class c>
  Method m in c
    Where (public * get* ())
      m += @Secure
      @Secure.SecureAction = READ
    Where (public void [set|add|remove]* ())
      m += @Secure
      @Secure.SecureAction = UPDATE
    Where (static public Object+ create* ()
      m += @Secure
      @Secure.SecureAction = CREATE
    Where (public * delete* (*)
      Parameter p in m
      p += @Secure
      Where ((Object+ *))
       p += (@Secure.SecureAction = DELETE)

MySecurity <"package1">
```

Figure 5: PBSE metadata for security framework .

```
1  Metadata MyWebService <Package p>
2    Class c in p
3      Where (public class *Impl)
4        c += @WebService
5        @WebService.name = (c.name =~ s/Impl$//)
6        Field f in c
7          Where (private * *)
8            f += @Autowired
9        Method m in c
10         Where (public * * ())
11           m += @WebMethod
12           @WebMethod.name = m.name
13
14 MyWebService <"package1">
```

Figure 6: PBSE metadata for Spring Web Service.

Java Bean naming convention. In addition, these classes have factory methods, which start with the "create" prefix. Finally, methods with the "delete" prefix deallocate systems resources passed to them as parameters.

The access control policy expressed by this specification controls access for every public class by using the `@SecureObject` metadata. The `@Secure` metadata and its `SecureAction` property are attached as follows. Every getter method is given the `READ` access, while every setter method as well as any method starting with prefixes "add" and "remove" are given the `WRITE` access. The `DELETE` access is given to reference parameters of the methods whose name starts with the "delete" prefix. We borrow the AspectJ syntax of `Object+` to express reference types.

Enforcing a consistent access policy requires that the entire codebase be annotated thoroughly, without any tolerance for omissions or mistakes. For example, giving the `UPDATE` permission to a wrong method may breach security. Naming conventions have become a mainstay of industrial software development to the degree that they are often enforced with automatic checkers. Integrated with a source control system, such an automatic checker can prevent committing any code edits that violate the naming convention in place. In light of that, applying a security policy based on structural patterns of the established naming convention is likely to prove more reliable than annotating methods individually.

3.1.4 Java Web Services

To support the ever-growing need for service-oriented applications, frameworks have been introduced to facilitate the exposition of regular classes as services. In particular, the Java Web Services (JWS) framework [27] provides a set of annotations that can be added to Java classes, methods, and fields, leaving it up to the underlying framework to provide the necessary plumbing to expose the annotated classes as externally-accessible Web services. If a class to be exposed as a Web service has many methods, each of them must be annotated individually.

Figure 6 shows the PBSE metadata specification that can be used to render all the public classes in a package as Web services. In particular, the logic required to annotate multiple classes and methods is expressed in only 12 lines of PBSE. The patterns expressed by this specification encode that the name of a Web service will differ from that of its corresponding class by the "impl" suffix (line 5). The `@Autowired` metadata is attached to all the private fields. And public methods are expressed as corresponding to Web service methods with the same names.

4. DESIGN AND IMPLEMENTATION

Having seen all the advantages of PBSE over annotations and XML, one may wish that PBSE becomes a new de-facto metadata standard. Such a transition, however, would require multiple divergent stakeholders of enterprise computing to come to a consensus. Thus, to make PBSE specifications immediately available to the enterprise programmer, we have implemented an automated translation tool that annotates Java source code based on its PBSE specification.

4.1 Language Summary

Figure 7 summarizes the syntax of PBSE. The language follows a minimalistic design, introducing new constructs only if necessary, with the goal of making it easier to learn and understand. For example, the class iterator can be used for iterating through both classes and interfaces of a package. PBSE expresses metadata declaratively and does not have explicit conditional or looping constructs. Nevertheless, a PBSE specification does contain a sufficient level of detail to describe the metadata information of a typical modern enterprise framework.

4.2 Translator Implementation

Our translator takes as input a PBSE specification and a collection of Java classes, and annotates the classes as guided by the specification. Since PBSE borrows its metadata definitions from Java 5 annotation interface declarations, the two formats correspond to each other closely. The translation process adds annotations by matching PBSE regular expression patterns against program constructs.

```
Metadata module_name
 <[Package|Class|Method|Field|Parameter]
 program_construct_variable>
   ...
   module_name<program_construct_variable>
   ...
   PBSE can call another module passing a parameter

[Class|Method|Field|Parameter] iter_var in collection
   An iterator for a collection of program constructs.

Where ([class_pattern | method_pattern |
      field_pattern | parameter_pattern])
   Patterns to match declarations of program constructs.

@Metadata
   Reflective metadata object.

@Metadata.property
   A property of a metadata object.

@Metadata.property = value
   Assign a value to the metadata's property.

~s/[^]original_value[$]/new_value/
   Substitute original_value with
   new_value, as specified by regular expressions.

program_construct_variable += @Metadata
   Add @Metadata to program_construct_variable.
```

Figure 7: PBSE constructs.

Figure 8 demonstrates the translator's process flow, in which input files—a set of Plain Old Java Objects (POJOs) [25] and PBSE metadata—parameterize the translator, and annotated POJOs serve as an intermediate build step. Once the automatically annotated files are compiled, they can be deployed to be executed by their respective frameworks—most modern frameworks do use annotations as their metadata format.

The PBSE translator processes a Java source file by using the Eclipse JDT API [7] to traverse the abstract syntax tree of a Java class, matching PBSE regular expressions against each encountered program construct's name including that of classes, methods, method parameters, and fields. If a construct's name is matched, the construct is annotated with the annotations corresponding to the metadata guarded by the `Where` clause of the regular expression.

4.3 Translation Semantics

Next we treat the translation process from PBSE to annotations more formally.

Figure 9 lists the symbols used in describing the translation process. The sets of program's structural constructs appear first. The structure of a program is defined by its classes, methods, method parameters, and fields, all of which are finite sets. Each of these structural constructs could be potentially annotated, and a set of available annotations appears as well. Each annotation is specific to the type of a program construct to which it can be added, including classes, methods, method parameters, and fields. The same annotation could potentially be used at multiple levels; for example, the `@Column` annotation can be applied to both methods and fields in the Java Persistence API.

c denotes a class
m denotes a method
f denotes a field
p denotes a parameter
a denotes an annotation

$A_C(c)$ denotes the set of annotations of class c
$A_M(m)$ denotes the set of annotations of method m
$A_P(p)$ denotes the set of annotations of parameter p
$A_F(f)$ denotes the set of annotations of field f

P_c denotes a class regular expression pattern
P_m denotes a method regular expression pattern
P_p denotes a parameter regular expression pattern
P_f denotes a field regular expression pattern

$RE(e, P_e)$ denotes a regular expression match of a language construct e over pattern P_e

Figure 9: Syntax definitions.

$$\frac{RE(c,\ P_c)\ \ (P_c,\ a) \in PBSE}{a \in A_C(c)}\ \text{[AnnotateClass]}$$

$$\frac{RE(m,\ P_m)\ \ (P_m,\ a) \in PBSE}{a \in A_M(m)}\ \text{[AnnotateMethod]}$$

$$\frac{RE(p,\ P_p)\ \ (P_p,\ a) \in PBSE}{a \in A_P(p)}\ \text{[AnnotateParameter]}$$

$$\frac{RE(f,\ P_f)\ \ (P_f,\ a) \in PBSE}{a \in A_F(f)}\ \text{[AnnotateField]}$$

Figure 10: Translation rules.

Each structural program construct can be matched with a regular expression pattern, which are formed by replacing some substring of a construct with a wildcard character (e.g., *) that can match multiple constructs. The regular expressions of PBSE `Where` clauses have been inspired by AspectJ pointcuts [15]. Figure 10 uses set operations to show how various constructs are annotated if they are matched by the PBSE regular expressions. Specifically, an annotation a is added to the annotations of a program construct e (i.e., class, method, parameter, or field) precisely when the construct matches a regular expression pattern, and the annotation a is attached to that pattern in the PBSE specification.

The presented translation semantics is a simplification, in that it describes only the main translation rules from PBSE to annotations. More complex features of PBSE, including nested patterns, module application, and substitutions, have been elided to save space and simplify the presentation.

4.4 Integration with Eclipse IDE

We have also prototyped the integration of the PBSE translator with the Eclipse Java code editor. Since PBSE metadata is entirely external to the main codebase, the programmer examining the source code may want to be informed about how the examined program constructs are related to

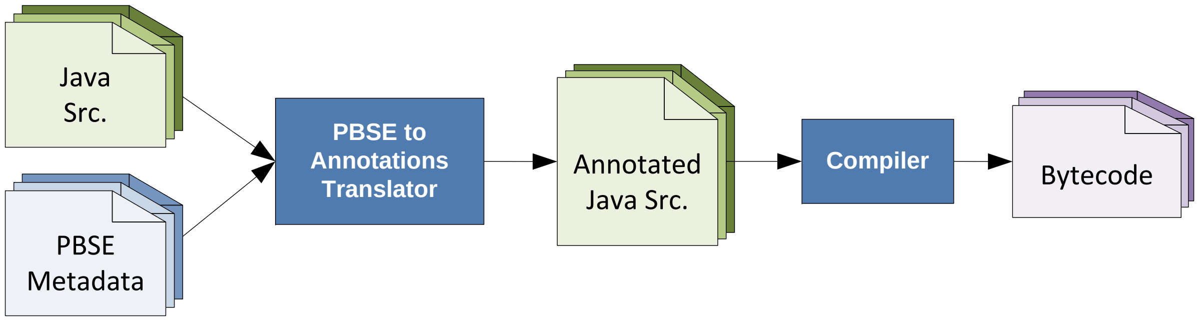

Figure 8: Translating PBSE to annotations as an intermediate build step.

metadata. Following the approach initially popularized by AspectJ, whose Eclipse plug in displays visual cues about AspectJ pointcuts affecting Java program constructs, our editor shows what we call *metadata tips* that inform the programmer about the metadata attached to the examined program constructs.

5. CASE STUDY

To validate the advantages of PBSE as compared to XML and annotations, we have conducted a case study with two J2EE reference applications and a medium sized commercial application. As our subject applications, we used JPetStore [32] and JAdventureBuilder(JAB) [30]—well-known reference applications that demonstrate various Java enterprise technologies. As a larger-scale subject application, we used the Prescription Monitoring Program(PMP)[3]—a real-world enterprise application built according to the J2EE [33] three tier architecture.

All three applications included transparent persistence functionality, implemented with EJB 2. This version of the framework employs XML configuration files as its metadata format. For our case study, we first upgraded our subject applications to EJB 3, which is based on annotations. Then, based on the annotated versions, we produced a corresponding PBSE metadata specification. As it turned out, all three of our subject applications needed only a single PBSE metadata specification, one written according to the structural source code patterns of the Java Persistence API (JPA) [31]. This specification, used in a prior example, appears in Figure 2 (2).

Table 1 compares the conciseness of each metadata format. As is expected, the total number of lines taken by both XML configuration files or annotations is directly proportional to the size of the application. Annotations manage to express the same metadata information significantly more concisely than XML (between 75% and 90% fewer lines of code on average). In addition to its reusability advantages, PBSE specifications are also quite concise. Compared to annotations, they express the same information in between 64% and 90% fewer lines of code on average. Of course, these numbers are dependent on the total size of the application, and PBSE leverages the fact that JPA metadata follows well-defined structural patterns.

[3]Developed by T4G (http://www.t4g.com).

Lines of code	JPetStore	JAB	PMP
Total source code	11,298	10,836	37,621
XML	503	538	890
Annotations	46	75	236
PBSE	17	17	17

Table 1: Conciseness of different metadata formats.

Reducing the amount of maintained hand-written source code offers tangible software engineering benefits. Since software complexity grows exponentially in relation to the size of a program [3], every line of programmer written code contributes to the software maintenance burden. Program changes to address new requirements or fix program defects require a program maintenance effort that is directly proportional to the size of a program [10].

6. DISCUSSION

Compared to both XML and annotations, PBSE provides programmability, understandability, maintenance, and reusability advantages, summarized next. PBSE also has certain applicability constraints that are discussed afterwards.

6.1 Programmability

PBSE expresses metadata concisely—a single pattern matches multiple program constructs, reducing the amount of metadata code that has to be written. We argue that PBSE will be fairly straightforward to learn for a developer familiar with object-oriented and declarative query (e.g., SQL) programming. Programming constructs such as iterators, `Where` clauses, and regular expressions, are part of the standard arsenal of a commercial software developer working with enterprise technologies. Finally, PBSE is more expressive than either XML or annotations, as they force the developer to understand and encode the structural correspondences between the main source code and its accompanying metadata.

6.2 Understandability

PBSE specifications capture the software architecture imposed by a given framework, something that neither XML or annotations can accomplish. XML was designed to facilitate the creation of automated parsers rather than to make XML documents easy to read and understand by a human.

In particular, the opening and closing XML tags often obfuscate the described data values and their relationship to one another. Annotations are easy to understand, but each individual annotation expresses only local information about its program construct; any correspondence between the name of an annotation and that of the program construct it annotates is only implicit. Any relationship between different program constructs annotated with the same annotation is only implicit also.

By contrast, by looking at a PBSE specification, programmers can understand the relationship between program constructs and their metadata. If the framework specific information can be encoded in PBSE specifications, programmers may not need to examine the source code—a short PBSE code snippet can capture complex invariants that hold throughout the entire codebase.

6.3 Maintainability

PBSE metadata alleviates the challenges of maintaining framework applications with respect to keeping the source code of an application consistent with metadata during program evolution. Although PBSE specifications are maintained separately from the main source code, their concise nature and the presence of explicit structural information make PBSE easier to maintain than either XML or annotations.

XML is disconnected from the main source code and must be evolved in parallel, thus doubling the maintenance programmer's burden. The location of the metadata information for a given program construct within an XML document is not immediately obvious, often taking a time consuming XML data exploration to discover. Because annotations remain next to the program construct they annotate, changing program construct names does not affect their annotations. However, newly added program constructs must also be annotated appropriately to ensure their proper interactions with the framework.

PBSE eliminates the Vendor-Lock In anti-pattern and preserves the correctness of metadata in the presence of program evolution, as long as a naming convention is followed. Switching to another framework to implement the same functionality is as straightforward as creating an alternative PBSE specification. A program can be enhanced with new methods, fields, and classes added or removed, and as long as the original naming convention is followed, no changes are required to keep the PBSE specification up to date. Having a PBSE specification to consult will make it less likely for the programmer to violate a naming convention when maintaining the code, as PBSE explicitly encodes the relationship between program constructs and metadata.

6.4 Reusability

PBSE metadata is reusable not only across different classes and packages within the same application, but also across different, and possibly unrelated, applications. By contrast, the existing metadata formats are not reusable. Neither XML or annotations are reusable, as they maintain a one-to-one relationship with the program constructs for which they provide metadata information. Annotations are particularly ill-equipped for reuse, as each program construct must be annotated individually.

The same PBSE metadata specifications can be reused in all the applications that use the same framework, as framework-dependent code is written to follow certain naming conventions. Furthermore, a PBSE specification can be easily modified for a different naming convention by changing only a few lines of code. For example, different prefixes or suffixes can be easily incorporated to accommodate the differences in naming conventions, and as long as the naming conventions are followed consistently, the slightly modified code can be fully reused.

6.5 Applicability Constraints

The advantages of PBSE are due to the structural patterns between the source code and metadata of modern enterprise framework applications. If, however, metadata is highly specialized for individual program constructs, without forming any patterns between the names of program constructs and their corresponding metadata, the utility and conciseness of PBSE can be compromised. For example, if every method or class in an application is annotated differently, the structural patterns of PBSE would not provide any conciseness or reusability advantages. One could still express such an application's metadata in PBSE, but each annotated program construct would require a separate `Where` PBSE clause.

The reliance on the naming conventions imposed by enterprise frameworks makes PBSE susceptible to what is known in AOP as the *fragile pointcut problem* [16]. Specifically, evolving a program can compromise the correctness of its PBSE declarations. However, the fragile pointcut problem of PBSE is alleviated by its reliance on the naming conventions and programming discipline imposed by enterprise frameworks.

As far as program evolution is concerned, program constructs could be added and removed, and their names could be changed. Removing a program construct or changing its name, within the confines of the naming convention in place, will not affect the correctness of PBSE declarations. If an added new construct is named according to a naming convention, it will be properly captured by PBSE. Annotations can make naming conventions optional. Take for example the difference between JUnit 3 and 4, the latter of which is annotation-based. With annotations, test methods no longer must start with "test", but can be named arbitrarily. One must ask, however, whether a test method's name should still contain the "test" substring. A naming convention that requires that a test method be named "testFoo" or "fooTest" improves readability, making the resulting code easier to understand and maintain. PBSE can capture such test methods irrespective of whether the naming convention in place uses "test" as the prefix or suffix.

Sometimes poorly-designed PBSE declarations can inadvertently capture program constructs that are not intended to be interacting with a framework. Consider expressing metadata as applicable to all getter methods and having non-getter methods, whose names start with "get"[4]. The PBSE examples introduced earlier would incorrectly capture such methods as getter. The PBSE declaration in Figure 11 avoids capturing such non-getter methods. Using a nested

[4]`void getUpset(), void getAlarmed(), etc.`

iteration over both fields and methods, this declaration defines a getter method as one whose name is a function of an existing private field's name.

```
Metadata Column<Class c>
 Field f in c
  Where (private * *)
   Method m in c
   Where(("get" + (f.name =~ s/^[a-z]/[A-Z]/)) == m.name)
    m += @Column
    @Column.name = f.name
    ...
```

Figure 11: Identifying getters for `private` fields.

Although enterprise frameworks follow well-defined naming conventions not just with respect to program constructs but also to metadata, sometimes the naming conventions are broken inadvertently. To ensure that the utility of PBSE is not compromised, approaches to dealing with the fragile-pointcut problem in AOP, including delta analysis [16, 28] and pointcut rejuvenation [14], can be adopted.

Another promising approach to this problem is to control how programs evolve to avoid unsafety and inconsistencies. In a recent work, Abdelmeged et al. [1] propose that correctness criteria be declared explicitly and define a stricter notion of compatibility to identify inconsistencies. In the same vein, one could express the naming conventions of enterprise frameworks explicitly (e.g., using a language extension) and verify them using an automatic checker. One could also execute PBSE declarations as a query against the underlying program and examine the number of matched program constructs. A stricter notion of compatibility can be expressed in terms of the expected delta in the number of matches, in response to evolving the program. For example, a JUnit notion of compatibility, *N_Test_Methods==(N_Test_Methods + N_New_Test_Methods)*, can protect against the unsafe evolution resulting from mistakenly naming the newly added test methods as starting with "tst" rather than "test."

Finally, although Java packages can be annotated, package annotations are rare and typically are specified in a special file whose name is fixed to `package-info.java`. Based on these observations, we chose not to support structural expressions over package annotations in PBSE.

7. RELATED WORK

Both AspectJ 5 [34] and JBoss AOP [13] provide language support for introducing annotations. AspectJ 5 does so thorough the `declare annotation` construct. JBoss AOP provides two ways to introduce annotations: using XML with the `annotation-introduction` tag and using a special meta-annotation. Both of these AOP languages provide wildcards to specify a set of program constructs to which annotations are introduced. Although these wildcards enable greater flexibility in expressing the set of annotated program constructs, the resulting pointcut expressions are not easily reusable. While one can express, for example, that a certain annotation be added to all the methods matching a certain name pattern, the resulting aspect program constructs are not parameterizable—one cannot reuse them with a different class. In addition, neither AspectJ 5 nor JBoss AOP make it possible to express annotation names and attributes as a function of the annotated program constructs. Overall, the pointcut expressions used in introducing annotations in AspectJ 5 and JBoss AOP do not capture the structural dependencies between the annotations and the annotated program constructs. As a result, neither of these AOP languages can express reusable enterprise framework metadata as concisely as can PBSE.

In addition, AspectJ 5 can match join points based on annotations by including pointcuts for all types of Java 5 annotations. The support of AspectJ for join point matching based on annotations may help manage some of the shortcomings of annotations as a metadata format. While AspectJ does not attempt to create a better metadata representation for enterprise frameworks, this was our intent behind creating PBSE.

PBSE is related to several research efforts whose objective is to validate the correctness of metadata. Eichberg et al. [8] check the correctness of annotation-based applications, in terms of their implementation restrictions and dependencies that are implied by annotations. The cases when the checked source code violates such restrictions and dependencies are reported by an automated, user-extensible tool. Noguera et al. [22] check the correctness of using annotations by adding to their declarations meta-annotations that define various constraints. Expressed as Object Constraint Language queries, these constraints must be satisfied when the declared annotations are used in the program. The definitions of annotation model constraints are validated at compile time by an automated tool. Cepa et al. [6] check the correctness of using custom attributes in .NET by providing meta-attributes that define dependencies between attributes. The attribute dependencies are expressed declaratively as a custom attribute and are checked using an automated tool.

These approaches are quite powerful and can catch many inconsistencies of using metadata. The need for these approaches, however, is a testament that the mainstream metadata formats, such as Java 5 annotations or .NET custom attributes, are not sufficiently expressive. This is the problem that PBSE aims to address. PBSE encodes the structural dependencies between program constructs and their corresponding metadata. By encoding such correspondences explicitly, PBSE specifications are less likely to contain unexpected inconsistencies and bugs. Furthermore, the declarative nature of PBSE makes it easier to ascertain complex metadata invariants by examining a single PBSE specification.

Pattern-based reflective declaration [9, 12, 11] extends C# and Java to declare program constructs such as fields and methods as a static, pattern-based, reflective iteration over other classes. Pattern-based reflective declaration is a metaprogramming technique for generating well-typed program constructs such as classes, methods, and fields. By contrast, PBSE is a new metadata format that uses patterns over the structure of program constructs to achieve conciseness and reusability.

Other more expressive metadata representations have been proposed in the literature. RDF [21]—an XML based metadata representation—improves network-based services such as the discovery and rating of resources. RDF associates values with properties and resources through a metadata schema. RDF provides flexibility and robustness advantages but is inapplicable to enterprise frameworks. SGF [17]—an XML based metadata representation—describes the structure of a web site to ease its navigation by creating interactive site maps; SGF also captures the semantic relationship between different web pages. The KNOWLEDGE GRID metadata [19] uses XML to represent information for managing the resources of a heterogeneous Grid, including computers, data, telecommunication, networks, and software. Orso, Harrold, and Rosenblum [23] discuss how metadata can be used to support a wide range of software engineering tasks with respect to distributed component-based systems. A particular focus of their work is testing and analysis of components. PBSE could simplify the analysis and testing of framework-based applications by capturing their architectural properties.

The complexity of enterprise metadata has motivated the creation of code generators that can automatically create metadata files based on higher level input. XDoclet, a popular open-source extensible code generator [36], is often used to automatically generate XML deployment descriptors for EJB from the source of a Java class. XDoclet works by parsing Java source files and special metadata tags (annotations inside Java comments) in the source code. Output is generated by using XDoclet template files that contain XML-style tags to access information from the source code.

The XDoclet metadata tags suffer from the same set of limitations as Java 5 annotations. Our translation tool described in Section 4 can be retargeted to output standalone XML deployment descriptors rather than annotated Java source files.

8. FUTURE WORK AND CONCLUSIONS

The applicability of PBSE is not limited to Java programs only. It can be applied to other languages with built-in metadata facilities such as C# and its attributes. What is more interesting is to apply PBSE to older languages such as C and C++ as an alternative to XML configuration files. PBSE can be used not only for frameworks, but as input for automatic code generators and adapters.

We plan to release our Eclipse Plug-in that automatically annotates Java code given a PBSE specification. The success of a new metadata format can only be ensured through a grassroots movement, with real enterprise developers trying out the new format and experiencing its benefits first hand.

In this paper, we have presented Pattern-Based Structural Expressions, a new metadata format that offers several software engineering advantages compared to the mainstream metadata formats such as XML and annotations. PBSE leverages the common patterns between the source code and its metadata exhibited by modern enterprise framework applications. By explicitly capturing and expressing these patterns, PBSE specifications convey metadata information concisely and can be reused not only within the same application, but also across other applications that use the same enterprise frameworks. Our evaluation using the domain of transparent persistence and the EJB 3 enterprise framework showed that PBSE can provide reusable metadata information significantly more concisely than either XML or annotations.

To increase the applicability of PBSE, we have implemented an automatic translation tool that annotates Java source code based on its PBSE specification. Our translation tool can also be useful for adding initial annotations to classes, even if it is not a developer's intention to use it continually. The ability to automatically add annotations to large codebases can reduce programmer burden.

As enterprise software development has become heavily dependent on frameworks for implementing most of the nonfunctional concerns, the role of metadata has gained prominence. Enterprise programmers spend a substantial amount of their time and efforts implementing and maintaining metadata. It is, therefore, important to take a close look at how metadata is expressed and whether improvement is possible. In that light, this work explores these difficult questions and proposes a new metadata format that improves on the existing state of the art.

ACKNOWLEDGMENTS

We would like to thank Ahmed Abdelmeged, Taweesup Apiwattanapong, Bryan Chadwick, John Edstrom, Karl J. Lieberherr, Barbara G. Ryder, Wesley Tansey, Dave Zook, and the anonymous AOSD reviewers for several helpful suggestions that improved the manuscript.

9. REFERENCES

[1] A. Abdelmeged, T. Skotiniotis, and K. J. Lieberherr. Controlled evolution of adaptive programs. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 89–98, New York, NY, USA, 2009. ACM.

[2] C. Bauer, G. King, and I. NetLibrary. *Hibernate in Action.* Manning, 2005.

[3] D. Berry. *Academic Legitimacy of the Software Engineering Discipline.* Carnegie-Mellon University, Software Engineering Institute, 1992.

[4] C. Beust and H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts.* Addison-Wesley Professional, 2007.

[5] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc. New York, NY, USA, 1998.

[6] V. Cepa and M. Mezini. Declaring and enforcing dependencies between .NET custom attributes. In *Generative Programming and Component Engineering*, pages 319–331. 2004.

[7] Eclipse Foundation. Eclipse Java development tools, March 2008. `http://www.eclipse.org/jdt`.

[8] M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *8th International Conference on Fundamental*

Approaches to Software Engineering (FASE 2005), volume 3442, pages 237–252. Springer, 2005.

[9] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, 2006.

[10] L. Gremillion. Determinants of program repair maintenance requirements. *Communications of the ACM*, 27(8):826–832, 1984.

[11] S. S. Huang and Y. Smaragdakis. Class morphing: Expressive and safe static reflection. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 79–89. ACM, June 2008.

[12] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424. Springer-Verlag, Aug. 2007.

[13] JBoss. JBoss AOP. `http://www.jboss.org/jbossaop/`.

[14] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 09)*, 2009.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*. Springer-Verlag, 2001.

[16] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[17] O. Liechti, M. Sifer, and T. Ichikawa. Structured graph format: XML metadata for describing Web site structure. *Computer Networks and ISDN Systems*, 30(1-7):11–21, 1998.

[18] V. Massol and T. Husted. *JUnit in Action*. Manning, 2004.

[19] C. Mastroianni, D. Talia, and P. Trunfio. Managing heterogeneous resources in data mining applications on grids using XML-based metadata. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 11, 2003.

[20] Maven. Security Annotation Framework. `http://safr.sourceforge.net/`.

[21] E. Miller. An introduction to the resource description framework. *Journal of Library Administration*, 34(3):245–255, 2001.

[22] C. Noguera and L. Duchien. Annotation framework validation using domain models. *Lecture Notes in Computer Science*, 5095:48–62, 2008.

[23] A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*. Springer.

[24] J. H. Perkins. Automatically generating refactorings to support API evolution. In *PASTE '05: Proceedings of the 6^{th} ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 111–114, New York, NY, USA, 2005. ACM.

[25] C. Richardson. Untangling enterprise Java. *ACM Queue*, 4(5):36–44, 2006.

[26] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[27] Spring. Java web service. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf`.

[28] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656. Citeseer, 2005.

[29] R. Stuckert. JUnit reloaded, December 2006. `http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html`.

[30] Sun Developer Network. Java Adventure Builder Reference application. `http://java.sun.com/developer/releases/adventure/`.

[31] Sun Developer Network. Java Persistence API. `http://java.sun.com/javaee/technologies/persistence.jsp`.

[32] Sun Developer Network. Java Pet Store 2.0 reference application. `http://java.sun.com/developer/releases/petstore/petstore1_3_1_02.html`.

[33] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition (J2EE), 2003.

[34] the AspectJ Team. The AspectJ 5 development kit developer's notebook. `http://eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[35] T. Tourwé and T. Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[36] C. Walls, N. Richards, and R. Oberg. *XDoclet in action*. Manning, 2004.