

Co-evolving Annotations and Source Code through Smart Annotations

Andy Kellens, Carlos Noguera, Kris De Schutter, Coen De Roover, Theo D’Hondt

Software Languages Lab

Vrije Universiteit Brussel

Pleinlaan 2

B-1050 Brussels, Belgium

{akellens | cnoguera | kdeschut | cderoove | tjdhondt}@vub.ac.be

Abstract—Annotations are a means to attach additional meta data to the source code of a system. Nowadays, more and more technologies rely on the presence of such annotations in the source code: beyond their use for documentation purposes, annotations impact the behaviour of the system. Since there exists little or no support to make sure that upon evolution of the system, the source code remains correctly annotated, source code can become miss-annotated. This in turn, can result in erroneous behaviour. In this paper we present Smart Annotations, an approach for co-evolving source code and annotations. Our approach enables developers to constrain the use of annotations in the source code and offers tool support to identify conflicts between source code and annotations. To illustrate the use of our approach, we demonstrate its applicability using examples from the domain of aspect-oriented programming and Enterprise Java Beans.

I. INTRODUCTION

Many modern programming languages include the concept of annotations, as is the case with .Net’s attributes or Java’s own annotations. Annotations are a means to introduce additional meta data at the level of the various entities (classes, methods, fields, ...) that are present in a program’s source code. The introduced meta data can communicate a developer’s intent, or it can be leveraged by software engineering tools. The *Doxygen* tool, for instance, generates web pages that document source code based on its annotations.

Since their inception, the use of annotations has gained popularity within the software engineering community. More and more, annotations are becoming an active part of the system. Rather than being used solely for documentation purposes, annotations are used to attach semantical properties to the source code of the system. Examples of such uses are found in annotation-based frameworks, aspect-oriented programming and pluggable type systems. Frameworks that are widely used within the Enterprise Java Bean world, such as Hibernate [1] and Spring [2], rely on annotations to know at which places in the source code they need to intervene. Similarly, within the aspect-oriented programming community, annotations have been proposed to tackle the so-called *fragile pointcut problem* [3] [4]. Within aspect-oriented programming, pointcut expressions are used to define at which places during the execution of the program an

aspect should be triggered. Since there exists a tight coupling between such pointcut expressions and the structure of the source code, seemingly safe changes to the source code of a system can result in erratic behaviour of the aspects. By using annotations, one can decouple these aspects from the structure of the source code, making them less fragile with respect to evolution. A last example of technology that relies on annotations are pluggable type systems. These systems allow imposing additional constraints over the source code of a system through annotations. If a particular field in the system is not allowed to contain a `null` value, for example, a developer can express this constraint by annotating the field. A pre-compiler enforces the expressed constraints by pointing out their violations.

Despite their benefits, annotations are fragile with respect to evolution of the source code they annotate. Whenever changes are made to the source code, existing annotations may no longer apply. Likewise, newly introduced source code entities may lack a required annotation. So far, development environments do not provide any support to guarantee that the source code remains annotated correctly whenever it evolves. It is therefore possible that a developer fails to annotate a particular source code entity, or that annotations are placed on the wrong entities. As a system’s behaviour can depend on whether or not its source code has been annotated correctly, such annotation fragility can result in erratic system behavior.

In this paper we present *Smart Annotations*, a tool for *co-evolving* [5] source code and annotations. The goal of our tool is to keep annotations and source code consistent upon evolution of either artefact. *Smart Annotations* achieves this goal by allowing developers to constrain the use of annotations in a system’s source code. To this end, it offers a declarative language in which developers can express constraints on where annotations are applicable. Tool support is offered to verify the validity of the annotated code with respect to the specified constraints. This way, developers can be informed about possible inconsistencies between the source code and the annotations.

This paper is structured as follows. In Section II, we introduce *Smart Annotations*. We discuss the declarative language in which constraints can be expressed and the meta-

```

1 public class AccessorExample {
2   public Integer my_field;
3   public Integer another_field;
4
5   @Getter("my_field") //Correctly annotated
6   public Integer getMy_field() {
7     return my_field;
8
9   //Annotation forgotten
10  public Integer getAnother_field()
11  {
12    return another_field;
13
14  @Getter("my_field") //Incorrectly annotated
15  public Integer getMy_field(Integer my_field)
16  {
17    return my_field;
18 }

```

Figure 1. A simple Java example using annotations.

annotations that allow configuring how these constraints are to be enforced. In Section II-E, we discuss the prototype implementation of *Smart Annotations*. To demonstrate its utility, Section III applies our approach as a means to cope with aspect fragility and as a means to validate persistency specifications of Enterprise Java Beans. We discuss the advantages and limitations of our approach in Section IV and position it with respect to existing work in Section V. Section VI concludes our exposition.

II. SMART ANNOTATIONS

We introduce the concepts underlying *Smart Annotations* using a straightforward running example. Figure 1 depicts a Java class named `AccessorExample` with two fields and three methods. We want to annotate all getter methods of this class with the `Getter` annotation.

Method `getMy_field()` (lines 5–7) corresponds to the prototypical implementation of a getter for the `my_field` field. It follows the Java naming convention (namely that the method name starts with the prefix ‘get’) and returns the value of the field `my_field`. It has therefore been annotated correctly with the annotation `@Getter("my_field")`.

Method `getAnother_field()` (lines 10–12) is a getter method for field `another_field`. However, it has not been annotated as such. Clearly, this is incorrect.

Finally, the method named `getMy_field(Integer my_field)` is depicted on lines 15–17. This method consists of a single statement that returns the method’s parameter `my_field`. While this method is not a getter method (the parameter shadows the field of the same name), it has been annotated as one. Clearly, this is incorrect as well.

The *Smart Annotations* tool enables enforcing constraints on the use of annotations in the source code of a system. To do this, the source code entities to which an annotation is applicable need to be characterized first. For our running

```

1 @Target (ElementType.METHOD)
2 @TargetVariable ("method")
3 public @interface Getter {
4   @Necessary
5   public static final String NAMING_CONVENTION
6     = "?method methodDeclarationHasName: {get*}";
7
8   @Sufficient
9   @Necessary
10  public static final String STRUCTURAL =
11    "?method isMethodDeclaration, " +
12    "?class definesMethod: ?method, " +
13    "?class definesVariable: ?field, " +
14    "?method returns: ?field";
15
16  @TargetVariable ("field")
17  String value ();
18 }

```

Figure 2. Smart Annotations applied to the `Getter` annotation.

example, this entails characterizing the concept of a getter method. For instance, by stating that getter methods start with the prefix ‘get’ (i.e. they follow the typical naming convention). Or by stating that all getter methods are methods that return the value of a field. In the next section, we show how these characterisations can be used to impose constraints on the use of the `Getter` annotation.

A. Declaring Smart Annotations

Figure 2 illustrates how the `Getter` annotation can be specified using *Smart Annotations*. Note that our approach does not require an extension of the Java language, but rather is compatible with standard Java annotations. In Java, an annotation is represented as a special kind of interface declaration (called an annotation interface). *Smart Annotations* integrates directly with these annotation interfaces and does not require any new language constructs. A Java annotation interface is made ‘Smart’ by using meta-annotations and by embedding logic queries into its declaration. While the latter express constraints on the use of the annotation, the former specify how these constraints are to be enforced. We will start by explaining the logic language in which the annotation constraints are described.

B. Logic Program Queries

Constraints in *Smart Annotations* are expressed as logic queries. Each query describes the characteristics that should be exhibited by annotated source code entities. Each use of an annotation is expected to satisfy each of the annotation’s queries. Annotated entities that do not satisfy a query are flagged as violations of the constraint.

Logic queries are embedded into the annotation interface by means of static fields. For the `Getter` annotation, Figure 2 defines two logic queries that characterize getter methods. The queries correspond to the two different characterisations of a getter method identified above: by means

of the convention after which they are named or by means of their structural shape.

Smart Annotations uses SOUL [6] as its language to express logic queries in. SOUL is a Prolog-like programming language with specialized features for reasoning about the source code of Smalltalk [7], [8], Java [8], [9], C [10] and Cobol [11] programs. The syntax of SOUL differs slightly from the one of Prolog.

To illustrate these differences, consider the NAMING_CONVENTION query (lines 5–6) in our toy example. This query consists of a single condition that expresses that the name of the annotated method should start with the prefix ‘get’. The first difference with Prolog is that variable names in SOUL are indicated by means of a question mark. The naming convention query contains one logic variable, namely `?method`¹. The second syntactic difference with Prolog is the fact that we do not use a classic predicate logic notation, but rather a Smalltalk-like keyword-based notation. This notation improves the readability of the resulting logic queries. The constraint on lines 5–6 uses one predicate, namely `methodDeclarationHasName:` which verifies whether its first argument (in our case the variable `?method`) is a method declaration whose name matches the second argument (the pattern ‘get*’).

The second query (lines 9–14) characterizes getter methods by means of their structural shape. It consists of four logic conditions. The first condition (line 11) checks whether the binding for variable `?method` is a method declaration. The second condition (line 12) subsequently binds variable `?class` to the class that defines this method. The third condition (line 13) verifies whether this class also defines a field `?field`. Finally, the last condition verifies that the return statement of the method `?method` returns the value of the field `?field`.

Logic program queries enable characterizing software entities in a declarative manner. Developers can focus on *what* they want to compute (i.e. an entity’s characteristics) instead of *how* this is to be computed (i.e. the operational search for each characteristic). The same predicate can furthermore be used in a condition that checks whether the bindings for its variables are in a particular relation, as well as in conditions that quantify over this relation (i.e. establish bindings for unbound variables). The queries of our example can therefore be used not only to check whether an annotated method exhibits the characteristics of a getter method, but also to retrieve all methods that exhibit these characteristics and should therefore be annotated. For a more in-depth discussion of the advantages of logic program queries, and other applications of the technique, we refer to [12].

¹This variable, as will be explained in the next section, will be bound to the method actually annotated. We specified this using the `@TargetVariable` annotation in line 2 of the example.

SOUL provides a library of predicates for reasoning over Java programs [8], [9]. These predicates quantify over an abstract syntax tree representation of the program which stems from the Eclipse JDT Core Component [13]. For each subclass of `org.eclipse.jdt.core.dom.ASTNode`, the library provides a unary predicate (e.g. `isClassDeclaration/1`) that quantifies over all nodes of this kind. Binary predicates (e.g. `classDeclarationHasName:/2`) quantify over the relations between each node and its children. In addition, the library provides higher-level predicates that quantify over relations between AST nodes that are not explicit in the AST representation. Examples include the aforementioned binary predicate `returns:/2` and binary predicate `inheritsFrom:/2` which quantifies over the inheritance relation between types. Table I lists some of the predicates in this library.

C. Meta-Annotations for Smart Annotations

Smart annotations are declared in three steps: specifying the annotation scope, embedding the constraints of the annotation (i.e. its logic queries), and binding the values of the annotation to logic variables. Specifying the scope of the annotation entails stating to which types of source code entities the annotation is applicable. Line 1 of Figure 2 illustrates that this is achieved by using the `@Target` annotation, which is part of the standard Java annotation system. For our running example, we specified that the `Getter` annotation is applicable only to methods. An annotation’s usage constraints are specified by embedding logic queries into the annotation interface using `static String` fields. The queries that characterize getter methods by means of their name and by means of their structural shape are assigned to the fields `NAMING_CONVENTION` and `STRUCTURAL` respectively.

Through a meta-annotation called `@TargetVariable`, developers can provide information about an annotation’s context of use to the logic queries. On line 2 of Figure 2, the entire annotation interface is annotated using `@TargetVariable("method")`. This meta-annotation states that the actual annotated methods can be accessed from within the logic queries through a variable named *method*. In our example, the `@Getter` annotation has a single parameter representing the name of the accessed field. The values of these parameters may be necessary in the definition of a logic query. The `@TargetVariable` meta-annotation can therefore also be used to bind logic variables to the value of one of an annotation’s parameters. Line 15 of Figure 2 uses a `@TargetVariable` annotation to bind the value of the `@Getter` annotation to the logic variable *field*.

Smart Annotations supports two kinds of constraints, namely sufficient and necessary constraints. A developer in-

Predicate	Description
<i>?class</i> isClassDeclaration	if <i>?class</i> is a class declaration
<i>?class</i> classDeclarationHasName: ?name	if <i>?name</i> is the name of the class declaration <i>?class</i>
<i>?class</i> isAnonymousClassDeclaration	if <i>?class</i> is an anonymous class declaration
<i>?type</i> inheritsFrom: ?anotherType	if <i>?type</i> inherits from <i>?anotherType</i>
<i>?class</i> definesConstructor: ?const	if <i>?class</i> defines a constructor <i>?const</i>
<i>?class</i> definesVariable: ?var	if <i>?var</i> is a variable (field) declared by <i>?class</i>
<i>?class</i> classDeclarationHasAnnotation: ?a named: ?aName	if an annotation <i>?a</i> , named <i>?aName</i> is placed on <i>?class</i>
<i>?anno</i> annotationHasValue: ?val	if the annotation <i>?anno</i> has a member whose value is <i>?val</i>
<i>?meth</i> methodDeclarationHasBody: ?body	if the block <i>?body</i> is the body of the method <i>?meth</i>
<i>?meth</i> returns: ?exp	if a method declaration <i>?meth</i> returns an expression <i>?exp</i>
<i>?if</i> ifStatementHasExpression: ?exp	if the expression <i>?exp</i> is the condition for a Java if-statement <i>?if</i>

Table 1
SAMPLE OF SOUL PREDICATES FOR REASONING ABOUT JAVA SOURCE CODE.

icates the kind of constraint a logic query represents by annotating the query with `@Sufficient` or `@Necessary`.

@Necessary: When a query is annotated as necessary, this means that every annotated location in the source code should exhibit the characteristics it describes. In our example, we have annotated the naming-based query as necessary. As a result, all method declarations that are annotated as a getter method should start with the prefix ‘get’. Necessary queries allow identifying source code entities that have been annotated improperly.

@Sufficient: Sufficient queries are dual to necessary queries. If a query is annotated as being sufficient, this implies that all entities that exhibit the specified characteristics should also be annotated. We have annotated the second query of our example, `STRUCTURAL`, as being sufficient. This means that all method declarations that return the value of a field should also be annotated with the `@Getter` annotation. Sufficient queries allow identifying source code entities that are missing an applicable annotation. In our example, this is the case for the `getAnother_field()` method. Note that a single query can be annotated with both the `@Necessary` and the `@Sufficient` annotation. In our example, this is the case for the `STRUCTURAL` query. As a result, all entities that exhibit the characteristics described by this query should be annotated with the `@Getter` annotation. Conversely, all entities that are annotated with the `@Getter` should exhibit the characteristics described by this query.

D. Declaring exceptions on the constraints

Sometimes, the usage constraints of an annotation can have been declared too generic or specific. Also, particular source code entities to which an annotation is applicable might be exceptions to the general rule. In these cases, verification of the annotation should not flag these exceptions as violations. *Smart Annotations* explicitly supports such exceptional cases. Two annotations are provided to this end; namely `@DoesApply` and `@DoesNotApply`. These annotations take a single argument (an annotation interface)

```

1 @DoesNotApply(Getter.class)
2 public Integer getAnother_field()
3 {
4     return another_field;}
5
6 @DoesApply(Getter.class)
7 @Getter("my_field")
8 public Integer doNothing()
9 {
10    return null;}

```

Figure 3. Example of the use of the `@DoesApply` and `@DoesNotApply` annotations.

and can be used to explicitly indicate that the annotation interface does, or does not apply for a source-code entity.

Figure 3 demonstrates the definition of exceptions to our running example:

- The first method of Figure 3 is the `getAnother_field()` method introduced above. While this method is conceptually a getter method, we forgot to annotate it as such. It therefore triggers a violation of the constraints that govern the use of the `@Getter` annotation. Now suppose that method `getAnother_field()` is actually an exception to the rule: we do not want the `@Getter` annotation to apply, even though the method exhibits all sufficient characteristics. We can achieve this by annotating the method with the `@DoesNotApply(Getter.class)` annotation. As a result, the method will no longer be indicated as an entity that is missing the annotation.
- Conversely, using the `@DoesApply(Getter.class)` annotation, we can also indicate that a particular annotation does apply to a source code entity—even though the entity does not comply with the annotation’s usage constraints. This is illustrated by the method `doNothing()` on lines 8-10 in Figure 3. While this method is conceptually not a Getter method, it is annotated as such. Since it neither matches the naming convention nor the structural

pattern of the `Getter` annotation, it is flagged as an exception using *Smart Annotations*. However, by using the `@DoesApply` annotation, we can still document this method as an exception.

Note the `@SuppressWarnings` annotation can be leveraged to indicate that any violations occurring at a particular source code entity should be ignored. Instead, annotations `@DoesApply` and `@DoesNotApply` make it explicit to developers that a particular source code entity has been documented as being an exception to the annotation's usage constraints.

E. Tool support

We provide rudimentary tool support for *Smart Annotations* by means of an Eclipse plugin. From within this Eclipse plugin, a developer can trigger a verification of the annotation constraints with respect to the source code of a system. Feedback is given by means of the default warning system of Eclipse: if a source code entity violates a constraint, this will be reported both in the list of warnings as well as by means of a marker in front of the line in the code that violates the annotation constraint. Furthermore, we also provide two quick fixes that allow for adding/removing a particular annotation and declaring a particular source code entity as an exception to the constraints for a particular annotation.

III. EXAMPLES

In order to illustrate the utility of Smart Annotations, we consider two possible applications: the use of annotation constraints to tackle aspect fragility in AJHotDraw, and the validation of Enterprise Java Beans annotations for persistence using the Java Persistence API (JPA).

A. Tackling aspect fragility

As a first example of the use of Smart Annotations, we take a look at an example from the domain of aspect-oriented programming. We already mentioned in the introduction that aspects are a new kind of module that aims at the modularization of cross-cutting concerns. To this end, aspects employ the concept of pointcut expressions: expressions that capture particular points in the execution of a system, at which time aspects can alter the behaviour of the system. As a concrete case study, we consider the AJHotDraw system [14]. AJHotdraw is an aspect-oriented refactoring of the popular JHotDraw open-source drawing application that was developed originally as a demonstrator for design patterns and object-oriented frameworks. AJHotDraw refactors some of JHotDraw's crosscutting concerns into aspects using AspectJ [15]. In this paper we take a look at one particular such crosscutting concern, namely the validation of the contract for the execution of commands. Within JHotDraw, all actions are implemented by means of a Command Pattern [16]. Before the actual action is allowed

to be executed, it should be verified that the view on which the command is to operate is not null. Within JHotDraw, this check is performed by all non-anonymous implementations of a command. Since it is scattered throughout the entire command implementation, it is a suitable candidate to be expressed using aspects.

Figure 4 shows part of the aspect-based implementation of this concern taken from AJHotDraw. Without going into too much details, we see the pointcut selects the execution of all `execute()` methods in the hierarchy of `AbstractCommand`. Since this expression would also capture the execution of anonymous classes (for which the contract verification should not happen), the pointcut explicitly excludes these anonymous classes means of `!within` primitive pointcut expressions: all the classes that contain anonymous classes are enumerated within the pointcut as `a` means to prevent the aspect from intervening at these classes. It is not hard to see that this pointcut expression is not robust with respect to evolution: if a developer adds a new anonymous class in the `AbstractCommand` hierarchy without also updating the pointcut, the pointcut will erroneously capture this new class, possibly leading to erratic behaviour. This problem, where seemingly safe changes to the base code of an aspect-oriented system can have an unexpected and unwanted effect is dubbed the fragile pointcut problem [4], [17].

This problem of aspect fragility is well documented, and annotations have been proposed as a way to deal with this problem [18], [19]. Instead of directly referring to actual source-code entities from within the pointcut, such schemes propose to write the pointcut expression in terms of annotations that are present in the source code. If we have correctly annotated for example all the `Commands` in JHotDraw using the `@CommandClass` annotation, we can rewrite the above pointcut (see Figure 5). This pointcut no longer needs to list an explicit list of exceptions and thus avoids the fragility of the original pointcut. Instead, it captures the execution of all `execute()` methods within a class that is annotated as being a `Command`.

However, the pointcut in Figure 5 can still suffer from the fragile pointcut problem: while the pointcut itself is decoupled from the implementation details of the base code, it relies on the fact that the source code is correctly annotated. If this however is not the case (e.g. if a developer forgot to annotate a particular `Command`), then the pointcut is equally brittle as the original AJHotDraw pointcut. In other words, the use of annotations to decouple a pointcut from the base code's structure does not solve the fragile pointcut problem; it only shifts the problem to another level of abstraction.

In order to overcome this problem, we propose to use a Smart Annotation `CommandClass`, shown in figure 6. This marker annotation will be placed on all classes that extend `AbstractCommand`, and are not anonymous. Thus

```

1 public aspect CommandContracts {
2     //Check the view's reference before command execution
3     pointcut commandExecuteCheckView(AbstractCommand acommand) :this(acommand)
4         && execution(void AbstractCommand+.execute())
5
6     //exclude the anonymous commands - no clean way to do it, so go for the enclosing types
7     && !within(*..DrawApplication.*)
8     && !within(*..CTXWindowMenu.*)
9     && !within(*..WindowMenu.*)
10    && !within(*..JavaDrawApp.*);
11 }

```

Figure 4. CommandContracts aspect

```

1 pointcut commandExecuteCheckView(
2     AbstractCommand acommand)
3     :this(acommand)
4     && execution(void AbstractCommand+.execute())
5     && within(@CommandClass)

```

Figure 5. CommandContracts pointcut using annotations

```

1 @Target(ElementType.TYPE)
2 @TargetVariable("item")
3 public @interface CommandClass {
4     @Sufficient
5     @Necessary
6     public static String COMMAND_RULE =
7         "?command isClassDeclaration, " +
8         "?command classDeclarationHasName:
9             {AbstractCommand}," +
10        "?item isClassDeclaration, " +
11        "?item inheritsFrom:?command, " +
12        "not(?item isAnonymousClassDeclaration)";
13 }

```

Figure 6. Source code of the @CommandClass Smart Annotation

a sufficient and necessary usage constraint is embedded in the annotation's interface definition (lines 4 – 12). The query takes all subclasses of the `AbstractCommand` class, and selects those that are not anonymous. If a class is annotated as being a `Command` class, but it is not in the correct hierarchy, or it is an anonymous class, Smart Annotations will flag it as a violation. Conversely, if a developer accidentally omits the annotation of a non-anonymous `Command` class, this will be detected using the Smart Annotation and reported to the developer. In combination with the annotation-based pointcut depicted in Figure 5, we are able to identify evolutions of the base code that would lead to aspect fragility.

B. Enterprise Java Beans

Enterprise Java Beans (EJBs) is one of the most popular component frameworks for business application development in the Java ecosystem. In its third version, annotations are introduced as an additional way to encode the meta data required by the framework for the configuration of the

services provided by the EJB runtime. One of the services that heavily relies on annotations is that of persistence. This service is known as the Java Persistence API [20]. It consists of the set of annotations defined in the EJB3 specification that deal with the Object-Relational mapping for entities. The JPA defines 64 annotations that are used by application developers to specify how their entities will be persisted in a database. The correct use of the annotations defined in the JPA depends on the developer's knowledge of the complex constraints that involve not only properties of the annotated source-code entities, but also other annotations present in the application. We will illustrate the utility of Smart Annotations by embedding usage constraints on three JPA-defined annotations: `@Entity`, `@Table` and `@AttributeOverride`. These three annotations, as well as their usage constraints, are explained below.

`@Entity`: Classes that carry the `@Entity` annotation represent entity beans in the EJB framework. Entities are mappable to tables when persisted. Each entity has a name which defaults to the name of the class. The JPA runtime relies on Java reflection to map entities to tables in the database. Because of this, the JPA specification imposes a number of restrictions on the modifiers and characteristics of the entity classes. In particular, it is required that:

- 1) Entity classes must define a public or protected constructor without arguments;
- 2) Entity classes cannot be final;
- 3) Methods belonging to entity classes must not be final.

The source code of the `@Entity` annotation interface is shown in figure 7. In it, each constraint is translated into a *necessary* query for uses of annotation to be correct. First off, the class on which the `@Entity` annotation is placed is bound to the `?class` logic variable using the `@TargetVariable` annotation on line 2. The first constraint is expressed in the `NO_ARGS_CONSTRUCTOR` (Lines 5 – 11) logic query. The query is divided into two parts: lines 7 – 9 verify that there exists a constructor for `?class` that does not have any arguments; lines 10 and 11 check whether the constructor is either `public` or `protected`. The second constraint (lines 13–16) does a similar check to ensure that the class is not declared `final`.

```

1 @Target (ElementType.TYPE)
2 @TargetVariable("class")
3 public @interface Entity {
4
5     @Necessary
6     public static final String NO_ARG_CONSTRUCTOR =
7         "?class definesConstructor: ?c," +
8         "?c methodDeclarationHasParameters: ?params," +
9         "?params isEmpty," +
10        "?c methodDeclarationHasModifiers: ?mods," +
11        "or(?mods isPublic, ?mods isProtected)";
12
13     @Necessary
14     public static final String NO_FINAL_CLASS =
15         "?class classDeclarationHasModifiers: ?mods," +
16         "not(?mods isFinal)";
17
18     @Necessary
19     public static final String NO_FINAL_METHODS =
20         "forall(?class definesMethod: ?meth," +
21         "and(
22             ?meth methodDeclarationHasModifiers: ?mods,
23             not(?mods isFinal)" +
24         "));";
25
26     @Sufficient
27     public static final String TABLE_REQUIRES_ENTITY=
28         "?class classDeclarationHasAnnotation: ?
29         named: {Table}";
30
31     String name() default "";
32 }

```

Figure 7. @Entity Smart Annotation

Finally, the third constraint (the `NO_FINAL_METHODS` field) uses SOUL’s *forall* predicate to iterate over the methods that are declared in *?class* and checking that each one is not declared *final*. Note that these constraints are marked as being necessary constraints: they need to hold in all locations where the `@Entity` annotation is used.

A fourth usage constraint (lines 26–29) is added to specify the relation between the `@Entity` and `@Table` annotations. First, the `@Table` annotation can only be placed on classes that are already annotated with `@Entity`. A class annotated with `@Table` but not with `@Entity`, will possibly lead to load-time errors. Therefore, this constraint is marked as being a `@Sufficient` constraint. All places annotated with `@Table` should also carry the `@Entity` annotation.

@Table: The `@Table` annotation configures the table to which a given entity will be persisted in relational database. As explained above, the correct use of the `@Table` annotation on a class requires the class to already carry the `@Entity` annotation. Restrictions on the relations between annotations, such as conditioning the use of an annotation to the presence of another one, are not uncommon in large annotation-based libraries such as the JPA. However, in the specification of the JPA, when certain conditions of the source entities are met, the framework behaves as if

```

1 @Target (ElementType.TYPE)
2 @TargetVariable("class")
3 public @interface Table {
4     @Sufficient
5     public static final String IMPLICIT_TABLE_IN_ENTITY =
6         "?class classDeclarationHasAnnotation: ?
7         named: {Entity}";
8     //...
9 }

```

Figure 8. @Table Smart Annotation

those entities were annotated. In section 9.1.1 of the JPA specification it is stated that “*If no Table annotation is specified for an entity class, the default values defined [for the Table annotation] apply.*” These *implicit* annotations hinder the comprehension of EJB applications, and thus complicate their maintenance. Using Smart Annotations, we can alleviate this problem by including a constraint in the definition of the `@Table` annotation interface, shown in figure 8 lines (4–8), that says that if a class is annotated with `@Entity`, the `@Table` annotation should also be applied. By doing this, all the places in which the `@Table` annotation is implicit are flagged by the Smart Annotation toolset.

@AttributeOverride: Finally, we consider the `@AttributeOverride` annotation. This annotation is used on entity classes that wish to override the Object-Relational mapping of an attribute defined on a super class by stating the name of the attribute to override, and the new column to which it will map. This information is expressed in the `@AttributeOverride`’s members `name` and `column` on the source code listing in figure 9. Notice that the `name` member of the `@AttributeOverride` annotation is linked to the name of a field defined on the entity’s super-class. Thus, if the referred super-class field is changed by means of a rename refactoring for example, the link is silently severed and the `@AttributeOverride` annotation becomes invalid. We capture this link in the necessary constraint embedded on the annotation’s interface in lines 5–10. The first line of the constraint binds the logic variable `?super` to the parent class of the annotated class `?class`. The second line retrieves all possible variables (fields) that are defined by this super class and binds them to the variable `?variable`. Finally, the last condition of the constraint checks whether the name of the variable `?variable` matches the value of the `name` argument of the annotation. In other words, the `OVERRIDES_SUPER_FIELD` constraint verifies whether the overridden properties’ name matches the name of a variable in the super class `?super`.

IV. DISCUSSION

Discussion: The two case studies above demonstrated the use of Smart Annotations. The goal of these two case

```

1 @Target (ElementType.TYPE)
2 @TargetVariable("class")
3 public @interface AttributeOverride {
4
5     @Necessary
6     public static final String OVERRIDES_SUPER_FIELD =
7         "?class extends: ?super," +
8         "?super definesVariable: ?variable," +
9         "?variable
10         variableDeclarationFragmentHasName:?name" ;
11
12     @TargetVariable("name")
13     String name();
14
15     Column column();
16 }

```

Figure 9. @AttributeOverride Smart Annotation

studies is to illustrate that Smart Annotations provide a useful means to translate the constraints that are often implied by the use of annotations — and that are only implicitly available — into a set of explicitly verifiable rules. By verifying these rules, Smart Annotations is able to pinpoint locations in the source code that are possibly either incorrectly annotated, or where a particular annotation is lacking. Although a full discussion of the merits of the use of a logic query language lies outside the scope of this paper, it is worth mentioning that this declarative means for expressing annotation constraints is vital to the usability of our approach. As we tried to show using the examples above, the logic query language offers an intuitive and expressive means to write down the constraints that apply to an annotation in a succinct way. Another key feature of our approach is the fact that we allow for declaring explicit exceptions to the annotations. While the above case studies did not illustrate an example of the use of these constructs, it is not hard to imagine situations where they can be useful. For example, in the second case study we required all classes annotated with the @Entity annotation to also carry the @Table annotation. While this is considered good practice, it is however not a formal requirement of the specification. If a developer wants to indicate for a particular class annotated with @Entity that the @Table annotation does not have to present, the developer can explicitly mark this exception using the @DoesNotApply construct of Smart Annotations.

Limitations: Our current implementation suffers from the following technical limitations. First, since we rely on the standard Java annotation framework, we are not able to support annotations at every level of detail in the source code. Currently, we can only annotate types, fields and methods. In order to express particular annotation constraints, this might be too coarse-grained. Especially since our logic query language offers support for reasoning at the level of individual statements and expressions, we plan to extend

Smart Annotations to also allow the annotation of individual Java statements.

Second, our existing tool support for Smart Annotations is very limited. While it allows us to declare constraints from within the body of an annotation declaration, these constraints are expressed as static String fields. This support for writing down the logic programming queries is fairly limited. We plan to extend it by providing dedicated tools on top of the current implementation that integrate the logic query language with Eclipse, offering debugging and syntax highlighting capabilities for the SOUL language.

Finally, our current implementation requires access to the actual source code of the annotation declarations. When applying our technique to libraries or frameworks that define annotations, this is not a reasonable assumption. One way of solving this problem would be to separate the definition of the Smart Annotations from the actual annotation declarations, for example by means of a dedicated tool.

V. RELATED WORK

There are three categories of approaches that are related to Smart Annotations, namely *Annotation Validators*, *Pluggable Type Systems* and *Code Checkers*. In what follows, we discuss each of these categories.

Annotation Validators: There exist a number of approaches that aim at validating the use of annotations. First, in [21] Cepa et al. propose a tool called ADC to validate dependencies between of custom attributes in the .NET platform. ADC offers meta-attributes to encode requires/excludes dependencies between attributes. In [22], Eichberg et al. propose to validate annotation constraints in Java programs by representing a program as an XML document, and representing the constraints as XPath queries. This allows the annotation developer to express constraints on the characteristics of the source-code entities that carry annotations. AVal [23], [24] offers a set of meta-annotations to express both dependencies between annotations and dependencies between annotations and the source code entities on which they lay. Additional constraints can be expressed using OCL queries over a meta-model of the source code.

With regard to the current state of the art in annotation validation, our approach offers a number of advantages: first, Smart Annotations is an extensible framework which allows annotation developers to include their own constraints using a *declarative* language. As we illustrated above, this use of a declarative language allows us to succinctly and expressively write down annotation constraints, in contrast to the use of imperative languages such as OCL and XPath. Second, in Smart Annotations two kinds of constraints are presented: those necessary for an annotation to be correct where it is used, and those that make a source code entity amenable to be annotated. So far, the approaches discussed above only support necessary constraints. The examples presented in section III reflect the need for sufficient annotations. Third,

in our approach, the application developer is allowed to state exceptions to the constraints expressed in the Smart Annotations. This is beneficial, since explicit documentation of this exceptions is kept embedded in the source code by means of `@DoesNotApply` and `@DoesApply` annotations.

Pluggable Type Systems: A second group of approaches that is related to Smart Annotations are pluggable type systems [25], [26]. These systems allow developer to add additional typing constraints to the source code of a system by annotating that source code. A prototypical example of the use of a pluggable type system is the annotation of particular statements in a system with a `@NotNull` annotation, indicating that the value of the annotation should never be null. Most of these approaches come with a predefined library of annotations that can be used to express common typing constraints in the source code. Furthermore, some pluggable type systems offer an interface to extend this set of predefined annotations.

Smart Annotations can be considered to be a sort of extensible pluggable type system. Using our approach, a developer can define annotations along with a set of constraints that should be satisfied by the annotated source code. However, pluggable type systems do not offer the possibility to express sufficient conditions (i.e. identify locations that should be annotated) and do not provide explicit support for exceptions to the annotation constraints.

Code Checkers: Smart Annotations is also related to code checking tools such as FindBugs [27], CheckStyle [28], PMD [29] and Lint [30]. These tools offer facilities to detect a wide variety of common mistakes and errors in source code. Next to an extensive catalogue of possible bugs, some of these approaches – such as PMD – offer facilities to extend the set of detectable errors. To this end, they allow developers to implement their own analysis by providing access to the underlying parse trees of the analyzed programs.

Smart Annotations allow developers to specify and check annotation-using code making Smart Annotations similar to these approaches. However, Smart Annotations attach checks to the definition of the elements they check. Because of this, constraints defined for Smart Annotations are closely related to the semantics of the concepts represented by the annotations, in contrast to the more general programming mistakes checked by traditional frameworks, which stem from characteristics of the programming language.

The constraints defined in Smart Annotations link the use of an annotation to the presence of naming conventions, structural patterns or other annotations in the application. In this regard, Smart Annotations is also related to some of our previous work, namely the IntensiVE tool suite [31]. IntensiVE provides a formalism and tool support to express, verify and visualize a wide range of so-called regularities: naming conventions, design patterns, implementation idioms, and so on. Similar to Smart Annotations, IntensiVE

leverages the Soul language to query the structure of programs, while providing dedicated support to deal with co-evolving annotations and source code.

VI. CONCLUSION

In this paper we have introduced Smart Annotations. Smart Annotations is an approach that enables developers to co-evolve annotations and source code. Our approach introduces the concept of imposing necessary and sufficient constraints on how the annotations should be related to the source code, our approach can suggest locations in the source code that are either incorrectly annotated, or where a particular annotations is missing. As a means to express these constraints, we employ the logic language Soul, that offers a declarative medium to write fine-grained queries about Java code. Furthermore, our approach recognizes the need to deal with exceptions to where the annotations should apply and offers explicit support for documenting such exceptions.

To illustrate the benefits of our approach, we applied it to examples from the domain of aspect-oriented programming and Enterprise Java Beans. In particular, we demonstrated how — using Smart Annotations — we can alleviate the fragile pointcut problem by having pointcuts rely on evolution-robust annotations. Second, we also demonstrated how we can make the implicit constraints that are imposed by the Java Persistence API explicit and verifiable with respect to the source code.

ACKNOWLEDGMENT

Andy Kellens is funded by a research mandate provided by the “Institute for the Promotion of Innovation through Science and Technology in Flanders” (IWT Vlaanderen). This research is supported by the IAP Programme of the Belgian State.

REFERENCES

- [1] C. Bauer and G. King, *Java Persistence with Hibernate*. Manning Publications, 2006, ISBN 978-1932394887.
- [2] “Spring Application Framework,” <http://www.springframework.org>.
- [3] T. Tourwé and T. Mens, “Identifying refactoring opportunities using logic meta programming,” in *7th European Conference on Software Maintenance and Reengineering, Benevento, Italy*, 2003.
- [4] C. Koppen and M. Stoerzer, “Pcdiff: Attacking the fragile pointcut problem,” in *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.
- [5] T. D’Hondt, K. De Volder, K. Mens, and R. Wuyts, “Co-evolution of object-oriented software design and implementation,” in *Software Architectures and Component Technology*, M. Aksit, Ed. Kluwer Academic Publisher, January 2001, pp. 207–224, proceedings of SACT 2000.

- [6] “The Smalltalk Open Unification Language (SOUL),” 2008. [Online]. Available: <http://soft.vub.ac.be/SOUL/>
- [7] K. Mens, I. Michiels, and R. Wuyts, “Supporting software development through declaratively codified programming patterns,” in *SEKE 2001 Proceedings*. Knowledge Systems Institute, 2001, pp. 236–243, international conference on Software Engineering and Knowledge Engineering, Buenos Aires, Argentina, June 13-15, 2001.
- [8] J. Brichau, C. De Roover, and K. Mens, “Open unification for program query languages,” in *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.
- [9] C. De Roover, “A logic meta programming foundation for example-driven pattern detection in object-oriented programs,” Ph.D. dissertation, Vrije Universiteit Brussel, August 2009.
- [10] C. De Roover, I. Michiels, K. Gybels, K. Gybels, and T. D’Hondt, “An approach to high-level behavioral program documentation allowing lightweight verification,” in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC06)*. IEEE Computer Society, 2006, pp. 202–211. [Online]. Available: <http://doi.ieeeecomputersociety.org/10.1109/ICPC.2006.10>
- [11] A. Kellens, K. D. Schutter, T. D’Hondt, L. Jorissen, and B. V. Passel, “Cognac: A framework for documenting and verifying the design of cobol systems,” in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR09)*, 2009, pp. 199–208.
- [12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, “Co-evolving code and design with intensional views: A case study,” *Elsevier Journal on Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.
- [13] “The Eclipse JDT Core Component Website,” <http://www.eclipse.org/jdt/core/index.php>, 2008.
- [14] A. van Deursen, M. Marin, and L. Moonen, “AJHotDraw: A showcase for refactoring to aspects,” in *Workshop on Linking Aspect Technology and Evolution*, 2005.
- [15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtoir, and J. Irwin, “Aspect-oriented programming,” in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS. Springer Verlag, 1997, pp. 220–242. [Online]. Available: citeseer.ist.psu.edu/kiczales97aspectoriented.html
- [16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [17] A. Kellens, K. Mens, J. Brichau, and K. Gybels, “Managing the evolution of aspect-oriented software with model-based pointcuts,” in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, no. 4067, 2006, pp. 501–525.
- [18] I. Nagy, L. M. J. Bergmans, W. K. Havinga, and M. A. sit, “Utilizing design information in aspect-oriented programming,” in *Proceedings of International Conference NetObjectDays, NODe2005, Erfurt, Germany*, ser. Lecture Notes in Informatics, R. Hirschfeld, Ed., vol. P-69. Bonn, Germany: Gesellschaft fuer Informatik (GI), September 2005, pp. 39–60.
- [19] G. Kiczales and M. Mezini, “Separation of concerns with procedures, annotations, advice and pointcuts,” in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS. Springer Verlag, 2005, pp. 195–213.
- [20] L. D. Michel and M. Keith, *Enterprise JavaBeans, Version 3.0*, Sun Microsystems, May 2006, Java Persistence API.
- [21] V. Cepa and M. Mezini, “Declaring and enforcing dependencies between.NET custom attributes,” in *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, ser. Lecture Notes in Computer Science, G. Karsai and E. Visser, Eds., vol. 3286. Springer, 2004, pp. 283–297.
- [22] M. Eichberg, T. Schäfer, and M. Mezini, “Using Annotations to Check Structural Properties of Classes,” in *Fundamental Approaches to Software Engineering, 8th International Conference*, ser. Lecture Notes in Computer Science, M. Cerioli, Ed., vol. 3442. Edinburgh, Scotland: Springer, 2005, pp. 237–252.
- [23] C. Noguera and L. Duchien, “Annotation framework validation using domain models,” in *Fourth European Conference on Model Driven Architecture Foundations and Applications*, Berlin, Germany, June 2008, pp. 48–62.
- [24] C. Noguera and R. Pawlak, “AVal: an extensible attribute-oriented programming validator for java,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. Volume 19 Issue 4, pp. 253 – 275, Jul. 2007.
- [25] N. Haldimann, M. Denker, and O. Nierstrasz, “Practical, pluggable types for a dynamic language,” *Computer Languages, Systems and Structures*, vol. 35, no. 1, pp. 48–64, 2009.
- [26] M. Papi, M. Ali, T. Luis Correa, J. Perkins, and M. Ernst, “Practical pluggable types for java,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 201–212.
- [27] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.
- [28] “Checkstyle,” December 2006. [Online]. Available: <http://checkstyle.sourceforge.net>
- [29] “PMD,” December 2006. [Online]. Available: <http://pmd.sourceforge.net>
- [30] S. Johnson, “Lint, a C program checker,” in *Unix Programmer’s Manual*, 7th ed., M. McIlroy and B. Kemighan, Eds. AT&T Bell Laboratories, 1979, vol. 2A.
- [31] K. Mens and A. Kellens, “IntensiVE, a toolsuite for documenting and checking structural source-code regularities,” in *Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 239–248.