Refactoring in the Presence of Annotations

Carlos Noguera, Andy Kellens, Coen De Roover, Viviane Jonckers
Software Languages Lab,
Vrije Universiteit Brussel, Belgium
Email: {cnoguera,akellens,cderoove,vejoncke}@vub.ac.be

***Abstract*—Current-day programming languages include constructs to embed meta-data in a program's source code in the form of annotations. More than mere documentation, these annotations are used in modern frameworks to map source-level entities to domain-specific ones. A common example being the Hibernate Object-Relational Mapping framework that relies on annotations to declare persistence configurations. While the presence of annotations extends the base semantics of the language, it also imposes restrictions on the annotated program. In this paper we consider the manner in which annotations affect automated refactorings, and in particular how they break the behavior preservation of these refactorings. As refactorings, during their condition checking phase, ignore the annotation's restrictions they can no longer guarantee the preservation of the domain-specific mappings. To address this problem, we propose to make the restrictions of the annotations explicit, and use them to steer the refactoring process. A prototype extension of the Eclipse IDE's refactoring engine is used to demonstrate our approach on three annotation libraries: Java Persistence API, AspectJ5 and Simple XML serialization.**

***Index Terms*—Refactoring, Annotations, Tool support.**

I. Introduction

Many current-day programming languages provide constructs that allow developers to augment their source code with meta data. Well-known examples of such constructs are pragmas in Smalltalk, attributes in C#, and annotations in Java. Various libraries and frameworks have adopted these meta data facilities as a means to embed configuration parameters in the source code. For example, Hibernate is a well-known Java framework, implementing the Java Persistence API (JPA), for persisting objects to a relational database. By means of a set of annotations, users of Hibernate can declare how source-code entities such as classes and fields are mapped onto database tables. Likewise, the JUnit testing framework employs annotations as a means to specify and configure test methods. As a final example, consider the AspectJ programming language, where developers can adopt an annotation-based style in which pointcuts and advices are configured by means of embedding meta data in the source code. This use of meta data has resulted in that annotations no longer solely serve as a form of documentation but become an integral part of the behavioral specification of the system.

Annotated programs, by virtue of the interpretation of these annotations, extend the base language semantics. A class sporting Hibernate's `@Table` annotation is behaviorally different from one sporting AspectJ's `@Aspect` annotation. These semantic changes vary depending on the set of annotations used in an application. This has an impact on the assumptions software engineering tools – and automated refactorings in particular – make on the source code. Automated refactorings [1] are an integral part of the tools of the trade of modern software developers. They are source code transformations that allow the developer to modify the application, to ease the introduction of a change or to improve its design quality, without affecting its semantics. When dealing with annotated programs, if the refactoring transformation is to remain behavior preserving, it must take into account the behavioral specification induced by the presence of the annotation.

This behavioral specification implies a number of assumptions on the placement and use of the annotations in the source code. First, annotation libraries potentially impose a requirement that the annotated source code needs to respect in order for the declared configuration to be *valid*. For example, the Java Persistence API (JPA) requires that all classes that are annotated with the `@Entity` annotation have a public or protected constructor without any arguments, and are not declared final. Second, these libraries make a number of assumptions regarding the *domain* and *mapping* that they express. As an example, consider a class that is mapped using the `@Table` annotation of JPA onto a database. The use of this annotation implies that there exists a table in the database named after the argument of the annotation, or if such an argument is not provided, the name of the class. However, both kinds of assumptions are not *explicitly* stated in the source code, existing only in documentation or specification files. As a result, there is little support for preventing these assumptions from being violated when altering or refactoring the system.

Within literature we can find numerous approaches [2]–[6] that aid developers in ensuring that, after the source code has been changed, the configuration expressed by means of the annotations is still *valid*. However, none of these approaches provides support for documenting the *domain* assumptions implied by the annotation libraries. As a consequence, seemingly safe modifications to the source code – such as the application of refactorings – can result in a valid, albeit *different mapping*, thereby introducing errors in the behavior of the system.

In this paper we tackle this problem by providing an approach that allows developers of annotation libraries to document, through annotation specifications, the domain dependencies that their annotations introduce. By integrating this specification with the refactoring engine of Eclipse, we are able to make Eclipse's refactorings annotation-aware:

978-1-4673-2312-3/12/$31.00 © 2012 IEEE

when a refactoring is applied to the annotated source code, our approach is able to warn developers about changes that potentially break the mapping expressed by the annotations, which would otherwise go unnoticed.

II. Problem Description

We illustrate the problem of refactoring annotated code through an example. This example consists of a simple Java system that uses the Java Persistence API (JPA) [7] to specify an object-relational mapping. Various implementations of this API exist, such as Hibernate[1] and OpenJPA[2]. The JPA defines a number of annotations that can be used to describe such a mapping. For example, the `@Table` annotation allows developers to define the name of the table to which the class will persist by an annotation element `name`. The JPA specification allows the developer to leave the `name` of the table unspecified in which case the table's name property will take as a default value the name of the entity. The `@Entity` annotation for specifying that a class is a persistent entity follows a similar scheme for defining the name of the entity: the `@Entity` annotation defines an element `name` which defaults to the name of the class it annotates.

```
@Entity(name = "Customer")
@Table(name = "CUST",  schema = "records")
class CustomerEntity{...}

@Entity
@Table
class Product{ ... }

@Entity
class Order{ ... }
```

Fig. 1. Different mapping specifications for entity classes defined in the JPA.

Figure 1 shows the code of three entity classes, `CustomerEntity`, `Product` and `Order`. The `CustomerEntity` class is mapped to a table called `CUST` as defined by the `name` property on its `@Table` annotation. The second class, `Product` uses the default values for both the name of the entity and table, and is thus mapped to a table called `Product`. Finally the `Order` class, although it does not carry explicitly a `@Table` annotation, will be mapped to a table called `Order` since the JPA specification states that "if no `@Table` annotation is specified for an `@Entity` class, the default values for the `@Table` annotation apply."

Now let us study what the impact is of refactoring the source code of this example. Applying a *rename* refactoring to change the name of the `CustomerEntity` class to just `Customer` will succeed and will not change the mapping of the entity class to the database: the values of the class will still be persisted and retrieved from the table called `CUST`. However, applying a similar *rename* refactoring to rename `Product` to `Item` will in fact affect the mapping to the database, since instances of the class will no longer persist to a table `Product` but will be persisted to table `Item`. Depending on the configuration of the underlying implementation of the JPA spec, the change of mapping might result in an error (no table named `Item` exists), or a change to the database schema (a new table `Item` is created and new instances persisted there). In either case, *the rename refactoring is no longer behavior preserving* in the presence of annotations. The problem is further compounded in the case of a rename refactoring applied to the `Order` entity class, since even the `@Table` annotation is implicit.

If refactorings are to remain behavior preserving in the presence of annotations, they must respect the (implicit) assumptions these annotations make over the code in which they are placed. Before providing a more in-depth analysis of this problem in Section IV and introducing our approach (Section V), we take a look at related work and position ourselves with respect to the state-of-the-art.

III. Related Work

a) Refactoring of multi-language applications: Nowadays, software systems are seldom implemented using a single programming language. Multi-language refactoring entails that, when refactoring artifacts that belong to one language, entities in another language used in the same system might also have to be altered. The problem of annotation-aware refactoring can be considered to be a specific instance of refactoring of multi-language applications, with regular Java as one language and the configuration expressed by means of annotations as another. Coupled transformations [8] appear to be suitable technique to implement such multi-language refactorings. Such transformations – when applied to an artifact expressed in one language – also transform dependent artifacts expressed in other languages. Note however that this technique does not guarantee that the transformed program is behavior preserving.

As a concrete example of multi-language refactoring, consider the work of Schink et al. [9]. Similar to the example presented in Section II, they have observed that refactoring Java source code with embedded JPA annotations that describe an object-relational mapping can result in that, after refactoring, this object-relational mapping is invalided. For the specific combination of Java, SQL and Hibernate their approach implements augmented versions of the "Push Down Method" and "Rename Method" refactorings that – under well-defined conditions – transform both the Java program and the dependent data in the database in order for both artifacts to remain consistent. The approach we present in this paper is more modest in its goals: we do not aim at transforming dependent artifacts, but rather to *warn* developers about non-behavior preserving refactorings. However, the approach we present in this paper is refactoring agnostic, and can be parametrized to support several annotation libraries.

b) Verifying implicit assumptions of annotation libraries: We are not the first to tackle the problems that arise when the implicit assumptions that govern the use of annotations

[1] http://www.hibernate.org
[2] http://openjpa.apache.org

are violated. There exists a number of approaches that aim at documenting and validating these implicit assumptions.

The *ADC* tool proposed by Cepa et al. [4] provides support for validating the dependencies between .NET custom attributes. By means of meta-attributes, the 'requires' and 'excludes' dependencies between these attributes can be documented; the tool provides support for validating these documented dependencies.

XIRC by Eichberg et al. [5] is an approach for validating annotation constraints in Java programs. This approach represents a program as an XML document; annotation constraints are expressed by means of XPath queries.

AVal [6] consists of a set of meta-annotations that allow for documenting both the dependencies between annotations, as well as how these annotations should be used within the annotated source code. Follow-up work [10] allows for documenting additional constraints by means of OCL queries over a meta-model of the source code.

Tilevich and Song [3] present meta-data invariants: an approach for documenting the naming and typing relationships between source code and annotations. To this end, their approach offers a declarative domain-specific language named *MIL*. This language offers annotation library developers a set of pattern-based constructs to characterize source-code entities and express the meta-data invariants governing these source-code entities. Furthermore, their approach provides support for automatically inferring such meta-data invariants from third-party applications [11].

In previous work we have introduced *Smart Annotations* [2]. *Smart Annotations* provide annotation library developers a means to augment meta-definitions with the constraints that govern the use of these annotations. These constraints are expressed using the logic program query language SOUL [12].

The above approaches share the common goal of aiding users of annotation libraries in verifying that the mapping declared by the annotations is *valid*. In other words, they warn developers about source code entities that are incorrectly annotated, or where particular annotations are missing. However these approaches do not allow detecting whether, when the source code is being refactored, that the mapping declared by the use of annotations is *preserved*. Song et al. [11] propose to prevent this problem by enforcing the naming conventions that encode the mapping defined by the annotations. This however severely limits the flexibility offered by the annotation libraries. The work presented in this paper complements the above approaches by focussing on preserving the *domain mapping* expressed by annotations when refactoring code.

IV. REFACTORING IN THE PRESENCE OF ANNOTATIONS

As illustrated in Section II, most refactoring tools are oblivious to the semantics that are introduced through the use of annotation libraries, and consider annotations to be part of the signature of a method, or the definition of a field or class. For example, when applying a "Move field" refactoring in the Eclipse IDE to an annotated field, the annotation is moved along with the field. Conversely, if a developer applies the "Extract method" refactoring on a piece of code belonging to an annotated method, the annotation is left in place (and not added to the newly extracted method). Such behavior is clearly not always desirable, as we have shown above. As each annotation library implies its own set of assumptions regarding the annotated code, there does not exist a one-size-fits-all solution for making refactorings annotation-aware. Instead, we propose an approach that makes the hidden assumptions made by annotation libraries explicit, and incorporates this information in the refactoring process.

Before introducing our approach, we take a brief look at what it means for a refactoring of annotated source code to be behavior preserving, and discuss the problem introduced by implicit configurations of annotation libraries.

A. Behavior Preservation

An important property of automatic refactorings is that they are behavior preserving: while refactorings change the *structure* of a program, they do not have an impact on the *observable behavior* of the program. As we have discussed earlier, the use of annotations as a means to configure frameworks potentially voids the behavior preservation guarantees of refactorings.

In order for a refactoring to take the semantics introduced by the use of annotations into account, it needs to preserve two properties of the annotated code: First, the refactoring must ensure that the mapping expressed by the annotations is still *valid* after the program has been transformed. In other words, the refactoring must take into account whether the transformed source code is still correctly annotated – according to the assumptions made by the annotation library. Second, refactorings must not have any impact on the actual *domain mapping* expressed by the annotations. The semantics associated with an annotation library place a number of *dependencies* on *resources external* to the source code. For example, in the case of JPA, the mapping presumes the existence of a particular set of database tables and columns onto which classes and fields are mapped. In order for the refactoring to be behavior preserving, it should depend on *the same* external resources – both before and after it has been applied. Applied to JPA, this means the refactoring should not have an impact on which database tables and columns the source code is mapped.

The problem of ensuring that the mapping is valid has been studied extensively in literature, both by ourselves as well as by others (see Section III). In the remainder of this paper we focus on the problem of maintaining the domain mapping introduced by the use of annotations. In particular, we present how, by translating the problem of behavior preservation into dependency preservation, and by providing support for declaring and validating these dependencies, we are able to render automated refactorings annotation-aware.

B. Implicit Configuration

Many annotation libraries make use of the principle of "configuration by omission". This entails that, in order to map a domain-specific concept onto the source code, the developer

is not required to explicitly provide the complete mapping. Rather, the library will implicitly assume the presence of other constituents of the mapping based on the annotations provided by the developer. We can distinguish between two kinds of such implicitly assumed configurations:

- **Implicit default values:** Parameters of a domain-mapping are often specified using the attributes of annotations. Consider for example the `@Table` annotation of JPA that has a member value called `name` corresponding to the database table onto which the annotated class is mapped. To ease configuration, annotation libraries often provide an implicit default value for the argument of an annotation, if no such value was provided for the developer. In many cases, this default value is directly linked to the source code entity that is annotated. For example, if no argument is passed to the `@Table` annotation, JPA will take the name of the `@Entity` annotation accompanying it as the name of the database table onto which the class is mapped.
- **Implicit annotations:** Furthermore, annotation libraries can introduce implicit annotations into the source code. Based on the annotations provided by the developer, the framework assumes the presence of other annotations, even if they have not been declared explicitly by the developer in the source code. Yet again, we consider JPA's `@Table` annotation as an example. If a developer annotates a class with this `@Table`, the library will implicitly assume that all non-transient fields of the class carry the `@Column` annotation, with as an implicit default value (indicating the column name) the name of the field.

While implicit configurations ease the use of annotation libraries, they increase the complexity of rendering automated refactorings annotation-aware. To circumvent this problem our approach – presented in the next Section – not only documents how the use of annotations creates a mapping between domain-specific concepts and source code, but also provides a means to explicitly express the implicit configurations assumed by the annotation library.

V. ANNOTATION-AWARE REFACTORING

In order for an IDE's refactoring engine to correctly handle annotated code, it must first be made aware of the semantics it must preserve. Our approach hinges on *annotation specification files* written by the annotation framework developer. An annotation specification denotes the dependencies on domain-specific concepts that are imposed by the set of annotations used by the framework. This specification is divided into two parts: the default values specification and the invariant dependencies. The former specifies the (implicit) default values of annotations defined by the framework, while the latter specifies which are the annotation values that represent dependencies on domain-concepts that should remain invariant upon refactoring. These invariant dependencies are the cornerstone of our approach, and are used by the refactoring engine to ensure behavior preservation.

Normally, refactoring engines assert whether a particular refactoring transformation will be behavior preserving by evaluating a number of preconditions. These preconditions depend both on the refactoring to apply and on its parameters (e.g., a rename refactoring is parametrized by the source code entity and its new name). If the preconditions are satisfied, then the transformation is performed, and otherwise an error is reported to the user.

While it would be possible to augment the refactoring preconditions with the annotation behavior specifications, we have decided instead to implement the annotation behavior preservation as *post-conditions*. The use of post-condition dependency checking in refactoring has already been proposed [13] to simplify the implementations of refactorings, although it has not been applied to the problem of annotated code. Since we express the problem of behavior preservation of annotated code as a dependency preservation problem, a post-condition that checks whether the dependencies were maintained by the refactoring transformation is more natural than trying to *a priori* predict whether the refactoring will affect said dependencies. When applying a refactoring, in addition to the refactoring's original pre-conditions, our tool determines the dependencies on domain concepts that are assumed by the annotations. Then, the refactoring is simulated and the new dependencies are determined and compared to those obtained in the previous step. Any discrepancy between the two sets of dependencies implies that the refactoring violated an invariant dependency, and therefore cannot be applied. This approach of post-condition checking has the benefit of being refactoring agnostic since we do not care what the refactoring actually does, but only that dependencies were maintained.

To express the annotation specification we use the SOUL [12] program query language. SOUL is a logic programming language, tailored towards reasoning over source code. Using this language, annotation framework developers specify default values through predicates, and invariants through logic queries over the source code of the application that is being refactored.

In the following subsections, we detail how default values and invariant dependencies are specified and used to restrict the refactoring of annotated code. We also present a prototype tool integrated into the Eclipse refactoring engine. Section VI demonstrates for three annotation frameworks how our approach can be used to make the refactoring of annotated code behavior preserving.

A. Defining Default Annotations and Values

The implicit default values specification takes two forms: alternative definitions for the predicates that bind source code entities to annotations (answering whether a code element carries an annotation), and alternative definitions for those that bind annotations to their member-value pairs (answering what is the value of a particular annotation). SOUL defines predicates to assert the presence of an annotation on a particular code element through the binary `XXXDeclarationHas:/2` predicate where `XXX` can

be either `class`, `field` or `method`. Thus, the query `if ?class classDeclarationHas: {Table}` will provide solutions binding the logic variable `?class` to each class that carries the annotation `@Table`. For instance, for the classes of Figure 1, `?class` would be bound to the classes `CustomerEntity` and `Product` *but not to* `Order`, as it does not carry the `@Table` annotation.

The underlying idea of how to encode default values in our approach is to augment SOUL predicates so that when applied to a target system, they find both annotated code and *implicitly annotated* code. To this end, the annotation framework developer (or someone with deep understanding of the annotation framework) writes alternative definitions for the predicates that identify annotations and annotation element values.

```
"Entity annotation implies Table"
?class classDeclarationHas: {Table} if
  ?class classDeclarationHasNoExplicit: {Table},
  ?class classDeclarationHas: {Entity}

"Default name of Entity is simple name of Class"
?class classDeclarationHas: {Entity} with: {name}
                             value: ?value if
  ?class classDeclarationHas: {Entity}
                             without: {name},
  ?class classDeclarationHasName: ?value
```

Fig. 2. Default definition for the presence of `@Table` and the default value for the name of an `@Entity`.

For example, a developer wishing to encode the fact that, in the JPA, the presence of the `@Entity` annotation on a class implies a `@Table` annotation with default values would provide an alternative definition for the `classDeclarationHas:/2` predicate like the one shown in Figure 2. When a query finding all classes that have the `@Table` annotation is evaluated, this predicate will first check that the class has no actual `@Table` annotation, and then check if the class carries a `@Entity` annotation.

In a similar manner, when encoding the rule that states that the default name of an `Entity` is the simple name of the class that carries it, an annotation framework developer can write an alternative definition for the `classDeclarationHas:with:value:/3` predicate. This predicate binds class declarations to their annotations and corresponding member-value pairs. In Figure 2 the second predicate states that when retrieving the name of an `@Entity` annotation, this name must be the name of the class, as bound by the `classDeclarationHasName:/3` predicate on the last line if the annotation has no name member declared in the source code. By including these two predicate definitions in SOUL, the query

```
if ?class classDeclarationHas: {Table}
                         with: {name} value: ?name
```

when applied to the code in Figure 1 will produce solutions for all three classes: binding `?class` to `CustomerEntity` and `Product` and `Order`, and `?name` to `CUST`, `Product` and `Order` respectively.

B. Defining Invariant Dependencies

Having expressed the default annotations through alternative predicates, the developer can now express the dependencies that must be respected by the refactoring. Dependencies are represented by queries that rely on the alternative predicates defined in the previous step. Each query identifies which bindings (representing domain concepts) should remain invariant by means of the `isInvariant/1` keyword. The refactoring engine then runs all registered invariant queries before the refactoring, and stores the bindings for variables that are marked as invariant. The invariant queries are then applied again to the simulated result of the refactoring, checking that the bindings for the invariant variables are the same. If this is not the case, an error is presented to the user. Such an error message communicates the bindings for the violated invariants both before and after the refactoring, as well as the message between quotes at the beginning of the query. To provide additional context to the error, the annotation framework developer can include in the invariant query the source code element on which the error occurred. These context variables are to be marked with a `isContext/1` keyword.

In our running example of the JPA `@Table` annotation, the mapping to a particular database table is directed by the `name` attribute of the annotation that corresponds to the name of a database table. For the transformation to be behavior preserving, the value of this attribute must remain invariant in the refactored code. Figure 3 shows the invariant query (called "Table mapping changed") that states that for classes (`?class`) that carry the `@Table` annotation, the name of the mapped table (`?name`) must be invariant. The annotated class `?class` is passed as the context of a potential violation. This query illustrates how it is possible to state that values of annotations must be preserved.

```
"Table mapping changed"
if ?class classDeclarationHas: {Table} with: {name}
             value: ?name,
    ?name isInvariant,
    ?class isContext

"Mapping between Tables and Columns Changed"
if ?class classDeclarationHas: {Table} with: {name}
            value: ?tname,
    ?class classDeclarationDeclaresField: ?field,
    ?field fieldDeclarationHas: {Column} with: {name}
            value: ?cname,
    tableColumn(?tname,?cname) isInvariant,
    ?field isContext
```

Fig. 3. Invariant specification for the name attribute in `@Table` annotations.

Note that our approach also allows expressing that *relationships between values* must be preserved. This is illustrated by the second query, named "Mapping between Tables and Columns changed", in Figure 3. This query specifies that the relationship between the mapping of a class to a table and the mapping of its fields to the table's columns must remain invariant after a refactoring. In other words, after the refactoring, the fields belonging to the class should be mapped to the same columns of the same table as before the

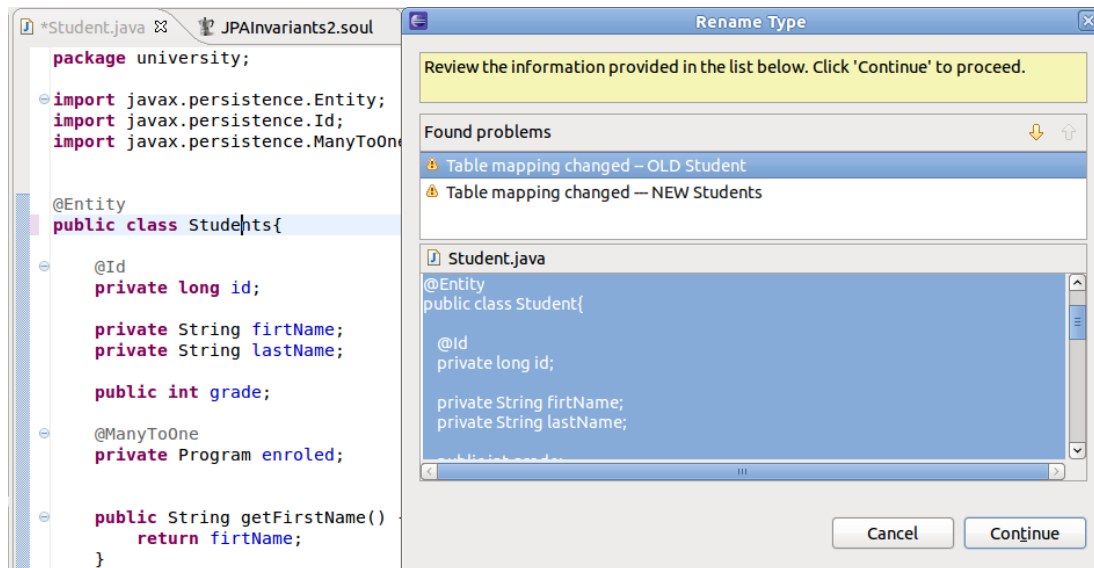

Fig. 4. Annotation refactoring support integrated with the Eclipse refactoring engine. Rename of entity class breaks implicit `@Table` annotation.

refactoring. The query reads as follows: for a class mapped to a table with name *?tname*, every field (*?field*) that is mapped to column named *?cname*, should maintain the relation `tableColumn(?tname,?cname)` as an invariant. If not, then the error must be reported with the *?field* source code element as its context.

C. Implementation

We have implemented our approach as an extension to the Eclipse refactoring engine by means of a refactoring participant plugin called AnnoRefactoring. The AnnoRefactoring plugin relies on our Barista [14] tool suite to perform the queries and assert the predicates that make up an annotation library's specification. Developers wishing to use an annotation library in their application must first load the corresponding annotation specification files into the AnnoRefactoring plugin. Once this is done, the Eclipse refactoring engine will invoke the AnnoRefactoring plugin to check whether the refactoring will break the behavior of annotations present in the source code. Figure 4 shows a screenshot of the message produced by the Eclipse refactoring engine. In this screenshot, we have tried to rename a class annotated with `@Entity`. As this refactoring would break one of the annotation's invariants – in this case the mapping to a table induced by an implicit `@Table` annotation, it is flagged as a warning.

It is important to note that in order to integrate our approach with the Eclipse refactoring engine, we had to extend the engine itself to allow for post-condition checking. By default, Eclipse's Language Toolkit (LTK) that provides the actual implementation of refactorings only allows refactoring participants to contribute preconditions to the refactoring process. We implemented the post-condition checking feature by leveraging the *preview changes* feature that the toolkit already provides to simulate the effect of a refactoring. Thus, the expected results of a refactoring are retrieved from this preview, and fed to the participants that contribute post-condition checks. In this manner, the AnnoRefactoring plugin has access to the source code of the application before and after the refactoring. The plugin then runs the invariant queries over the original source code, and over the results of the simulated refactoring. Bindings for variables marked as `isInvariant` in the queries are kept in two sets (one for the original code, one for the refactored one). The sets are then compared, and bindings only present in the original set are reported as "OLD" and bindings only present in the refactored one are reported as "NEW". This can be seen in the screenshot in Figure 4 where the original name of the table is marked as "OLD" where as the one resulting from the refactoring is marked as "NEW".

VI. Example Annotation Specifications

To illustrate our approach, we present annotation specifications for three annotation libraries: the `@Entity`, `@Table` and `@Column` annotations defined in the Java Persistence API, the `@Pointcut` annotation defined in AspectJ 5 and the `@Element` annotation defined in the Simple XML mapping library. Our motivation for choosing these libraries and particular annotations is two-fold. First, these annotations induce dependencies on external resources and allow us to demonstrate our handling of complex default values. Second, as the annotations are extracted from industrial, well-known frameworks, we believe that they are representative of the kind of situations that developers encounter on a day-to-day basis.

A. Entity, Table and Column from JEE

We first consider three annotations defined in the Java Persistence API (JPA) [7]: `@Entity`, `@Table` and `@Column`. Together, these three annotations form the backbone of persistence configuration in JEE. An entity is a lightweight persistent domain object in the business application, normally defined by placing an `@Entity` annotation on a class. Each entity

Implicit annotations and values	
Default name for entity is name of class	?class classDeclarationHas: {Entity} with: {name} value: ?value if ?class classDeclarationHas: {Entity}, ?class classDeclarationHas: {Entity} without: {name}, ?class classDeclarationHasName: ?value
Default name for Table is name of Entity	?class classDeclarationHas: {Table} with: {name} value: ?value if ?class classDeclarationHas: {Table}, ?class classDeclarationHas: {Table} without: {name}, ?class classDeclarationHas: {Entity} with: {name} value: ?value
Default name of Column is Field	?field fieldDeclarationHas: {Column} with: {name} value: ?value if ?field fieldDeclarationHas: {Column}, ?field fieldDeclarationHas: {Column} without: {name}, ?field fieldDeclarationHasName: ?value
Entity implies Table	?class classDeclarationHas: {Table} if ?class classDeclarationHasNoExplicit: {Table}, ?class classDeclarationHas: {Entity}
Table implies Column	?field fieldDeclarationHas: {Column} if ?field fieldDeclarationHasNoExplicit: {Column}, ?class classDeclarationDeclaresField: ?field, ?class classDeclarationHas: {Table}
Invariant dependencies	
Entity mapping	if ?c classDeclarationHas: {Entity} with: {name} value: ?name, ?name isInvariant, ?c isContext
Table mapping	if ?c classDeclarationHas: {Table} with: {name} value: ?value, ?value isInvariant, ?c isContext
Column mapping	if ?field fieldDeclarationHas: {Column} with: {name} value: ?value, ?value isInvariant, ?field isContext
Table to column relation	if ?class classDeclarationHas: {Table} with: {name} value: ?tname, ?class classDeclarationDeclaresField: ?field, ?field fieldDeclarationHas: {Column} with: {name} value: ?cname, tableColumn(?tname,?cname) isInvariant, ?field isContext

TABLE I
ANNOTATION SPECIFICATION FOR THE REFACTORING OF THE @ENTITY, @TABLE AND @COLUMN ANNOTATIONS DEFINED IN THE JPA.

class is persisted as directed by a @Table annotation, which specifies the primary table for the annotated entity through the annotation element name. In a similar manner, fields of entity classes are mapped to columns of the primary table through a @Column annotation that specifies as its name element the column name for the annotated field. In addition to field-based mapping, the JPA specification supports *property-based* mapping, in which case the @Column annotations are placed on the property's accessor (getter method). For this example we consider field-based mapping only. However, the annotation specification for the property-based style would be very similar. In order to make the use of JPA annotations on an application convenient, the JPA specification follows a configuration by omission scheme in which default values for annotation elements take care of the most common usages. In the case of the @Entity and @Column annotations' name element, the default value is the name of the source code entity on which they are placed (unqualified name of the class, and name of the column respectively). For the @Table name, the default value is the name of the associated @Entity. Similarly, an implicit @Table annotation is defined on classes sporting the @Entity and @Column annotations on the classes carrying a (possibly implicit) @Table.

Table I shows the annotation specification for the three JPA annotations: the first five rows show the default annotations and values, whereas the last four show the invariants. The first three rows cover the cases for implicit name attributes of @Entity, @Table and @Column annotations by providing alternative definitions for the classDeclarationHas:with:value/3 predicate. The fourth and fifth rows express the implicit annotations induced by the presence of @Entity and @Table on a class. The "Entity implies Table" predicate states that a class declaration has an implicit @Table annotation if the class declaration does not have the annotation explicitly, and it has an @Entity annotation. The "Table implies Column" predicate states that a field has a @Column if the field does not have an explicit @Column and the class that declares the field is annotated with @Table. Four invariants are defined in Table I: "Entity mapping" checks that the name of an entity is not affected by a refactoring, as the name of an entity is used in queries; "Table mapping" and "Column mapping" check that the mapping of the entity class to the database table and columns remains invariant; and finally the "Table to column" relation checks that fields mapped to a column are consistent with the class to table mappings.

By loading this annotation specification in our prototype tool, refactorings that affect the names of classes or fields, or

Invariant dependencies	

Pointcut name changed	`if ?m methodDeclarationHas: {Pointcut},`
	`?m methodDeclarationHasName: ?name,`
	`?name isInvariant, ?m isContext`

TABLE II
ANNOTATION SPECIFICATION FOR THE REFACTORING OF THE @POINTCUT ANNOTATION DEFINED IN ASPECTJ5

move fields to other classes will be prevented of changing the mapping of entity classes to a database.

B. Pointcuts and Advice of AspectJ5

In version 5, the aspect-oriented language AspectJ [15] introduced a new style to define aspects based on annotations. In this annotation-driven style[3], aspects are defined as Java classes carrying annotations that map Java source code elements to AspectJ ones. An AspectJ aspect is mapped to a class carrying the `@Aspect` annotation, and pointcuts and advice are mapped to methods annotated with `@Pointcut` and either `@Before`, `@After` or `@Around`. The AspectJ weaver then loads the aspect classes and instruments the application's code according to its annotations, so that the correct advice is executed either before, after or around the methods specified in the pointcuts. This means that there is a dependency between the advice and the pointcuts. Pointcuts take their name from the name of the method on which the `@Pointcut` annotation is placed. This name is then used to bind an advice to one or more pointcuts. While this dependency is similar to the one between a `@Column`'s name and the name of the field it is placed, AspectJ5's `@Pointcut` annotation does not define a `name` element. This implies that the name of a `@Pointcut` is *always* implicit, and changing the name of methods carrying the `@Pointcut` annotation will surely render any advice that refers to them faulty. Table II depicts the annotation specification for the `@Pointcut` annotation in AspectJ. This specification is composed of a single invariant dependency, stating that the name of methods carrying the `@Pointcut` annotation must remain invariant.

C. Simple XML serialization

As a final example, we present the annotation specification for the `@Element` annotation defined in the Simple XML serialization and configuration framework[4]. Simple allows developers to serialize Java objects to XML by relying on annotations to specify the schema of the resulting XML file, thus doing away with the need for a mapping configuration required by other XML serialization libraries for Java.

Simple defines annotations to express that a class will serve as the root of the XML document (`@Root`) and annotations to specify how the fields and properties of the class map to either attributes (`@Attribute`) or elements `@Element` of the XML document. Like the JPA, Simple allows configuration by omission: the developer is not required to explicitly annotate each field or property of the class with `@Element`. However, in contrast to the JPA, Simple requires the developer to explicitly declare that default values should apply for a class by means of the `@Default` annotation. This annotation takes a parameter stating whether the `@Element` annotations should be implicitly applied to fields (`DefaultType.FIELD`) or to properties (`DefaultType.PROPERTY`).

Each field or method annotated with `@Element` will be mapped to a corresponding XML element with the name defined by the annotation's `name`. If `name` is not specified, the `@Element` takes the name from the annotated element.

Table III shows the annotation specification for the `@Element` annotation. It defines four implicit annotations and values, two stating that a `@Default` annotation induces implicit `@Element` annotations on fields or properties (two first rows). In the case of default elements for fields, the specification of the first row states that a field carries an `@Element` annotation if the field is not annotated as `@Transient` and the class that defines the field is itself annotated with a `@Default` annotation that has `DefaultType.FIELD` as parameter. The specification for default elements properties is similar, except that we first check that the method on which the implicit `@Element` is placed complies with the accessor naming convention by verifying that its name matches the regular expression `{get.*}`.

The following two rows of the table specify the implicit `name` of the `@Element` annotations through an alternative definition for the `XXXDeclarationHas:with:value:/3` predicate. In the case of `@Element` annotations, a special predicate `isPropertyForName:/2` is used to extract the property's name from the accessor method by stripping the prefix "get" from the method's name and converting the remaining string to lowercase. Finally, in the two last rows of Table III, two invariants to protect the mapping of fields/properties to elements are expressed. The invariant queries have a similar form stating that the name of `@Element` must remain invariant, be it (implicitly) placed on a field or method.

VII. DISCUSSION AND FUTURE WORK

In this section we discuss the advantages and limitations of our approach, along with how we propose to overcome these limitations in future work.

Refactoring-agnostic approach.: Our approach offers the advantage that it is agnostic to the specifics and implementation of the refactorings that can be applied to the annotated system. As we treat behavior preservation of annotated code as a problem of preserving dependencies on domain concepts and that the conditions to validate a refactoring are checked

[3]http://www.eclipse.org/aspectj/doc/released/adk15notebook
[4]http://simple.sourceforge.net/

Implicit annotations and values	
@Default(FIELD) implies Element	?field fieldDeclarationHas: Element if not(?field fieldDeclarationHas:{Transient}), ?class classDeclarationDeclaresField: ?field, ?class classDeclarationHas:{Default} with: {value} value: {DefaultType.FIELD}
@Default(PROPERTY) implies Element	?method methodDeclarationHas: {Element} if ?method methodDeclarationHasName: {get.*} not(?method methodDeclarationHas:{Transient}), ?class typeDeclarationDeclares: ?method, ?class classDeclarationHas:{Default} with: {value} value: {DefaultType.PROPERTY}
Default element name is name of field	?field fieldDeclarationHas: Element with: name value: ?val if ?field fieldDeclarationHas: Element, ?field fieldDeclarationHas: Element without: name, ?field fieldDeclarationHasName: ?val
Default name of Element is name of property	?method methodDeclarationHas: Element with: name value: ?val if ?method methodDeclarationHas: Element, ?method methodDeclarationHas: Element without: name, ?method isPropertyForName: ?val
Invariant dependencies	
Element mapping changed	if ?field fieldDeclarationHas:{Element} with:{name} value: ?name, ?name isInvariant, ?field isContext
Element mapping changed	if ?meth methodDeclarationHas:{Element} with: {name} value: ?name, ?name isInvariant, ?meth isContext

TABLE III
ANNOTATION SPECIFICATION FOR THE REFACTORING OF THE @ELEMENT ANNOTATION DEFINED IN SIMPLE.

as post-conditions, we do not require annotation library developers to specify the interaction between each refactoring and the assumptions made by the annotations. As a result, our approach is rather light-weight: in order to make a wide variety of automated refactorings annotation-aware, developers only need to document the dependencies of their annotation libraries on domain-specific concepts.

No need to model the semantics of the annotation library.: Note that our approach does not require a full-fledged, explicit model of the semantics of an annotation library in order to detect refactorings that would not be behavior preserving. For example, in order to support JPA, we do not need to model the database structure onto which an application is mapped, nor how this mapping is realized by a concrete implementation of JPA. Instead, our annotation specification consists only of the implicit annotations that are assumed by the annotation library, and a declaration of which tables and columns in the database the source code is mapped onto. As demonstrated in Section VI, providing such an annotation specification for a library requires only a minimal effort.

Support for implicit annotations and implicit annotation members.: Our approach provides support for the implicit annotations and annotation members that are often assumed by annotation libraries. As a result, we do limit our approach to annotated programs that provide an explicit configuration, but also support “configuration by omission”, which is a common feature of annotation libraries. Such support is achieved by documenting both the domain dependencies and the implicit configuration of a set of annotations. To this end, we offer a single specification language that is based on the SOUL logic program query language.

Broader applicability.: While this paper focuses on refactoring in the presence of annotations, our approach is not limited to this problem domain. Our approach can be useful to prevent refactorings from altering the observable behavior when using other technologies that introduce a dependency on external entities. For example, many popular JEE frameworks such as *Struts*, *Spring*, and so on rely on the use of XML descriptors in order to configure the framework. This introduces dependencies between entities in the source code on the one hand and references to these entities within the XML documents on the other. As another example, consider frameworks such as JUnit that rely on naming conventions for configuration. Within JUnit, all methods on a test class prefixed with ‘test’ will be considered unit tests. It is clear that applying a simple rename refactoring to such methods can result in that they are no longer detected as test methods. In other words, there exists an implicit dependency between the test methods and the naming convention that identifies these methods. In future work, we plan to tailor our approach to express the dependencies between the source code and any external concepts as described above in order to prevent refactorings from altering the implied semantics.

Only support for domain dependencies.: The approach presented in this paper only focusses on one facet of the problem of refactoring annotated code, namely dependency preservation. While we feel that many issues present in the refactoring of annotated code result from the dependencies that these annotations impose on domain-specific concepts, this is not necessarily true for all available annotation libraries. As we already discussed above, refactoring of annotated code can potentially break the semantics expressed by the annotation

by no longer resulting in a piece of code that – according to the assumptions imposed by the annotation library – is still correctly annotated. These kinds of violations are however not detected by the approach presented here.

Our previous work on *Smart Annotations* [2] complements the work presented in this paper in that it enables developers to document and verify the *structural* assumptions implied by annotation libraries. By means of SOUL queries, developers can describe the structural constraints that dictate *where* in the source code annotations should apply, and which structural properties annotated source code entities should exhibit. In future work, we will integrate these Smart Annotations with our AnnoRefactoring plugin, such that our tool can warn developers about both structural and dependency violations. However, such an integration might not suffice to identify the violation of any kind of semantics introduced by annotations. While we are not aware of annotation libraries that introduce assumptions other than structural requirements and dependencies on domain-specific concepts, further study of annotation libraries is required.

Correctness of annotation specifications.: Our approach reports all dependency violations that could result from a refactoring by leveraging annotation specifications. These annotation specifications, that express the implicit annotations, implicit member values and dependency invariants of an annotation library, are expected to be provided by developers of the annotation library, or by expert users of such a library. Therefore, we cannot make any guarantees that this annotation specification is complete and correct, and that our tool will therefore warn about the correct set of dependency violations.

Restriction of refactorings.: Our AnnoRefactoring plugin makes refactorings annotation-aware by warning developers about transformations that would result in changes to the external behavior of a system. We achieve this by imposing additional (post-)conditions for the refactoring to be applicable. While this effectively aids in preventing refactorings from resulting in erratic behavior, it restricts the situations in which the refactoring can be applied. In future work we plan to overcome this limitation by associating each violation of a domain dependency with a compensating transformation of the annotation. For example, consider a class annotated with the JPA `@Table` annotation. If the annotation does not provide the name of the mapped database table as an explicit value, renaming this class would result in our approach warning the developer that the refactoring breaks the semantics induced by the annotation. By providing a compensating transformation, we could add the explicit value – corresponding to the name of the table – to the annotation.

VIII. SUMMARY

In this paper we have presented the problem of refactoring in the presence of annotations. Many libraries and frameworks use annotations to embed configuration parameters in source code. By means of an example taken from JPA, we have illustrated that the semantics that this use of annotations introduces can result in that refactorings are no longer behavior preserving. As one of the underlying causes of this problem, we have identified that such annotations introduce dependencies on external domain-specific concepts. By documenting these dependencies using SOUL, the approach presented in this paper allows developers to be warned about violations of domain dependencies introduced by the use of annotations when attempting to apply a refactoring. To support our approach, we have provided a proof-of-concept implementation in the form of the AnnoRefactoring Eclipse plugin. We have illustrated the applicability of our approach using three annotation libraries, namely JPA, AspectJ5 and Simple XML.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt, "Co-evolving annotations and source code through smart annotations," in *Proceedings of the 14th European Conference on Software Maintenance and Re-engineering (CSMR10)*, 2010.

[3] E. Tilevich and M. Song, "Reusable enterprise metadata with pattern-based structural expressions," in *International Conference on Aspect-Oriented Software Development (AOSD)*, 2010, pp. 25–36.

[4] 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.

[5] 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.

[6] 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.

[7] L. D. Michel and M. Keith, *Enterprise JavaBeans, Version 3.0*, Sun Microsystems, May 2006, jSR-220.

[8] A. Cunha and J. Visser, "Strongly typed rewriting for coupled software transformation," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 1, pp. 17–34, 2007.

[9] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel, "Hurdles in multi-language refactoring of hibernate applications," in *International Conference on Software And Data Technologies (ICSOFT)*, 2011, pp. 129–134.

[10] C. Noguera and L. Duchien, "Annotation framework validation using domain models," in *ECMDA-FA*, ser. Lecture Notes in Computer Science, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 48–62.

[11] M. Song and E. Tilevich, "Metadata invariants: Checking and inferring metadata coding conventions," in *International Conference on Software Engineering (ICSE)*, 2012.

[12] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The soul tool suite for querying programs in symbiosis with eclipse," in *International Conference on the Principles and Practices of Programming in Java (PPPJ)*, 2011, pp. 71–80.

[13] M. Schaefer and O. de Moor, "Specifying and implementing refactorings," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 286–301. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869485

[14] C. Noguera, C. De Roover, A. Kellens, and V. Jonckers, "Program querying with a SOUL: The barista tool suite," in *International Conference on Software Maintenance (ICSM)*, 2011, tool demo.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS. Springer Verlag, 2001, pp. 327–355.