Daileon: A Tool for Enabling Domain Annotations

José Roberto C. Perillo
jrcperillo@yahoo.com.br

Eduardo M. Guerra
guerraem@gmail.com

Clovis T. Fernandes
clovistf@uol.com.br

Aeronautical Institute of Technology
Praça Marechal Eduardo Gomes, 50
Vila das Acácias - CEP 12228-900
São José dos Campos - SP, Brazil

ABSTRACT

Software developers currently understand that the real complexity in most applications lies in the problem domain the software is dealing with. The approach known as domain-driven design follows the premises that, for most software projects, the primary focus should be on the domain and domain logic, and that complex domain designs should be based on a model. For this type of project, the concept of domain annotations, which is a way to define annotations specifically for domain objects, can be applied. Currently, most part of metadata-based components still does not use this concept. This paper proposes two techniques for enabling the use of domain annotations in new frameworks and existing ones, and introduces the Daileon framework, which enables such development in Java-based applications.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods.

General Terms

Algorithms, Design, Experimentation.

Keywords

Metadata, Annotations, Domain-Driven Design, Frameworks, Java.

1. INTRODUCTION

Over the two past decades, the software development area evolved to a point where people have come to understand that the real complexity, in most cases, is in understanding the business domain itself. The approach known as domain-driven design [4] follows the premises that, for most software projects, the primary focus should be on the domain and domain logic, and that complex domain designs should be based on a model.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RAM-SE '09, July 7, 2009 Genova, Italy.
Copyright 2009 ACM 978-1-60558-548-2/09/07... $10.00.

In the development field, a way to provide data about data, called metadata, is heavily used. One of the ways to define explicit metadata is the technique called annotations, which is available for the Java language and is a powerful mechanism to provide metainformation about class members, or even about classes themselves.

Some indirect problems may occur when using annotations. One of the problems is that sometimes a particular class or class member requires several annotations, and this can cause pollution of code and decrease its readability and maintainability. Another common problem is that annotations related to infrastructure are frequently placed in classes related to the domain. For instance, when creating business components using EJB 3 [3] beans, methods that should relate exclusively to the business domain end up being annotated with annotations that are related to infrastructure. Another problem is that creating code that directly uses annotations of external frameworks inevitably leads to tight coupling between the code being created and these frameworks. For infrastructure components that work with crosscutting concerns, it also breaks their obliviousness.

When following the domain-driven design approach, metadata that is added or that belongs to the domain model can be expressed as annotations on domain objects. A practical way to describe these annotations is domain annotations. The first work that proposed this term was [2], where the author exemplifies it with a case study in .NET. The goal of this paper is to propose two techniques for enabling domain annotations in Java-based applications. A framework called Daileon [8] is currently being developed to support such approach.

This paper is structured as follows. Section 2 addresses the domain modeling activity. Section 3 addresses metadata and annotations. Section 4 addresses the domain annotations concept, where the Daileon framework is also introduced. It is shown in its subsections how new frameworks can offer support to domain annotations and also how to use this concept with existing frameworks. The conclusion of this paper is shown as a final consideration in Section 5, where future works are also identified.

2. DOMAIN MODELING

Following the domain-driven design principles, domain modeling is the most important activity. Essentially, model is an abstraction that describes selected aspects of a domain that are relevant to solving problems related to that domain, and domain is the subject area to which the user applies a program. Domain model can be defined as a rigorously and

selective abstraction of the knowledge that the domain experts have of a particular domain. Commonly, it is represented by UML's class diagram, but it can practically be represented by anything able to represent the domain model, since it is not a particular diagram, but the abstraction behind it. Thus, the model can be represented by a diagram, by a graphic or even by text. Languages of a higher level of abstraction that are able to hide the complexity of general-purpose programming languages and that are created specifically to solve problems of a domain model are known as domain-specific languages [4].

In the domain-driven design reality, the critical complexity is in understanding the domain itself, which makes the model central to both analysis and implementation of a solution. Domain modeling is about selecting the relevant aspects of a domain and representing them, allowing software to enter the domain. The intimate link between model and implementation allows verifying if the analysis that went into the model applies to the software created and eases its continuous maintenance and development. It also allows developers to talk about the software in the domain language, promoting the ubiquitous language. Ultimately, if the model is not reflected in the code, it is irrelevant.

In terms of design, the "domain layer" is the manifestation of the domain model. This layer should concentrate all business logic, and should be isolated from other layers, such as infrastructure or application layers. The code that implements the domain model is where domain annotations should be applied.

3. METADATA AND ANNOTATIONS

Metadata is essentially a way to provide data about data. It is structured information that describes the characteristics of a particular resource. In other words, it consists of a number of pre-defined elements that represent the specific attributes of a particular resource. A classical example of metadata is a database schema.

Metadata can be represented in several different ways. For instance, names of class elements and their types are intrinsic metadata. Declarative metadata is frequently stored in XML files. Naming conventions are also a type of metadata (i.e. the "getter" and "setter" methods of the JavaBeans [10] convention). The annotation type, specifically for the Java language, is a special case of metadata. It enables attribute-oriented programming, which is a program-level marking technique that allows developers to annotate programming elements (i.e. classes and methods) to indicate application-specific or domain-specific semantics [9]. This technique was first introduced by the XDoclet [11] engine, which allows the developer to add more significance to the code, by adding metadata that are actually special JavaDoc tags. The concept of attribute-oriented programming was then introduced in the JSR 175 [1], and annotations became part of the Java language.

Annotations do not directly affect program semantics, they just add to the code complementary information that are used by tools, libraries or frameworks, which can in thesis affect the semantics of the program. It uses a special keyword, `@interface`, to be defined. It acts more like an interface rather than a class; hence it cannot contain code, which means that they need other code to consume them. For the domain annotations scenario, annotations are naturally the metadata type to be added to domain objects.

Some metadata-based components use annotations in domain classes to customize their logic properly. EJB 3, for instance, is a framework that uses annotations to configure variabilities in crosscutting concerns. The use of annotations by aspect-oriented frameworks [6] in some cases can reduce significantly the amount of advices and the syntactic coupling [7]. One point to be observed when using annotations in aspect-oriented frameworks is that it breaks part of their obliviousness [5], since classes will contain explicit information about crosscutting concerns. The use of domain annotations in this scenario restores the aspect obliviousness and also reduces the semantic coupling [7].

4. DOMAIN ANNOTATIONS

Creating domain annotations is a way to utilize domain-driven design with annotations that are created for a specific domain model implementation. With it, annotations placed on domain objects clearly have a relationship with the domain itself, not with matters out of it, helping the domain layer to be isolated from other layers. The dependency on annotations of external frameworks that has to exist so the application can be created is also encapsulated, which allows changing them by other annotations without effectively changing the code that implements the domain model. Another benefit is that having fewer annotations in the code makes it more readable and eases its maintenance.

Essentially, domain annotations either represent metadata that belongs to the domain or represent and replace annotations that do not belong to the domain model implementation, helping the domain layer to be isolated from other layers. For the latter case, a way to identify and interpret these annotations is required. Therefore, this study proposes a framework called Daileon (Domain Annotations Identifier and LEgacy annOtations arraNger), that is being developed to leverage such concept in Java-based applications. Until the moment this paper was published, this framework was still under construction. Daileon will provide mechanisms for new frameworks to identify and interpret domain annotations, and will also provide mechanisms for applications that use existing frameworks that do not support this concept, replacing domain annotations by their corresponding annotations via bytecode manipulation. The following subsections present these two approaches.

4.1 Domain Annotations for New Frameworks

One of the possible ways to create domain annotations is when the frameworks used support their creation. In this case, the annotations provided by these frameworks are replaced by domain annotations on domain objects, and these frameworks are able to recognize the domain annotations and identify which known annotations they represent.

In the context of the present work, the difference between new frameworks and existing ones is that new frameworks can be created allowing their users to create and use domain annotations instead of their own annotations. Figure 1 shows the technique that this paper proposes to be applied in such cases.

The example shown in Figure 1 shows a reduced code that illustrates how domain annotations can be created to be used with frameworks that support their creation. In this example, the `@Administrative` domain annotation is annotated with two annotations of a framework that supports the creation of domain annotations, plus

```
/* The domain annotation */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@DomainAnnotation
@FrameworkAnnotation1
@FrameworkAnnotation2
public @interface Administrative {
    // This annotation does not actually have attributes.
    // The code that consumes this annotation only uses the
    // annotations defined in it.
}

/* A method of a domain class, annotated with the domain annotation */
@Administrative
public void placeCurrentOffer() {
    // A business method of a domain class.
}
```

Figure 1: Domain annotations for new frameworks.

`@DomainAnnotation`, provided by the Daileon framework. To support domain annotations, frameworks must allow their annotations to annotate other annotations, and also have to be able to distinguish which known annotations a domain annotation corresponds to. For the latter case, Daileon provides several ways to identify a domain annotation and recognize which annotations it represents. The `@Administrative` domain annotation, shown in Figure 1, is also annotated with `@DomainAnnotation` so Daileon can identify that this is a domain annotation and inform which annotations it represents. Figure 2 shows an example of how it can be done.

```
// A reference to a domain class
Class<?> domainClass = DomainClass.class;
Method offerMethod = domainClass.getMethod("placeCurrentOffer");

if (DomainAnnotationsHelper.isAnnotationPresent(offerMethod,
        FrameworkAnnotation1.class)) {
    System.out.println("@FrameworkAnnotation1 found in offerMethod.");
    // Continue processing...
}
```

Figure 2: A framework that supports domain annotations using Daileon.

In the example addressed in Figure 2, the framework that supports the creation of domain annotations verifies if a known annotation is present in a method of a domain class, and for such, it uses a class called `DomainAnnotationsHelper`, which is provided by Daileon and has several static methods to identify and translate domain annotations. It is also possible to annotate a domain annotation with other domain annotations. In this case, the domain annotations of lower level would be represented by the domain annotation of highest level.

4.2 Domain Annotations for Existing Frameworks

The second technique proposed by this paper allows creating domain annotations in such a way that they replace annotations of existing frameworks in the domain model implementation. But since most existing frameworks do not support this concept in any way, the domain annotations have to be replaced by their corresponding annotations before the application is run, since these frameworks do not expect to find annotations other than their own in the code.

Figure 3 shows the technique that this paper proposes to be used with existing frameworks that do not support domain annotations. A method of a domain class is annotated with a domain annotation, which just indicates where to find

```
/* The domain annotation */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@DomainAnnotation
@MethodTemplate(annotatedClass = "br.ita.annotations.Annotations",
                method = "administrativeMethod")
public @interface Administrative {
    // The @MethodTemplate annotation indicates where the annotations
    // of external components are located, so they can be achieved and
    // replace the domain annotations via bytecode manipulation.
}

/*
 * The class that defines the external annotations to replace
 * the domain annotation
 */
public class Annotations {
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    @RolesAllowed(value = {"Administrator"})
    public void administrativeMethod() {
        // This method is used just to indicate which annotations
        // of existing frameworks have to be used.
    }
}

/* A method of a domain class, annotated with the domain annotation */
@Administrative
public void placeCurrentOffer() {
    // A business method of a domain class.
}
```

Figure 3: Domain annotations for existing frameworks.

its corresponding annotations (in the example shown in Figure 3, the domain annotation corresponds to annotations of the EJB 3 API). This is necessary because, in most cases, it is not possible to annotate an annotation with annotations of existing frameworks, since only methods or class elements can be annotated with them. Consequently, it is necessary to annotate the domain annotations with annotations that simply indicate a template class that contains the annotations that are represented by the domain annotation.

Even though creating domain annotations to be used along with frameworks that do not support their creation means replacing their annotations by other annotations that represent them, these annotations do not have any effect in their environment. For instance, annotating a domain class with a domain annotation that represents annotations of the EJB 3 API does not have any effect in the JEE container. Therefore, a mechanism capable of translating domain annotations to their corresponding annotations before running the application is necessary to enable the concept. One of the main functionalities Daileon provides is the ability to manipulate the bytecodes that correspond to domain annotations, replacing these annotations by their corresponding annotations. Essentially, it looks for annotations annotated with `@DomainAnnotation`, reads the elements of the indicated classes and translates the domain annotations to their corresponding annotations. For such, all the developer has to do is create a template class, annotate its elements with annotations of existing frameworks and create the domain annotation itself, indicating the template class, as shown in Figure 3. It is also possible to annotate a domain class element with two or more domain annotations, or even to annotate a domain annotation with other domain annotations.

Figure 4 shows the equivalent code of the `placeCurrentOffer` method, shown in Figure 3, after Daileon is run.

5. CONCLUSION

This paper proposes two techniques for enabling the use

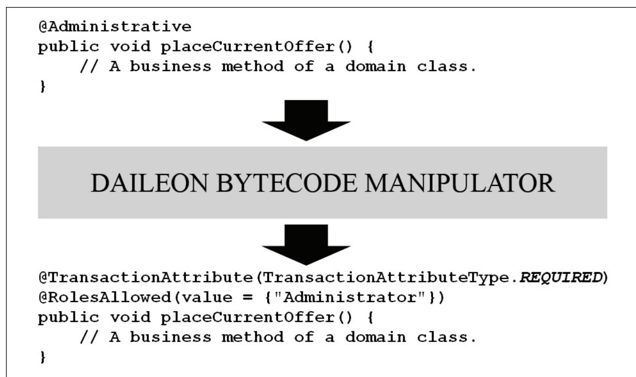

Figure 4: A domain annotation translated to the known annotations of the EJB 3 API.

of domain annotations on Java-based applications: one for new frameworks, where they support the creation of domain annotations, and another one for existing frameworks that do not support their creation. When creating them to be used with new frameworks, these frameworks must allow annotations to be annotated with their annotations and also must be able to identify domain annotations. For existing frameworks that do not support this concept, some bytecode manipulation is required, in order to replace the domain annotations by their corresponding annotations before the application is run. The Daileon framework provides these two functionalities, and thus enables such development.

Frequently, annotations that are related to infrastructure are added to elements that should relate exclusively to the domain, which shows that concerns are not clearly separate. Sometimes, it may be necessary to annotate a class element with several annotations, which may pollute the code and decrease its readability and maintainability. For the domain annotations scenario, a rich domain model is required, which leads to the domain-driven design approach. Not only it promotes a better object-oriented structure, it also promotes reusability.

Domain annotations bring more quality to the code, reducing the number of annotations in the domain model implementation, increasing its readability and maintainability and bringing more cohesion and consistency to the domain model. When applied, annotations placed on domain objects clearly have a relationship with the domain itself, not with matters out of it, and thus developers can concentrate their efforts exclusively on business-related concerns. The dependency on annotations of external frameworks that has to exist so the application can be created is also encapsulated, and when these frameworks have to be replaced, the code that reflects the domain model implementation does not have to change. At most, the domain annotations would have to be translated again to their corresponding annotations before running the application, which can be easily done with the Daileon framework.

Future works to be done regarding domain annotations will show the results of the usage of the techniques proposed by this paper, using Daileon in a real application, and will also contemplate techniques to create frameworks that support domain annotations in a clean and practical way. Other points to be covered are how to apply domain annotations in aspect-oriented frameworks, how to parameterize domain annotations, and also how to create domain annotations for parameters of domain elements.

6. REFERENCES

[1] A metadata facility for the java programming language. http://jcp.org/en/jsr/detail?id=175, 2003.

[2] E. Doernenburg. *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, chapter 10. Pragmatic Bookshelf, Raleigh, NC, USA, March 2008.

[3] Enterprise javabeans 3.0. http://jcp.org/en/jsr/detail?id=220, 2006.

[4] E. Evans. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*. OOPSLA, October 2000.

[6] E. M. Guerra, J. O. Silva, F. F. Silveira, and C. T. Fernandes. Using metadata in aspect-oriented frameworks. In *2nd Workshop on Assessment of Contemporary Modularization Techniques (AcoM.08)*. OOPSLA, October 2008.

[7] A. C. Neto, M. de Medeiros Ribeiro, M. Dosea, R. Bonifacio, P. Borba, and S. Soares. Semantic dependencies and modularity of aspect-oriented software. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 11, Washington, DC, USA, 2007. IEEE Computer Society.

[8] J. R. C. Perillo. The daileon framework. http://sourceforge.net/projects/daileon/, 2009.

[9] R. Rouvoy and P. Merle. Leveraging component-oriented programming with attribute-oriented programming. In *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP 2006)*, Nantes, France, Jul 2006.

[10] The javabeans specification. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[11] Xdoclet: Attribute-oriented programming. http://xdoclet.sourceforge.net/xdoclet/index.html.