# 1 Java Annotations

## 1.1 Introduction

The purpose of this thesis is to evaluate the usefulness of Smart Annotations. Of course, before we can discuss anything alike, we first have to explain what Smart Annotations are and why they came into existence. The answer to that last question lies with Java annotations. For this reason we'll first give more information about them before continuing with the rest of this document.

First we'll explain how metadata can be incorporated into source code (Section 1.2). Then we'll be more specific and talk about one kind sort of metadata, specifically Java annotations (Section 1.3). We'll start by saying why they are used (Section 1.4), before we discuss how they are used and what kind of problems the use of Java annotations creates (Section 1.5).

## 1.2 Metadata in source code

Additional information about source code used to be only accessible in the documentation of the source code. For example when you wanted to specify the characteristics that are used to enforce security, this had to be done in an external file. Modifiers can be used to specify certain properties of source code, like the accessibility to other classes with the keywords `public`, `private` and `protected`, but these modifiers are all predefined. Also, compilers are explicitly designed to recognize these predefined keywords, which don't allow a developer to introduce his own metadata.

Now it is possible in different language for a developer to add metadata to his code. In C# attributes can be used for this purpose. A developer can create his own descriptive elements in C# without having to rewrite the compiler. All he has to do is declare an instance of one of the special classes that derives from System.Attribute and place it inside square brackets just before the associated declaration along with any arguments. Attribute information can later be retrieved at runtime using reflection.

Pragmas in Smalltalk serve the same purpose. They are a standardized form of comment that has meaning to the compiler although they don't change how a method is executed. They can be attached to method definitions and then queried from the language, which makes them useful as declarative registration mechanisms.

Java also offers a means to add metadata to source code. There a developer can create annotation types, which he can then use to annotate his code. Since this document is about Smart Annotations, which relies on Java annotation, the focus will be on them.

Before Java annotations were available there was no standard mechanism to insert metadata in the code, so developers had to find different ways to try to get the same result. In J2SE 1.4 and earlier examples of metadata were

- The `transient` keyword,
- Marker Interfaces (Serializable Interface, SingleThreadModel Interface, …),
- Xml descriptors
- The META-INF/MANIFEST.MF file,
- The BeanInfo interface,

- @deprecated Javadoc comment,
- XDoclet Javadoc tags.

These are all different ways to achieve the same result, adding metadata, but it's hard to generalize them. Like mentioned before users can't create their own keywords, so using keywords doesn't scale. Using marker interfaces isn't useful either since they don't provide any information beside their existence. Another drawback they have is that they only work on classes, unlike annotations, which can be placed on classes, methods, variables, parameters and packages. The new metadata facility, annotations, solves some of the drawbacks of these methods.

In many cases, XML support files are still a good way to hold metadata in, but it in some cases it can be more efficient to put information within the code so that it can be directly linked to it. It's a problem of locality versus modularity. By using annotations, it is possible to let the XML descriptor files only contain the deployment-related decisions.

In the following section, Java annotations will be explained in further detail.

## 1.3 Java Annotations

Java annotations are a form of metadata that can be inserted into source code. They don't directly influence the program semantics, but when the Java source is compiled, the annotations can be used for reflection or can be processed by compiler plug-ins called annotation processors. These can produce information or additional Java source files or resources, which can then be compiled and processed. If the retention policy of an annotation is CLASS or RUNTIME the Java Virtual Machine or other programs can look for the metadata and then determine how to interact with the program elements or change their behavior. Because then the Java compiler also stores the annotation meta-data in the class files.

Java Annotations can be used for several purposes:

- Information for the compiler
  - To detect errors or suppress warnings
- Compiler-time and deployment-time processing
  - Software tools can process annotation information to generate code, XML-files, …
- Run-time processing
  - Some annotations are available to be examined at run-time

Annotations can be added to the Java source code as a modifier, it consists of the name of an annotation type and zero or more element-value pairs, each of which associates a value with a different element of the annotation type. With these elements a developer can associate information with the annotated program element. Since annotations are a special kind of modifier, they can be placed on classes, methods, variables, parameters and packages, just like any other modifier. But unlike other modifiers, annotations can have parameters to populate their data members and can be declared by the user. There also exist some pre-defined annotations, which we'll discuss shortly in the following sections.

### 1.3.1 Predefined Annotations

There are seven predefined annotations:

- @Override
  - used on methods from a subclass to indicate that this method should override the corresponding method in the superclass.
- @SuppressWarnings
  - used in situations where the developer expects warnings but wants the compiler to suppress them.
- @Deprecated
  - indicates that a Java Element is deprecated and shouldn't be used anymore.
- @Target
  - used to specify the target elements that can be annotated with the annotation type that is being declared. There are seven possible target elements: TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE and ANNOTATION_TYPE.
- @Retention
  - used to specify where and how long the annotation should be retained. There are three possible retention policies:
    - CLASS (default): annotation is retained by the compiler at compile-time, but will be ignored by the Virtual Machine;
    - RUNTIME: annotation is retained by the Virtual Machine so it can be read at run-time;
    - SOURCE: annotation is retained at the source level and will be ignored by the compiler.
- @Documented
  - used to specify that the declared annotation should be included in the Javadoc.
- @Inherited
  - used to specify that subclasses should inherit annotations. This means that when a developer wants to know whether a class has this annotation but it doesn't, then the query the developer performed will go up the hierarchy of the class. This until the specific annotation is found or the top of the hierarchy is reached. In the former case, the query will return true, which means that this class is annotated with this annotation and in the latter case, the result is false.

The first three annotations can be used on Java elements like classes, methods, variables, parameters and packages while the rest of the annotations are considered to be meta-annotations because they can only be placed on an annotation type declaration.

Besides the predefined annotations available in Java, it is now also possible to create your own annotations. In the next section we'll discuss how this is done.

### 1.3.2 Defining New Annotations

Next to the predefined annotations, there's also the possibility to create your own domain-specific annotations. In this section, an example of how such an annotation can be created is presented. In this example we'll declare an Enterprise JabaBeans annotation. EJB introduced annotations in version 3.0 as an alternative way to achieve persistence. It is still possible to use XML descriptor files to specify the relational mappings between domain objects and the database. These files can also be used to override the annotations.

The following code snippet shows the declaration of the @Entity annotation. The other annotations available in the EJB framework will be discussed in a later chapter (Chapter 4).

```
@Target(TYPE)
```

```
@Retention(RUNTIME)
public @interface Entity {
	String name() default "";
}
```

**Code 1: EJB Entity annotation type declaration**

From this code snippet we can determine that the @Entity annotation can only be placed on type declarations; this is enforced by the meta-annotation @Target on top of the annotation type declaration. As mentioned in the previous section (section 1.3.1) the argument value RUNTIME to the meta-annotation @Retention specifies that the Java Virtual Machine should retain the @Entity annotation so it can be read at run-time.

The declaration of annotation types is very similar to interface declaration which is also noticeable by the fact that the keyword used for declaring an annotation type is also `interface` but then preceded by an at-sign. Annotation type elements are declared using method declarations but here they can't have any arguments or a `throws` clause and the return types are restricted to primitives, `String`, `Class`, enums, annotations and arrays of the preceding types. These methods act as getters so that these elements can be called at run-time. The `default` keyword is used to specify the default value of element. When an annotation is used without specifying the value for an element that doesn't have a default value, a compile error is thrown. In the example of the Entity annotation one element is declared, name. It is used to refer to the entity in queries.

Now that we're able to declare an annotation type, we can use the declared annotation in our source code. When we want our class Person (Code 2) to be seen as an entity, we just place the Entity annotation on top of the class declaration. Since the only element declared inside the Entity annotation has a default value, it is not necessary to specify one when using the annotation. Now an EntityManager can use this class as an entity.

```
@Entity
public class Person {
	@Id
	private String name;

	public void setName(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}
}
```

**Code 2: Entity class Person**

Because the Entity annotation only has one element, name, it is also called a single-element annotation. Most of the time, the element will then be called value so that when using the annotation only the value for that element is specified instead of the element-value pair. The value element, because of its special name, is assumed when the annotation is only passed one argument. Beside this sort of annotation type, you can also have one without any elements and such annotations are called marker annotation. If the annotation type declaration contains more than one element it is called a full-value or multi-value annotation and then it is necessary to use element-value pairs when specifying the arguments.

## 1.4 Uses of Java Annotations

Since annotation are just meta-data and can be used to associate this data with the annotated code. A misconception that many developers have is that annotations do not affect the semantics of a program. It is true that they don't directly change the semantics. But there are tools that do change them and they can do this based on the presence or absence of annotations. When such a tool reads the annotations, it can process them in some fashion and produce additional Java source files, XML documents or other files that can be used in combination with the program containing the annotations. The EntityManager of the EJB framework is an example of this. To look back at our entity Person (Code 2) we see that name is annotated with the @Id annotation. This annotation can be used by the EntityManager to automatically assign an identity to the entity. Another annotation we have mentioned before is @Deprecated. It doesn't change the behavior of deprecated elements but it does tell the compiler to produce warnings when such elements are being used. So indirectly annotations do affect the semantics of a program, which developers tend to forget when writing their code.

Besides annotations, also a new interface, the java.lang.reflect.AnnotatedElemented interface, is supported by Java 5. The reflection classes in Java (Class, Constructor, Field, Method and Package) implement this interface which gives access to the annotations that have run-time retention via the getAnnotation(), getAnnotations() and isAnnotationPresent() methods. Since annotation types are just like regular Java classes, the annotations returned by these methods can be queried just like any regular Java object.

Tools to write boilerplate code use annotations and this leads to a declarative programming style. Developers can just say what should be done by using annotations and the tools produce the code to do it. This is useful when there are many repetitive coding steps.

Annotations are mainly used by frameworks, which we'll discuss in the following section (Section 1.4.1). But they can also be used in combination with aspects, which will be explained later in this section (Section 1.4.2).

### 1.4.1 Frameworks

Frameworks often use annotations as a way of conveniently applying behaviors to user-defined classes and methods that otherwise would have been declared programmatically or in external sources like XML configuration files.

One of those frameworks is JavaServer Faces (JSF) 2.0, which introduces new annotations to mark classes as managed beans. The annotations are a replacement for the XML entries in the faces-config.xml file. Although it is still possible to use this file, the annotations will only be considered if the file isn't there or the metadata-complete attribute is not set to true. Composite components, converters, validators and renderers can now be indicated as such by the use of annotations.

The Spring framework supports two different kinds source-level metadata, attributes and annotations. The framework has specific Java annotations for transactional demarcation, JMX and aspects (AspectJ annotations). For example the Transactional annotation is used for transaction configuration and the Required annotation for marking a field as required so that an IllegalArgumentException will be thrown if this field is not set. Of course not just the annotations are

needed to get this behavior. A component, the RequiredAnnotationBeanPostProcessor, is necessary to be aware of the Required annotations and to process beans appropriately.

The Struts framework uses annotation for actions, interception, validation and type conversion. Instead of specifying individual mappings through XML configuration, the action annotations are a set of annotations that can be used to specify actions and can be used to allow the framework to find Action classes when it scans the class path. The interceptor annotations are used to say when an action method should be executed. The validation annotations are used to mark fields that need to be validated in a certain way. The type conversion annotations can be used to avoid using any ClassName-conversion.properties files.

Another framework that started using annotations since the availability of Java 5 is Enterprise JavaBeans (EJB). There annotations can be used to define a bean's business interface, the O/R mapping information, resource references and anything else that can be defined through deployment descriptors or interfaces in previous versions of the framework. This makes the deployment descriptors and the home interface redundant and it's no longer necessary to implement a business interface because the container can generate one.

Many frameworks make use of the Java annotations, which means that they rely on the developer to use them correctly. Incorrectly annotated code is like having an error in a configuration file, whether it is a missing annotation or one in that is in the wrong place. The consequences of using annotations incorrectly will be handled in section 1.5 but first we'll have a look at how annotations are used in combination with Aspects.

### 1.4.2 Aspects

Since AspectJ 5, it is also possible to declare aspects in an annotation-based style. This style is an alternative for the code-based declaration of aspects which of course still can be used and they can even both be used in the same application. The AspectJ weaver ensures that both styles result in the same semantics. The new annotations that can be used in the annotation-based style are called the @AspectJ annotations. This new way of declaring aspects allows them to be compiled by a regular Java 5 compiler and then woven by the AspectJ weaver. This shows that AspectJ introduced annotations just like other frameworks which were previously mentioned (1.4.1). But besides that, AspectJ also included constructs so that it could support matching join points based on the presence or absence of annotations and gave the means to expose the annotation values within the body of advice.

Aspects often rely on implicit rules when to declare their pointcuts. For example, there could be a convention saying that every test class has a name that ends with the word Test. Then the pointcut definition could be relying on the fact that developers follow this convention and in say in their pointcut that they only want classes with a name that ends with Test. This is described as the fragile pointcut problem, which specifies that pointcut definitions can easily include unintended join points and exclude intended join points. Annotations can be used to make these conventions the pointcuts rely on, explicit. This doesn't solve the fragile pointcut problem completely because it is still possible to place annotations incorrectly or to forget to them. In the next section (1.5) we'll explain the problems that occur when this happens.

## 1.5 Java Annotations and evolution

Java Annotations are a convenient way to add metadata to source code but the responsibility of annotating this code lies with the developer. A framework that relies on annotations can't work correctly if not all the annotations are present or if the wrong piece of code was annotated. The more the code evolves, the higher the probability of such a situation.

We'll illustrate this with an example (Section 1.5.1). Then we'll explain what the fundamental problem of annotations is (Section 1.5.2) and how this causes the need for tool support (Section 1.5.3).

### 1.5.1 Illustrating example

For illustrating the problem that can arise when source code evolves, we'll look at a situation in the EJB framework. We'll show how a class is made persistent using the framework and then we'll see what happens when the source code of our example changes but without considering the annotations in the code.

If we consider a class Customer like in the following code (Code 3).

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable
{
   private int id;
   private String name;
   private Set<Order> orders;

   @Id
   @Column(name="ID", unique=true)
   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   @ManyToOne(cascade=ALL, mappedBy="customer")
   public Set<Order> getOrders() {
      return orders;
   }

   public void setOrders(Set<Order> orders) {
      this.orders = orders;
   }
}
```

**Code 3: Customer class (Version 1)**

We use the Entity annotation on top of the class to allow the EntityManager to manage this class as an Entity. Several other annotations are used to specify the mapping with the database schema. With the Table annotation we can say in which table we want to store the instances of this entity, in this case we'll store all Customer objects in the CUSTOMERS table. Each table must have a primary key and to indicate that a field represents that key it can be annotated with the @Id annotation. By default all the fields of an entity class are persistent but if a developer wants to specify in which column a certain field should be stored, he can use the Column annotation. This annotation has several optional arguments, two of which are specified in our example for the `id` field. With the name argument, the developer can give the name of the specific column this field should be stored in and with the boolean value unique he can say that this column should only contain unique values. The relationship between the CUSTOMERS table and the ORDERS table is made with the @ManyToOne annotation. The argument cascade specifies the set of cascade operations that are propagated to the associated entity, in this case all operations. The mappedBy argument identifies the field in the Order class which owns the relationship. In our example it means that the Order class contains a field customer of the type Customer that has an annotation @OneToMany.

When the Customer class is declared like this, the EntityManager will have no problem keeping the entity persistent. Of course the database has to be configured correctly as well. If later the source code of the project this class is used in, evolves and this class requires a new field just for some computational reason, then a developer shouldn't forget that by default all fields are persistent. If he did forget that and he just added this field without saying it should be transient, then the EntityManager will try to include this field in the CUSTOMERS table. This could cause an error, if the table is static and the developer has to create the column himself if he would a new one. Because the EntityManager will try to insert this field into a column that doesn't exist. It could also be that the table is more dynamic and a new column is created for this field. But this is also not desirable; we don not want to save this field in the database.

We can illustrate this problem with an example. Let’s say we have a situation like in Code 3. There we have three fields that are persistent and are stored in the table CUSTOMERS of a static database. If we now add a new field,  count, which will only be used for computational reasons inside the program and thus the developer doesn’t want to store it in the database. The database remains the same but the Customer class is changed as follows.

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable
{
   private int id;
   private String name;
   private Set<Order> orders;
   private int count;

   …
}
```

**Code 4: Customer class (Version 2)**

This new version of the Customer class may seem correct but when running the code an error will arise from the moment the program wants to store an instance of this class in the database. Because

there is no column in the database that can store the value of the count field. The correct way to change the class is to add the new field but annotated with the @Transient annotation like in Code 5. Then the tool that takes care of the persistency knows that the count field should not be stored in the database.

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable
{
   private int id;
   private String name;
   private Set<Order> orders;
   @Transient
   private int count;

   …
}
```

**Code 5: Customer class (Version 3)**

Another problem that could arise when the code of the Customer class evolves, is that a new field is added, a Date field specifying the birthday of the customer seen in Code 6. The developer can add this field just like any the other fields. But he does have to remember that a persistent field of the type Date should always be annotated with the @Temporal annotation. If not, an exception will be thrown with the following description: "The attribute [someDate] from the entity class [class domain.Customer] does not specify a temporal type. A temporal type must be specified for persistent fields or properties of type java.util.Date and java.util.Calendar".

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable
{
   private int id;
   private String name;
   private Set<Order> orders;
   private Date birthdate;

   …
}
```

**Code 6: Customer class (Version 3)**

Even a small change in the source code can cause the project to throw exceptions. This could happen when we were to change the field name of customer in the Order class to another name, for example client, and we'd change every occurrence of that field to the new name like in Code 7. Since this is a private field, we would probably just think that this field is only used inside this class and that other classes just use the getter for this field. The relationship to the database isn't affected because we used the Column annotation to specify the name of the column we want the field to be saved in and this hasn't changed. But what we could be forgetting is that in the Customer class we used the name of this field as a value to the mappedBy argument in the @ManyToOne annotation on the `getOrders()` method. Now the EntityManager will try to find a field named customer in the Order class but that isn't present there and then an exception will be thrown.

```
@Entity
@Table(name=”ORDERS”)
public class Order implements java.io.Serializable
{
      private Customer client;

      …
}

@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable
{
   …

   @ManyToOne(cascade=ALL, mappedBy="client")
   public Set<Order> getOrders() {
      return orders;
   }
}
```

**Code 7: Customer class (Version 3)**

There are many situations where annotations can be used in a wrong way or just forgotten. As stated in the example above, in many cases this will cause an error in the program. It's also possible that the program runs fine but that just the data isn't consistent and thus unreliable. For example if the developer were to forget to add mapping annotations, then the database would contain tables without any relationships between them. If data is inconsistent then the behaviour of the program is also unpredictable.

To show that implicit rules are not only characteristics of EJB annotations, a few examples from the Spring framework will be mentioned.

In the Spring documentation it states that the Transactional annotation should only be placed on methods with public visibility. Otherwise the method will not exhibit the configured transactional settings. So while a developer thinks his code is transactional, since he annotated it as such, this is not the case and not one warning is given to the developer.

### 1.5.2 Fundamental problem

The example in the previous section (Section 1.5.1) shows that when writing or altering source code, annotations should always be kept in mind. Both the annotations that are already present on the code and all the other annotations related to the task the code performs. Like in the previous example all the annotations related to persistence should be considered whether they are applicable or not.

Many annotations are associated with a set of implicit rules for when they should or shouldn't be used. Just like in the example in the previous section (1.5.1), the Temporal annotation can only be used on fields of the type Date or Calendar and are not permitted on any other type of field. But it's also necessary that every field that is persistent and of the type Date or Calendar is annotated with this annotation. If a framework is used like EJB, it's necessary to go thoroughly through the documentation to be sure not to miss any of these implicit rules. But when a developer is working on

a project for a long time, he isn't necessarily keeping all the annotations with all their rules in mind. Then it can easily happen that he forgets a small, but still important, rule of such an annotation.

With new code, it can be considered acceptable that the developer should think of the needed annotations himself. But when code is already annotated and by changing some other piece of code, the annotation on this code is suddenly no longer valid, then you can hardly blame the developer for this. The link between code and annotation is very weak as it can break easily. The developer is entrusted with the responsibility of keeping this link intact.

### 1.5.3 Need for tool support

The developer is only human and the human mind can always forget something. Especially when his mind is not occupied with the specific subject at a regular interval. The implicit rules applicable on annotations can be considered as such a subject. The developer might inform himself of these rules when he is working on the task that these annotations help to perform but once he has moved on to the next phase of the project, there's nothing reminding him of these rules. If it is necessary in the next phase to change the source code of the project, he can easily forget one of the many rules associated with all the related annotations.

These rules are only mentioned in the documentation of the annotations, if that is even the case. Sometimes a developer just becomes aware of such a rule out of experience. Besides that there is nothing forcing him to apply these rules in his code and certainly not warning him when he has forgotten to place an annotation or placed it incorrectly. It was obvious then that there was a need for some kind of tool support, which could inform the developer that an annotation is misplaced or missing. These people decided to work out a concept that could help with this task and that's how the idea for Smart Annotations was formed.

## 1.6 Conclusion

Java annotations bring the metadata closer to the source code, in comparison to configuration files and many tools and frameworks have already picked up on this new facility Java 5 offers. But these annotations also bring with them a set of rules, which aren't always that explicit and the developer is left the responsibility of informing himself with these rules. There does not exist a mechanism yet that checks all the requirements of the annotations. In the next chapter (Chapter 2) Smart Annotations are introduced as a possible solution for this problem and there we'll explain what they exactly are.