Features of Annotations and their Applications

Shiva Prakash
Department of Computer Science & Engineering,
Madan Mohan Malaviya Engineering College,
Gorakhpur, U.P., INDIA

J.P. Saini
Madan Mohan Malaviya Engineering College
Gorakhpur, U.P., INDIA

R. K. Singh
Department of Electronics & Communication
Engineering, B.C.T. Kumaon Engineering College,
Dwarahat, Uttarakhand, INDIA

Karan Singh
School of ICT, Gautam Buddh University
Greater Noida, U.P., INDIA

ABSTRACT
Annotations are descriptive declarative information that can be associated with program elements. They can be used to represent metadata. This is useful in many cases like providing documentation, connecting to database etc. These annotations are later read by the execution environment and appropriate action will be taken. For custom annotations, a process called reflection is used to take necessary action. Although Java introduced this feature recently, attributes / annotations have been a topic of interest for researchers. There are tools and applications which make use of annotations. This paper, we will first have a look at annotations and annotation types. Then we go on to discuss in detail how annotations support injecting dependencies into resources like data sources, mail sources, environment entries, EJBs, web services, and so on. The new features are targeted to shift the responsibility of writing the boilerplate code from the programmer to the compiler or other tools. The resulting code is less likely to be bug-prone.

Keywords
Annotations, Security, Dependency Injection, Transactions..

1. INTRODUCTION
Annotations were in use for a long time. Researchers have used annotations for various purposes like optimization, automatic documentation etc. The oldest publication regarding the uses of annotations dates back to 1978 in which the authors use them for automatic documentation of Algol programs. Annotations were proposed for many languages like Ada, C etc but they were not implemented. .NET first implemented them and called them as attributes. It has a very rich set of attributes. Later on, Java in its latest version J2SE 1.5 introduced annotations [2][4]. One of the important objectives of Java EE 5 Platform is ease of development [8]. The new features are targeted to shift the responsibility of writing the boilerplate code from the programmer to the compiler or other tools. The resulting code is less likely to be bug-prone. One of these new ease-of-development features is the metadata facility-*annotations*. Annotations provide a mechanism for decorating Java classes, interfaces, fields, and methods with metadata information. Annotations are considered an alternative to the XML files required by the earlier versions of Java EE. Annotations are also used for dependency injections of resources, web services, and lifecycle notifications into a Java EE 5 platform application. In this paper, we will first have a look at annotations and annotation types. Then we go on to discuss in detail how annotations support injecting dependencies into resources like data sources, mail sources, environment entries, EJBs, web services, and so on.

J2SE 5.0 introduced a new facility known as annotations. Annotations are a metaprogramming facility that allows you to mark code with arbitrarily defined tags. The tags (generally) have no meaning to the Java compiler or runtime itself. Instead, other tools can interpret these tags. Examples of tools for which annotations might be useful include IDEs, testing tools, profiling tools, and code-generation tools [3][11]. In this paper you will learn to build a testing tool, similar to JUnit, based upon Java's annotation capabilities. The testing tool, like JUnit, will need to allow developers to specify assertions. Instead of coding assertion methods, the Java's built-in assertion capabilities are as follows:

- assertions
- annotations and annotation types
- retention policies for annotations
- annotation targets
- member-value pairs
- default values for annotation members
- allowable annotation member types
- package annotations
- compatibility considerations for annotations

JSR 308 extends Java's annotation system [17] so that annotations may appear on nearly any use of a type. This generalization removes limitations of Java's annotation system, and it enables new uses of annotations. This proposal also notes a few other possible extensions to annotations. This paper specifies the syntax of extended Java annotations, but it makes no commitment as to their semantics. As with Java's existing annotations [17], the semantics is dependent on annotation processors and not every annotation is necessarily sensible in every location where it is syntactically permitted to appear. Common Annotations for the Java Platform [13], and proposed annotations, such as those to be specified in JSR 305, Annotations for Software Defect Detection [18]. The proposal merely makes annotations more general and thus more useful for their current purposes, and also usable for new purposes that are compatible with the original vision for annotations [16]. A JSR, or Java Specification Request, is a proposed specification for some aspect of the Java platform.

1.1. Why Annotations?

In programming paradigm, there are two approaches: *imperative programming* and *declarative programming* [1][7]. Imperative programming specifies the algorithm to achieve a goal, whereas declarative programming specifies the goal and leaves the implementation of the algorithm to the support software. In other words, "specifying how" describes imperative programming and "specifying what is to be done, not how" describes declarative programming. The Java EE platform has always been very supportive for using a declarative approach. Many of the APIs require boilerplate code, using external files-deployment descriptors-to specify the declarative information [6]. For example, in an Enterprise JavaBean (EJB), much of the information about the bean goes into the deployment descriptor file. It would be better and more convenient if the code that goes into the deployment descriptor were to be maintained as part of the program itself, but in a declarative way. This is where the actual usage of annotations comes in: they allow us to move the information into the source code and thus help in avoiding unnecessary boilerplate code. Writing EJBs using the annotations in EJB 3.0 is much easier compared to in earlier releases.

Annotations significantly simplify the developer's work-instead of using an API [1]or creating an additional deployment descriptor to request the container to do something, the request for the container can be put in the code directly. The biggest advantage of annotations is that they significantly reduce the amount of code a developer needs to write. Another advantage is that they are very open and flexible in terms of the functionality they offer, so that types of logic that were tedious to write using earlier versions can be written easily using annotations.

1.2 Motivation

Many APIs require “side files” to be maintained in parallel with the programs. These are maintained for configuration, mapping, and documentation purposes. In Hibernate[1], an XML file which contains mapping information is maintained. Hibernate is a powerful, ultra – high performance object/relational persistence and query service for Java. Hibernate lets us develop persistent classes and along with the classes their associations can also be persisted. The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational model with a SQL – based schema. It provides simple ‘load’ and ‘save’ operations using which complex SQL statements could be avoided. Hibernate needs to know how to load and store objects of the persistent class. This is where the Hibernate mapping XML file is used. The mapping file tells Hibernate what table in the database it has to access, and what columns in that table it should use.

We have a class Event which contains the information about the events like the date and name. To make a class persistent, it has to contain a unique id. The Event class will look as follows:

```
import java.util.Date;
public class Event {
private Long id;
private String title;
private Date date;
protected Event() {}
public Long getId()
{
    return id;
 }
public void setId(Long id)
{
  this.id = id;
 }
public Date getDate()
{
   return date;
 }
public void setDate(Date date)
{
   this.date = date;
 }
public String getTitle()
{
   return title;
 }
public void setTitle(String title)
{
  this.title = title;
 }
}
```

The XML mapping file for the above class will look as follows:

```
<hibernate-mapping>
   <class name="Event" table="EVENTS">
      <id name="id" column="EVENT_ID">
         <generator class="increment"/>
      </id>
      <property name=”date” type=”timestamp”
                column=”EVENT_DATE”/>
      <property name="title"/>
   </class>
</hibernate-mapping>
```

This shows that the ‘Event’ class should be mapped to the Events relational table ‘Event’ and the properties of the class should be mapped to their respective columns in the table [1][3]. The id element is the declaration of the identifier property, name="id" declares the name of the Java property - Hibernate will use the getter and setter methods to access the property. The column attribute tells Hibernate which column of the EVENTS table we use for this primary key. The nested generator element specifies the identifier generation strategy; in this case increment was used, which is a very simple in-memory number increment method useful mostly for testing (and tutorials). Hibernate also supports database generated,

globally unique, as well as application assigned identifiers (or any strategy we have written an extension for).

This XML file was a side file which has to be maintained along with the program files. The disadvantage with this approach was that if something was changed in the program file then the developer should remember to change the XML mapping file as well. This was the case till Hibernate version 3.0.5. But Hibernate 3.1 used Java annotations for this mapping making this a simple process. These annotations could be written along with the source code files. So if there are any changes to the source code then it becomes easy to change the mapping part as well. The annotated class 'Event' [9][10] will look as follows:

```
import java.util.Date;
@Table (name="Events")
 public class Event
 {
  private Long id;
  private String title;
  private Date date;
  protected Event() {}
  @Id (generator=GeneratorType.SEQUENCE)
  public Long getId()
  {
      return id;
   }
  public void setId(Long id)
  {
    this.id = id;
   }
  @Column (name="EVENT_DATE")
  public Date getDate()
  {
     return date;
   }
  public void setDate(Date date)
  {
     this.date = date;
   }

@Column (name="TITLE")
public String getTitle()
{
   return title;
 }
 public void setTitle(String title)
 {
    this.title = title;
  }
}
```

The above code looks more compact and it is easy to maintain it when compared to the XML mapping technique used previously.

2. ANNOTATIONS

Annotations [2] are declarative and descriptive statements which represent metadata. In Java, annotations were introduced in its latest version J2SE 1.5 (called version 5) code named Tiger [14][17]. In previous versions of Java, there was a very limited support for including metadata in the form of javadoc tags. These tags were mainly for documentations purposes. But with J2SE 5, Java's metadata capability went to a new level. It provides several built-in annotations and we can create our own annotations as well. These are called custom annotations.

2.1 Built – in Annotations

There are six built – in annotations. They are:

Overrides: They are used only on methods. They are used to specify that a method on which it is decorated must override a method in its superclass.

Documented: This is applied only on other annotations. This is a meta-annotation. It is used to specify that the annotation should be documented. This is just a hint.

Deprecated: This is used to give a hint to the compiler to warn the users who are using the element this was annotated with. A better alternative for the element would exist.

Inherited: This is a meta-annotation. If an annotation is decorated with this one then all the subclasses of a class will also inherit this annotation

Retention: This is also a meta-annotation. This is used to specify the decorated annotation's availability. There are three values for this one – source, class, runtime.

Target: This annotation is used to indicate the type of program elements to which the declared annotation is applicable. The possible program elements to which annotations can be applied are class, method, field, package declaration, constructor, parameter, local variable or another annotation.

Eg:
```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target
{
   ElementType[] value();
 }
```

Except for Overrides annotation, all the others can be used as meta-annotations i.e. they can be used to annotate another annotation just like meta meta-data.

2.2 Custom Annotations

Apart from these built-in annotations, we can create our own annotations which are called custom annotations. The general syntax for creating an annotation is as follows:

```
@Typename
{
   MemberValuePairs
 }
```

Any member can be given a default value.

Eg:
```
public @interface RequestForEnhancement
  {
     int id();
     String synopsis();
     String engineer();
```

```
        String date();
    }
```

This annotation can be used as follows

```
@RequestForEnhancement(id=1234, synopsis="provide better GUI",
  engineer="John", date="12/10/2005");
  public void UIManager()
  { }
```

There are other types of annotations called marker annotations which have no members and single member annotations which have only a single member.

2.3 Annotations for Transactions

Annotations can be used to specify the transaction management types [5], instead of writing transaction-related XML tags in the deployment descriptor. The annotation @TransactionManagement specifies whether a bean uses container-managed or bean-managed transactions. The type of transaction is specified using TransactionManagementType in the value element of the annotation. The default type is container-managed transaction. To specify the transaction attributes to all methods of a business interface or to the individual methods of the bean class, the @TransactionAttribute annotation can be used. This annotation requires a single argument of the enumerated type TransactionAttributeType, the values of which are MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, and NEVER. If the @TransactionAttribute annotation is not specified, the default attribute REQUIRED will be applied. These annotations are defined as part of EJB 3.0 specification available in the javax.ejb package.

```
.....
@TransactionManagement
  (value=TransactionManagementType.CONTAINER)
public class StockQuoteImpl implements StockQuote{
  .....

@TransactionAttribute(TransactionAttributeType.REQUIRED)
  public double getStockQuote(String symbol) {
    .....
  }
    .....
  }
```

2.4 Annotations for Security

Security for enterprise applications can be specified using the deployment descriptors [3], using either the declarative or programmatic model. Starting with the Java EE 5 platform, annotations can be used to specify security constraints instead of using deployment descriptors. To support the declarative security model, we have annotations defined in JSR 250 to facilitate the usage of security annotations to all managed components. The annotation @DeclareRoles is used to declare security roles. This annotation provides support for declaring more than one role. Other security related annotations, including @RolesAllowed, @DenyAll, and @PermitAll, give permission to the role to invoke the methods.

```
@RolesAllowed("admin")
public class Department {
  public void setDeptId () {...}
  public void getDeptId () {...}
  ...
}
public class RegularCourse extends Course {
  @RolesAllowed("courseowner")
  public Course addCourse() {...}
  @RolesAllowed("admin")
  public void removeCourse() {...}
  ...
}
```

2.4.1 Annotations for Dependency Injection:

Annotations serve multiple purposes [9][15]. Some annotations provide an alternative to XML deployment descriptors, while some annotations allow components to request the container's help for doing certain tasks that the components would otherwise have to perform themselves. While we have already discussed the first type, we will now look at this second type of annotation. Most enterprise application components like servlets, JSPs, and EJBs used in enterprise applications use external resources and services such as data sources, mail sources, environment entries, and EJB context. Prior to Java EE 5, applications had to explicitly declare their dependency on these external resources in the deployment descriptor files, and obtain a reference to these resources using JNDI. Developers found it very difficult to understand this model because of its complexities. To reduce the complexity of this model, Java EE 5 introduced another important concept called dependency injection.

Dependency injection is a mechanism [2] in which a component's dependencies are supplied by the container. The dependency on a resource is marked using annotations or declared in the deployment descriptor file. Dependency injection (aka "resource injection") is nothing but the inverse of JNDI. Instead of the component explicitly requesting a resource, now the Java EE 5.0 container injects an instance of the resource when it's required. The injection could be at the field level or at the method level (typically on a setter method). The Java EE 5 platform defines annotations to support a variety of injections.

Dependency injection can be applied to all resources that a component needs. Dependency injection can be used in EJB containers, web containers, and application clients. However, resource injection can be requested only by components that are managed by the container. To request injection of any type of resource, a component uses the @Resource and @Resources (for more than one resource) annotations. These annotations are defined as part of JSR 250, in the javax.annotation package. To request injection of an EJB, the @EJB and @EJBs annotations are used, defined in the javax.ejb package as part of JSR 220. To request injection for web services, @WebServiceRef and @WebServiceRefs annotations are used. Annotations for web services are defined as part of JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0.

The following resources can be injected using the @Resource, @EJB, and @WebServiceRef annotations:

2.4.2 Using @EJB

A client can obtain a session bean's business interface through dependency injection using the @EJB annotation instead of using JNDI lookups. Java EE application clients refer to enterprise bean instances by annotating static fields with this annotation. For example, the business interface StockQuote of the bean can be obtained using:

```
.....
@EJB
StockQuote stockquote;
double amount = stockquote.getStockQuote("INFY");
.....
```

2.4.3 Using @WebServiceRef

The @WebServiceRef annotation provides a reference to a web service. The reference to a web service can be declared as:

```
.....
@WebServiceRef
(wsdlLocation =
"http://localhost:8080/stockquoteservice/stockquote?wsdl")
.....
```

2.4.4 Using @Resource

The @Resource annotation can be used to declare a reference to any resource that is not annotated using the @EJB and @WebServiceRef annotations.

This annotation allows dependencies to be injected directly into the component [15], avoiding the need for JNDI lookups. The annotation @Resource can decorate a class, a field, or a method. The container will inject the resource referred to by @Resource into the component either at runtime or when the component is initializing, depending on whether field/method injection or class injection is used. With field/method-based injection, the container will inject the resource when the application is initialized. For class-based injection, the resource is looked up by the application at runtime.

The annotation@Resource (javax.annotation. Resource) has the following elements:

name: The JNDI name of the resource.

type: The Java language type of the resource.

authenticationType: The authentication type to use for the resource.

shareable: Indicates whether the resource can be shared.

mappedName: A non-portable, implementation-specific name to which the resource should be mapped.

description: The description of the resource.

Let us look at how Java EE 5 platform has simplified the access to resources using @Resource annotation.

2.4.5 Using @Resource for Data Source

The @Resource annotation can be used instead of using JNDI API to inject a data source into a component that needs to make a connection to a database. For example, if you have a data source named StockQuoteDS, you can obtain a reference to that data source as follows:

```
.....
@Resource
javax.sql.DataSource stockquoteDS;
public Object getStockTicker() {
  Connection con = stockquoteDS.getConnection();
  .....
}
.....
```

The @Resource annotation allows developers to skip the boilerplate code that was previously required to get access to a resource.

2.4.6 Using @Resource for EJB Context

The @Resource annotation can also be used to inject the bean's context--SessionContext and MessageDrivenContext--through a dependency injection.

```
.....
@Resource
javax.ejb.SessionContext ctx;

@Resource
private javax.ejb.MessageDrivenContext mdbCtx;
.....
```

2.4.7 Using @Resource for JavaMail

The @Resource annotation can be used to inject an instance of a JavaMail Session into the application. The Session class represents a mail session, which collects together properties and defaults used by the mail APIs.

```
.....
@Resource
private javax.mail.Session session;
.....
```

2.4.8 Using @Resource for Transactions

The @Resource annotation can be used to inject a UserTransaction into a component that is managing the transactions on its own. The javax.transaction.UserTransaction is an interface to the underlying JTA transaction manager. After obtaining a reference to the UserTransaction resource, a component can invoke the begin, commit, and rollback methods on the UserTransaction object to mark the boundaries of the transaction.

```
.....
@Resource
UserTransaction utx;
.....
try {
  utx.begin();
  .....
  utx.commit();
} catch(Exception err) {
  .....
  utx.rollback();
  .....
}
.....
```

2.4.9 Using @Resource for Timer Service

Timed notifications can be scheduled for all types of enterprise beans (except for stateful session beans) with the help of a timer service provided by the enterprise bean container. EJB components can access to the container-managed timer service using javax.ejb.TimerService. The @Resource annotation can be used to inject a TimerService into an enterprise bean component.

```
.....
@Resource
javax.ejb.TimerService timerService;
.....
timerService.createTimer(100, "Sample");
.....
```

2.4.10 Using @Resource for JMS

The @Resource annotation can be used to inject JMS resource factories, such as connection factories and JMS destinations like Queue or Topic. The dependency of the component on

these resources can be specified using @Resource annotation. The component can inject the connection factory and the destination resources using @Resource annotation as demonstrated below:

```
.....
@Resource(mappedName = "jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(mappedName = "jms/Queue")
private Queue simpleQueue;

@Resource(name="jms/StockTopic", type=javax.jms.Topic)
.....
```

2.4.11 Using @Resource for Environment Entries

Environment entries are configuration parameters used to customize the enterprise bean's business logic. For example, in your application, suppose that you want to give a discount on the final amount for a user who had purchased items for more than $1000. It doesn't make sense to hardcode this into the application, because you might want to change the value of the discount in the future. The annotation @Resource can be used to inject simple environment entries into a field or a method of the bean class as follows.

```
.....
@Resource
double maxDiscount = 0.2;

@Resource
double minDiscount = 0.05;
.....
```

Earlier, this was achieved using the XML tags in the deployment descriptor. However, using the deployment descriptor is the best suitable solution for using environment entries, as it doesn't make sense to have annotations to specify them, because annotations are part of application code.

2.4.12 Using @Resources for Declaring Multiple Resources

The annotation @Resources can be used to inject multiple resources or used to group together multiple @Resource declarations. The following code sample uses the @Resources annotation to group two @Resource declarations. One is a JMS message queue connection factory and the other is a data source.

```
@Resources ({
@Resource(name="stockQueue", type=
      javax.jms.QueueConnectionFactory),
@Resource(name="sqDB", type=java.sql.DataSource)
})
public class StockMDB {
.....
```

3. REFLECTION IN HIBERNATE:

Annotations are compiled into the bytecode and read at runtime using reflection. Using reflection, Hibernate can read the annotations present in the program and take appropriate action. Annotations are present on top of classes and fields, after reading them Hibernate connects to the database server and updates the tables [9]. This part is invisible to the users making the ORM process easy. Normally if Hibernate is not used, then users themselves have to do these things.

3.1. Reflection:

The JVM reads the annotations present in the code and takes action appropriately. But this can only be done for built-in annotations. For custom annotations, we have to specify what action should be taken for a particular annotation. Reflection [3] helps in this regard. Reflection API helps in the introspection of meta-information in the programs. A program that can analyze the capabilities of classes is called reflective. Generally reflection API is used when writing tools such as debuggers, class browsers and GUI builders. With the reflection API we can do the following

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors and superclasses.
- Find out what constants and methods belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.
- Read the annotations present in the program and take appropriate action.

The main classes in this API are java.lang. Class, Field, Method, Constructor of the java.lang.reflect package [13]. They describe the class information, fields, methods, constructors respectively. The interface AnnotatedElement has been added to Java 5. All the before said classes implement this interface. This interface specifies methods to retrieve all the annotations, annotations by type, annotations declared in a particular class and also a method to query whether or not a class contains an annotation of a particular type.

Eg: Suppose an annotation called SampleAnnotation is created and it is present on the class AnnotatedElementTest and on method dummy(), then the following code snippet can be used to read those annotations at runtime.

```
Class<AnnotatedElementTest> c =
AnnotatedElementTest.class;
        System.out.println("Is annotation present:" +

c.isAnnotationPresent(SampleAnnotation.class ));

System.out.println(c.getAnnotation(SampleAnnotation.class))
;
        Method m = c.getMethod("dummy");
        for(Annotation a: m.getAnnotations())
        {
              System.out.println("Annotation:" + a);
         }
```

This code prints all the annotations present on the method dummy and on the class AnnotatedElementTest. Instead of printing something more useful can be done like connecting to the database or producing documentation.

3.2 .NET Attributes:

Similar to annotations in Java, there are attributes in .NET. They are used to represent metadata. .NET supports many built – in attributes. Unlike in Java, attributes follow the normal syntax of a class. Fields are defined normally. Some examples are given below

Eg: A custom attribute called HelpAttribute is made which points to a URL which contains help topics.

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
  public readonly string Url;
  public string Topic {
  get
  {
      return topic;
    }
    set
    {
     topic = value;
    }
  }
  public HelpAttribute(string url)
  {
    this.Url = url;
  }
  private string topic;
}
```

Now this attribute can be used as follows:

```
[HelpAttribute("http://localhost/MyClassInfo")]
class MyClass
{
 }
```

This specifies the help file of 'MyClass'.

4. APPLICATIONS

Usage of annotations in Java EE applications introduces a simpler, POJO-based model and eliminates much of the boilerplate code that is required in earlier versions. Annotations make the deployment descriptors optional][11][16]. Deployment descriptors are typically used when there is a need to overwrite the value specified by the annotations. Also, the greatest advantage of the annotation framework is that it is completely extensible, so future versions of Java EE can expand the existing annotations and introduce new ones.

Few other possible applications of annotations:

- Object-relational mapping - support for persistent objects
  @Entity, @Table, @Id, @OneToMany, etc.
- Web services:
  @WebService, @WebMethod, etc.
- Remote objects (EJB)
  @EJB, @Remote
- Event Handling
  @Action,@ActionListenerFor
- Unit testing
  @Test
- Concurrency
  @ThreadSafe, @GuardedBy
- Static analysis
  @NonNull, @PreCondition

Alloy Annotation Language: The Alloy Annotation Language (AAL)[4] is a language for annotating Java code. This was developed before Java introduced annotations. But this was developed with the same intention and takes advantage of using annotations in source code. AAL, in addition to providing opportunity for generation of runtime assertions, can be used for automatic compile-time analysis. It supports several kinds of analysis such as:

- Checking the code of a method against its specification.
- Checking that the specification of a method in a subclass is
- compatible with the specification in the superclass.
- Checking properties relating method calls on different objects such as that the equals method of a class (and its overridings induce equivalence).
- Using partial models in place of code, it is also possible to analyze object-oriented designs in the abstract: investigating, for example, a view relationship amongst objects.

AAL provides a light weight approach to code annotation. It can be used to test pre and post conditions. AAL is based on simple first-order logic with relational operators, and is thus more in the tradition of semantic data modeling (now called "object modeling"). It can be used to check code against specifications, producing counterexample traces that show how a method's code misbehaves. It can be used to generate test cases from invariants and preconditions fully automatically; this is especially useful for elaborate data structures, such as trees, which cannot be generated randomly because of intricate structural constraints, and are tedious to generate manually. It can also be used for more elaborate checks that are easily expressed in our logic.

AAL includes the following parts. Each method may be annotated with three formulas – a precondition (labeled requires), a post condition (labeled ensures), and a behavior model (labeled does) – and a frame condition (labeled modifies). The precondition, post condition and frame condition together form a specification, which is used to check the code of the method. In the checking of a client, the specification may be used as a surrogate.

Eg: Here we consider the equals method of java.awt. Dimension that is a non - final class from the standard Java library[7]. An object of Dimension class has two integer fields' width and height. AAL can be used to annotate this method.

```
class Dimension
  {
     int width, height;
       // @ does
      // @ {
      // @       \ result = (obj instanceof  Dimension &&
```

```
                    // @          this.widht = obj.width &&
                    // @          this.height = obj.height)
                    // @  }
public boolean equals (Object obj)
{
      if(!( obj instance of Dimension))
          return false;
Dimension d = (Dimension)obj;
Return (d.width == this.width && d.height ==  this.height);
                        }
                }
```

The does annotation specifies a model for the method. Such model can be provided by the programmer or can be generated automatically from the code. A model is a formula over the method parameters, classes and fields.

Annotations need not be used in applications which do not require metadata to be stored or in applications which do not need configuration or mapping. Stand alone applications like games – Chess, MineSweeper etc need not use annotations. But they are useful for client – server, ORM tools, testing frameworks type of applications.

5. CONCLUSION:

Attribute-based programming is an emerging field and many programming languages have only recently added support for attributes / annotations. Java started supporting it in its latest version J2SE 1.5 code named Tiger. .NET calls this feature as attributes. But .NET's attribute feature is advanced when compared to Java's annotations. The former provides many built-in attributes. Annotations are used in many applications and tools. They are also used for configuration. Applications which use Java's annotations are starting to come up. Annotations are an important new language feature in Java, the use of annotations in Java libraries and tools is rapidly increasing, both in standard Java APIs and 3rd party add-ons. A number of JSR groups are working on standard annotations for particular aspects of Java. The J2SE 6 compiler can be extended by adding annotation processors to the class path of the compiler, Annotation processing can also be done at load time, and thus, the power of annotations is limited only by the imagination of the annotation creator. In the future, we can see more and more applications and tools which make use of annotations.

REFERENCES:

[1] Lilian Burdy, Marieke Huisman, and Mariela Pavlova, " Preliminary design of BML: A behavioral interface specification language for Java bytecode", In Fundamental Approaches to Software Engineering, pages 215–229, Braga, Portugal, March 27–30, 2007.

[2] Sun Documentation on Annotations http://java.sun.com/j2se/1.5.0/docs/guide/language/ annotations.html

[3] Sun Documentation on Reflection http://java.sun.com/docs/books/tutorial/reflect

[4] Sarfraz Kurshid, Darkov Morinov, Daniel Jackson, "An Analyzable Annotation Language." ACM November, 2002.

[5] Pominville, Feng Qian, Raja Vallee-Rai, Laurie Hendren, Clark Verbrugge. "A Framework for Optimizing Java Using Attributes. Patrice ACM November 2000.

[6] Tom Mens, Roel Wuyts, Kris De Volder, Kim Mens, "Declarative Meta Programming to Support Software Development: Workshop Report." ACM SIGSOFT March 2003.

[7] Michael Thies, "Annotating Java Libraries in Support of Whole – Program Optimization," ACM 2002.

[8] " Web Tier to Go With Java EE 5: A Look at Resource Injection"

[9] Mats Skoglund and Tobias Wrigstad. 'A mode system for read-only references in Java", In FTfJP'2001: 3rd Workshop on Formal Techniques for Java-like Programs, Glasgow, Scotland, June 18, 2001.

[10] Ma X, Lee H, Bird S, Maeda K, "Models and tools for collaborative annotation," In *Proceedings of 3rd Language Resources and Evaluation Conference (LREC'2002)*, Gran Canaria, Spain.

[11] Cunningham H, Maynard D, Bontcheva K, Tablan V "GATE: A framework and graphical development environment for robust NLP tools and applications",. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*. Philadelphia, US.

[12] Chris Male, David Pearce, Alex Potanin, and Constantine Dymnikov" Java bytecode verification for @NonNull types", In Compiler Construction: 14th International Conference, CC 2008, pages 229–244, Budapest, Hungary, April 3–4, 2008.

[13] Rajiv Mordani, "JSR 250: Common annotations for the Java platform", http://jcp.org/en/jsr/ detail?id=250, May 11, 2006.

[14] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst, "Practical pluggable types for Java. In ISSTA", Proceedings of the 2008 International Symposium on Software Testing and Analysis, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[15] An Oracle White Paper, Introduction to Java Platform, Enterprise Edition 6, April 2010

[16] Trevor Harmon and Raymond Klefstad, "Toward a unified standard for worst-case execution time annotations in real-time Java" In WPDRTS 2007, Fifteenth International Workshop on Parallel and Distributed Real-Time Systems, Long Beach, CA, USA, March 2007.

[17] Joshua Bloch, "JSR 175: A metadata facility for the Java programming language", http://jcp.org/en/ jsr/detail?id=175, September 30, 2004.

[18] William Pugh, "JSR 305: Annotations for software defect detection", http://jcp.org/en/jsr/detail? id=305, August 29, 2006. JSR Review Ballot version.