Attribute-oriented Programming

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
JKU Linz, May 2011
About Myself

- Associate professor at
  - Technical University of Košice, Slovak Republic
  - Faculty of Electrical Engineering and Informatics
  - Department of Computers and Informatics

- Head of Informatics and Computer Languages Laboratory

- Teaching
  - Java Technology
  - Web Technologies
  - Computer Language Processors
  - Model Driven Software Development

- Research
  - Computer Languages
  - Programming Paradigms
  - Software and Language Evolution
Košice, Slovak Republic

- Capitol of Eastern Slovakia
- Second largest city in Slovakia
  - appr. 230 000 inhabitants
- Hosting 2011 IIHF World Championship
- European Capital of Culture 2013
Technical University of Košice

- Founded in 1952
- Universitas Cassoviensis was established in Košice in 1657
- About 13 000 students at 9 faculties
- http://www.tuke.sk
Questionnaire

- What is your **name**?
- Where are you **from**?
- What is your **field of study** (bachelor/master, year of study)?
- What are your primary **educational/research interests**?
- How are you familiar with?
  - Java platform
  - .Net platform/C# language
  - Java reflection
  - Java annotations
The Course

- **Practical** introduction to *Attribute-oriented programming* with examples and case studies
  - Java - main programming language used in examples
  - C# - secondary language for comparison

- **Prerequisites**
  - Object-oriented programming
  - **Java programming**
  - Java reflection principles
1. Attribute-oriented programming, the role of attributes in programming
2. Java platform annotations, .Net platform attributes
3. Runtime annotation reflection
4. Processing annotated source code
5. Case studies and examples: frameworks based on attributes
6. **Case Study: Annotation-based Framework**
Attribute-Oriented Programming

- **Program-level** (model) marking technique

- Developer can annotate program elements with **annotations** describing specific semantics

- Annotations are **structured pieces of information** bounded to program elements (e.g. classes, methods, fields)

- Annotations are **interpreted** in the context of annotated element
Annotation or Attribute?

- **Attribute**
  - Used on .Net platform
  - Attribute is very common (general) term
    - Class members in UML class diagram are attributes and operations
    - Attributed grammars

- **Annotation**
  - Used on Java platform
  - Annotate means “to supply with critical or explanatory notes”
  - More natural name than attribute
Annotation Example

- Java source code

```java
@Author(
    name = "Jaroslav Porubän",
    date = "10/05/2011"
)
public class ExampleClass{
    //...
    public void method() {
    }
}
```
Annotations Usage

- **Development decisions and comments**
  - Development process description (author, task, warning)
  - Patterns instantiation

- **Tool based code transformations**
  - Compiler optimization
  - Code generation

- **Runtime application reflection**
  - Application configuration
Benefits of Annotations

- Information are directly presented on program elements
- **Separate** application's core **logic** from application-specific or domain-specific semantics
  - Compose separated concerns
- **Hide implementation details**
- **Increase** the level of programming abstraction
- **Reduce** programming **complexity**
  - simpler and more readable programs
Working with Annotations

- **Human readable and writable**
  - editable using standard code editor
  - in-code highlighting
  - IDE support for code completion

- **Suitable for machine processing**
  - source code processing
  - runtime processing
Crosscutting Concerns

- Classical programming approaches mixes domain logic with crosscutting concerns
- Domain-specific (application-specific) code is mixed with general application services e.g.
  - Logging
  - Security
  - Transaction processing
  - Exception handling
  - Remote access
  - Caching
public class OrderService {

    public void createOrder(Order order) {
        // verify privileges
        // log call
        // begin transaction if not started
        try {
            // DOMAIN LOGIC
            // log success
            // commit transaction
        } catch (Exception e) {
            // log failure
            // rollback transaction
        }
    }
}

Crosscutting Concerns

domain terms
Separating Concerns using Annotations

- Concerns are specified using annotations

```java
class OrderService {
    @Log(Level.INFO)
    @Security(Role.PRIVILEGED_USER)
    @Transaction(Type.REQUIRED)
    @WebMethod
    public void createOrder(Order order) {
        // DOMAIN LOGIC
    }
}
```
Processing Annotations

- **Compile time**
  - Source code
  - Compiler
  - Compiled code
  - Loader
  - In memory code

- **Loading** time before runtime execution
  - Source code
  - Compiler
  - Compiled code
  - Loader
  - In memory code

- **Runtime**
  - Source code
  - Compiler
  - Compiled code
  - Loader
  - In memory code
  - Reflection
  - Search for
Annotations in Programming Languages

- Annotations on Java platform (Java language)
  ```java
  @FixMe("Incorporate 1.5 features")
  class MyClass {
      // ...
  }
  ```

- Attributes on .Net platform (C# language)
  ```csharp
  [FixMe("Incorporate 3.5 features")]
  class MyClass {
      // ...
  }
  ```
Java Annotations

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
JSR-175 A Metadata Facility for the Java Programming Language

- Defined in Java since version 5
- **Annotations** provide data about a program that is not part of the program itself
- They have no direct effect on the operation of the code they annotate
- Part of the Java Language Specification
- **Weak element binding** with annotation – if the class loader do not find the annotation type then annotations are thrown away
- **Annotation** is instance of an **annotation type**
Java contains general purpose annotation (metadata) facility that permits you to define and use your own annotation types.

The facility consists of:
- a syntax for declaring annotation types
- a syntax for annotating declarations
- a class file representation for annotations
- APIs for reading annotations via reflection
- an annotation processing tool (apt) – integrated into javac
Predefined Java Standard Annotation Types

- 3 annotation types are defined in package `java.lang`
  - @Deprecated
  - @Override
  - @SuppressWarnings

- 5 annotation types are defined in package `javax.annotation`, known as Common Annotations 1.0
  - @Generated
  - @PostConstruct
  - @PreDestroy
  - @Resource, @Resources
Annotation `@Deprecated`

- Indicates that the marked element is **deprecated** and should no longer be used.
- Compilers warn when a deprecated program element is used or overridden in non-deprecated code.

```java
@Deprecated
public void oldMethod() {
    ...
}
```
Annotation @Override

- Indicates that a method declaration is intended to **override** a method declaration in a superclass/superinterface.
- If a method is annotated with this annotation type but does not override a superclass/superinterface method, compilers are required to generate an error message.

```java
@Override
public int hashCode() {
    return 0;
}
```
Annotation @SuppressWarnings

- Indicates that the named compiler warnings should be suppressed in the annotated element
- Every compiler warning belongs to a category
  - deprecation
  - unchecked

```
@SuppressWarnings("unchecked")
public void test() {
    new java.util.ArrayList().add("Hello");
}
```
Creating Custom Annotation Types

- Annotation type declarations are similar to interface declarations.
- Method declarations, called elements, must not have any parameters or a throws clause.
  - Return types are restricted to:
    - primitive types
    - String
    - Class
    - enumeration types
    - annotations
    - arrays of the preceding types (not arrays of arrays)
  - Methods can have default values
Examples of Annotation Types and Annotations

```java
public @interface InProgress {}

@InProgress

public @interface FixMe {
    String value();
}

@FixMe("Update")

public @interface RequestForEnhancement {
    public enum Severity {
        CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION;
    }

    Severity severity() default Severity.IMPORTANT;

    String item();

    String assignedTo() default "Guru";
}

@RequestForEnhancement(item = "Test", assignedTo = "JP")
```
Examples of Annotation Types and Annotations

```java
public @interface ExampleAnnotation {
    //Primitive types
    int intValue();
    double doubleValue();
    boolean booleanValue();
    byte byteValue();

    //String and Class types
    String stringValue();
    Class classValue();
}

@ExampleAnnotation(
    intValue = 123,
    doubleValue = 2.34,
    booleanValue = true,
    byteValue = 127,
    stringValue = "Hello",
    classValue = String.class
)
```
Annotation Type Elements

- If the name of an annotation type element is `value` then you do not need to provide element name in the annotation

```
public @interface FixMe {
    String value();
}
@FixMe("Update")
```

- Every annotation type element without default part should be explicitly defined in annotation

```
public @interface RequestForEnhancement {
    String item();
    String assignedTo() default "Guru";
}
@RequestForEnhancement(item = "core")
```
Meta-annotations in Java SE

- Annotations for annotating annotation types
- Specify behavior of annotation types and annotations
- Defined in package `java.lang.annotation`
  - `@Target`
  - `@Retention`
  - `@Documented`
  - `@Inherited`
Annotation @Target

- Specifies **which program elements can have annotations** of the defined type
- If not present on an annotation type declaration, the declared type may be used on any program element

```java
public enum ElementType {
    TYPE, FIELD, METHOD, PARAMETER,
    CONSTRUCTOR, LOCAL_VARIABLE,
    ANNOTATION_TYPE, PACKAGE
}

@Target({ElementType.PACKAGE,
         ElementType.TYPE})
public @interface InProgress {}
```
Annotation @Retention

- Indicates **how long annotations** with the annotated type **are to be retained**
  - compile time, load time, runtime
- If it is not present on an annotation type declaration, the retention policy defaults to RetentionPolicy.CLASS.

```java
public enum RetentionPolicy {
    SOURCE, CLASS, RUNTIME }

@Retention(RetentionPolicy.SOURCE)
public @interface InProgress {
}
Annotation @Documented

Indicates that the defined annotation should be considered as part of the public API of the annotated program element and should to be documented by javadoc and similar tools by default.

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress {}
Annotation @Inherited

- Intended for use on annotation types that are **targeted at classes**
- Indicates that an annotation type is **automatically inherited**
  - If user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class's superclass will automatically be queried for the annotation type

```java
@Inherited
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress {}```
Duplicate Annotations

- Java **does not support multiple annotations** of the same type on an element
- Common practice is to create additional annotation type with annotation type element of an array type and the name `value`

```java
public @interface Resource {
    ...
}

public @interface Resources {
    Resource[] value();
}

@Resources({@Resource(...), @Resource(...)})
class StudentDao { ... }
```
Not Supported Features

- **null** value is not supported as the annotation element value
- **Annotation type inheritance** is not supported
  - It is just matter of the implementation – authors wanted to have it simple to program
- **General tree structures** are not supported
  - Cyclic annotation element type not supported
- Later on I am expecting changes in the annotation implementation
Java Reflection and Annotations

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
Reflection

- **Reflection** is the ability of a running program to examine itself and its software environment and modify them
  - Introspection – examine itself
  - Intercession – modify itself

- In an object-oriented world, metadata is organized into objects, called **metaobjects** *(e.g. Class in Java)*

---

Reflection – Dynamics in Runtime

- Determine the class of an object
- Get information about a class in the runtime
- 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
Retrieving Class Objects

- Class object is metaobject

Obtaining Class object

Class<?> clazz;

- Using class read-only property
  clazz = Point.class;
- Using getClass() method
  Point point = new Point();
  clazz = point.getClass();
- Using getClass() method
  clazz = Class.forName("reflection.Point");
**Class Objects**

A `Point` object

- X: 23
- Y: 94

Another `Point` object

- X: 10
- Y: 20

A `Class` object

`getClass()`

metaobject characterizes `Point` objects
Creating Class Instances

//Creates instance of a class
String className = "reflection.Point";
Class<?> clazz = Class.forName(className);
System.out.println(clazz.getCanonicalName());

//Using constructor: Point()
Object o = clazz.newInstance();

//Using constructor: Point(int x, int y)
Constructor<?> constructor =
    clazz.getConstructor(Integer.TYPE,
                        Integer.TYPE);
o = constructor.newInstance(2, 3);

Example: CreateInstance.java
Identifying Fields and Methods

- Class object describes
  - Superclass
  - Implemented interfaces
  - Constructors
  - Fields
  - Methods
  - Annotations

Example: DescribeClass.java
Class<?> clazz = Point.class;

//Execute method with reflection
Object point = new Point();
Method method =
    clazz.getMethod("move",
        Integer.TYPE, Integer.TYPE);
method.invoke(point, 4, 5);
System.out.println(point);

Example: ExecuteMethod.java
Reflecting Annotations

- **Interface** `AnnotatedElement` represents an annotated element of the **program currently running in this VM** and allows annotations to be read **reflectively during runtime** (package `java.lang.reflect`)

- Only usable for annotations with **Runtime retention policy**

- **Implementing Classes**:
  - `Class`
  - `Constructor`
  - `Field`
  - `Method`
  - `Package`
public interface AnnotatedElement {
    // Returns true if an annotation for the specified type is present on this element, else false
    boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);

    // Returns this element's annotation for the specified type if such an annotation is present, else null
    <T extends Annotation> T getAnnotation(Class<T> annotationClass);

    // Returns all annotations present on this element
    Annotation[] getAnnotations();

    // Returns all annotations that are directly present on this element
    Annotation[] getDeclaredAnnotations();
}
Reflecting Annotations Example

Author author =
    clazz.getAnnotation(Author.class);
Table table =
    method.getAnnotation(Table.class);

boolean b = clazz.isAnnotationPresent(
    Author.class);

for(Annotation a :
    clazz.getAnnotations()) {
    System.out.println(a);
}
All annotations are instances of `Annotation` interface from `package java.lang.annotation`

Annotation behaves like an object implementing interface defined by annotation type

Runtime implementation is done via dynamic Java proxy
Annotation Processing Tool

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
Annotation Processing Tool

- Annotation Processing Tool
  - Previously standalone tool – apt
  - Now seamlessly integrated into javac

- Provides facilities for annotation processors

- Multiple processors can operate on source code

- Annotation processing happens in a sequence of rounds
  - On each round, a processor may be asked to process a subset of the annotations found on the source and class files produced by a prior round
Annotation Processor

- Plug-in for annotation processing tool
- Processes source code and **looks for annotations**
- Can be used to find annotations of all retention types
- Can also generate source code for the compiler
- API is defined in package `javax.annotation.processing`
- Source code model is defined in package `javax.lang.model`
- Implements `Processor` interface or extends `AbstractProcessor` class
@SupportedAnnotationTypes("annotation.types.FixMe")
public class FixMeProcessor extends AbstractProcessor {

    public boolean process(Set<? extends TypeElement> annotations,
                RoundEnvironment roundEnv) {

        //Iterates over annotation types (annotation.FixMe only)
        for (TypeElement annotationType : annotations) {
            Set<? extends Element> elements =
                roundEnv.getElementsAnnotatedWith(annotationType);

            //Iterates over elements annotated with FixMe
            for (Element element : elements) {
                FixMe annotation = element.getAnnotation(FixMe.class);
                System.out.println(element + ": " + annotation.value());
            }
        }
        return false;
    }
}
Useful Resources in Processor

- **ProcessingEnvironment**
  - Facilities provided by the processing framework to write new files, report error messages, and find other utilities
  - Obtain it from protected field `processingEnv` in class `AbstractProcessor`

- **RoundEnvironment**
  - Query for information about a round of annotation processing
  - Obtain from parameter of the method `process(...)` which is called by the processing tool
Adding Annotation Processor to Compilation Process

- Running processor within compiler
  javac -cp dist\AnnotationTest.jar -processor annotation.FixMeProcessor -proc:only *.java

- Automatic Processor Discovery
  - Just add the full name of processor class into META-INF/services/javax.annotation.processing.Processor file in your processor's jar archive
.Net Attributes

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
Attributes

- Attributes are applied in C# using square brackets

```csharp
[Serializable, Obsolete("Use car!")] public class Horse {
    ...
}
```

```csharp
[Serializable]
[Obsolete("Use car!")] public class Horse {
    ...
}
```
Building Custom Attributes

- .NET attributes are class types that extend the abstract `System.Attribute` base class

```csharp
public sealed class FixMeAttribute : System.Attribute
{
    public string Value { get; set; }

    public FixMeAttribute() {}

    public FixMeAttribute(string value) {
        Value = value;
    }
}
```

```csharp
[FixMe("Incorporate 3.5 features")]
[FixMe(Value = "Incorporate 4.0 features")]
```
Restricting Attribute Usage

- By default, custom attributes can be applied to just about any element of a code
- Restriction are defined using attribute `AttributeUsage` on attribute type definition

```csharp
public enum AttributeTargets
{
    All, Assembly, Class, Constructor,
    Delegate, Enum, Event, Field,
    Interface, Method, Module, Parameter,
    Property, ReturnValue, Struct
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct, AllowMultiple = false, Inherited = false)]
```
Reflection on Attributes

- For reflection use method `MemberInfo.GetCustomAttributes()`

```csharp
Type clazz = typeof(MyClass);
Object[] atts =
    clazz.GetCustomAttributes(
        typeof(FixMeAttribute), false);

foreach (FixMeAttribute att in atts)
{
    System.Console.WriteLine("Fix " +
        clazz.FullName + " : " +
        att.Value);
}
```
Annotation Based Frameworks

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
Examples of Java Mainstream Frameworks Based on Annotations

- Java Persistence API – JPA
- Spring Framework
- Enterprise Java Beans – EJB 3.0
- Java Architecture for XML Binding – JAXB
- Java API for XML Web Services – JAX-WS
Java Persistence API

- Framework managing relational data in applications
- Object-relations mapping

```java
@Entity
@Table(name="SUBJECT")
public class Customer {
    private int id;
    private String name;
    private Collection<Order> orders;

    @Id
    public int getId() {
        return id;
    }

    @Column(name="TITLE")
    public String getName() {
        return name;
    }

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

    ...
```
Spring Framework

- Open source lightweight framework for building Java SE and Java EE applications
- Integration of popular technologies with **middle-tier components** in loosely-coupled style

```java
@Component("orderService")
public class OrderService {
    @Autowired
    @Qualifier("main")
    private CustomerService customerService;

    ...
}
```
Enterprise Java Beans – EJB 3.0

- Managed, server-side component architecture for modular construction of enterprise applications, part of Java Enterprise Edition

```java
@Stateless
@Interceptors(LogInterceptor.class)
public class ProjectService implements ProjectServiceIf {

    @PostConstruct
    private void init() {
        setClass(Project.class);
    }

    @Resource(name = "jms/projectQueue")
    private Queue projectQueue;

    ...
```
Java Architecture for XML Binding

- Allows Java developers to map Java classes to XML documents – XML data serializer

```java
@XmlRootElement(name = "employee")
public class Employee {

    @XmlAttribute
    private int id;

    @XmlElement
    private String firstName;

    @XmlElement
    private String middleName;

    ...
```
Java API for XML Web Services

- Java programming language API for creating web services, part of Java Enterprise Edition

```java
@WebService
public class Hello {
    private String message = "Hello ";

    public void Hello() {};

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```
Examples

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
List of Examples

1. Simple Object-Relation Bridge
2. Logging proxy
3. Task processor
4. YAJCo parser generator
Example 1: Simple Object-Relation Bridge

- We would like to persist objects in tables of relational database
- We would like to query the database obtaining list of objects

```
SELECT
INSERT
UPDATE
DELETE
```
Example 1: Simple Object-Relation Bridge

- **Object to database relation mapping**
  - Class is mapped onto table
  - Field is mapped onto column
  - Object is mapped onto table row

- **DDL** **script should be generated** from the class declaration

- **Rules**
  - The name of table and mapping class may be different
  - The name of column and mapping field may be different
  - At least one field should be marked as identifier
Example 1: Simple Object-Relation Bridge

- **SELECT command**
  
  ```java
  public <T> List<T> select(Class<T> clazz, String condition)
  ```

- **INSERT command**
  
  ```java
  int insert(Object object)
  ```

- **UPDATE command**
  
  ```java
  int update(Object object)
  ```

- **DELETE command**
  
  ```java
  int delete(Object object)
  ```

- **Additional task**
  
  - Update the SORBProcessor to generate PRIMARY KEY in DDL script
  - Add support for @PostConstruct annotation in select method
Example 2: Logging proxy

- We want to **extend** our domain methods with standard logging facility
  - Log call data (name, parameters) at the begging of a call
  - Log call data (name, parameters, result) at the end of a call
    - Success or Failure (if exception is thrown)
  - Some calls are logged and some not, selection is known during development
Example 2: Logging proxy

- The Proxy pattern forces method **calls to an object to occur indirectly** through a proxy object that acts as a surrogate for the other object, delegating method calls to that object.
- If Proxy is used for extension it is called Decorator (Wrapper) Design Pattern.
Example 2: Logging proxy

- The configuration of logging should be very simple
  - one marker annotation
- Logging logic should be general enough to be usable even in other projects
  - it will be perfect if it could be domain independent
- Additional task
  - Create another authorization proxy
  - The required role for method execution is specified using annotation, e.g. `@Security(Role.ADMIN)`
Example 3: Task Processor

- The Minesweeper Game Project
- Student project is built incrementally in a series of tasks according to the instruction manual
- The tasks are divided into 10 modules
  1. Data types
  2. Classes and objects
  3. Interfaces
  4. Interactive programs
  5. Exceptions
  6. Collections
  7. I/O Streams
  8. GUI
  9. Threads
  10. JDBC
Example 3: Task Processor

Cvičenie: 1 2 3 4 5 6 7 8 9 10 11 12

8. Cvičenie - Kolekcie (Minesweeper Module 6)

Cíle
1. Oboznámte sa s významom a druhami kolekcií v jazyku Java.
2. Naučte sa používať implementáciu `ArrayList` rozhrania `List`.
3. Naučte sa používať všeobecné algoritmy pre prácu s kolekciami v triede `Collections`.
4. Pochopite sekvenčný prístup k elementom kolekcie prostredníctvom objektu `Iterator`.
5. Implementovať meranie trvania hry.

Postup
1. Ďalší krok pri implementácii hry Minesweeper je pridanie podpory pre meranie trvania hry.

   **Úloha:** Pridajte do triedy Minesweeper súkromnú členskú premennú `long startTimeMillis`. Táto premenná bude slúžiť na uloženie času začiatku hrania hry. Nastavte danú premennú na začiatku hrania hry použitím metódy `System.currentTimeMillis()`.

   **Úloha:** Pridajte do triedy Minesweeper verejnú metódu `int getPlayingSeconds()`, ktorej návratovou hodnotou je počet sekúnd hrania hry.

   **Poznámka:** Na určenie počtu sekúnd hrania hry použite aktuálny čas (`System.currentTimeMillis()`) a hodnotu premennej predstávajúcej čas začiatka hrania hry `startTimeMillis`. Nezabudnite na to, že dané hodnoty sú v milisekundách.
Example 3: Task Processor

- If you will be a teacher, how will you start the creation of such a case study?
- How to measure students progress in the project every week?
- How to validate students project every weak against the instruction manual?
Example 3: Task Processor

- We are validating student’s project against the teacher’s project
- Task are defined via annotations in the teacher’s project
- It is (partially) general solution applicable even for other projects

[Diagram showing relationships between teacher’s project, student’s project, and structural tasks definition with annotations processed by different processors]
Example 3: Task Processor

```java
@Task(module = "02", id = "getters")
public int getRowCount() {
    return rowCount;
}

/**
 * Returns column count.
 * @return column count
 */
@Task(module = "02", id = "getters")
public int getColumnCount() {
    return columnCount;
}
```
Example 4: YAJCo parser generator

- Main idea is to generate parser (and other language tools) directly from a set of related classes.
- Classes, interfaces and their relationship defines abstract syntax.
- Missing information is added via annotations – multiplicity, lexical units.
Example 4: YAJCo parser generator

Abstract syntax

Concrete syntax

Language Developer
+ YAJCo Parser Generator
Tool Assistance

Concrete Syntax
Pattern Extensions

YAJCo Parser Generator
Example 4: YAJCo parser generator

Language Specification
- Annotated Classes
- Tokens, white characters

Parser Generator - YAJCo

Java Annotation Processor
- generates

JavaCC Input File
- generates

JavaCC

YAJCo Parser Class
- uses

JavaCC Parser Classes

Parser
Conclusions

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
Annotations Are...

- Structured pieces of data directly embedded into source code

- They are always bounded to a program element (e.g. class, method, field)

- Even they directly do not change program semantics, they are interpreted by tools or program themself
Criticism of Annotations

- Annotations are compiled
  - if you want to change configuration, you should rewrite and recompile code

- Annotations can specify only one configuration
  - if there are many different configurations only one can be written directly in annotations

- Annotation approach do not separate development roles
  - Programmer is responsible for code and configuration since both are in the same file
Replacing Annotations with External Files

- We can survive even without annotations
- It is possible to address the program element even outside the code
- Example of addressing program element in XML document
  
  ```xml
  <deprecated>
    <method>
      java.lang.Thread.stop()
    </method>
  </deprecated>
  ```

- **Question:** How to keep synchronized two separate files referencing one another (XML to Java)?
Annotations are Not Suitable for Everything

- Use annotations wisely – they just annotate

```java
public String method(String s) {
    return s.toLowerCase();
}
```

```java
@Public
@Return(String.class)
@Params({
    @Param(name="s", type = String.class)
})
@Body("return s.toLowerCase();")
method() {}
```
Student’s Project

- Create simple annotation based framework using
  - Java Annotation Processor or
  - Java Runtime Annotation Reflection or
  - .Net Runtime Attribute Reflection

- Examples:
  - UI form defined via annotations on entity class
    - Generate form source from entity source
    - Using reflection construct form during runtime
  - Value constraint validation defined via annotations

- Deadline is the **June 10, 2011**
Thank you for your attention