

Uqbar Class Descriptor

Documentación de uso

Versión: 1.1.0

Autor: Leonardo Gassman

Índice

| | |
|---|----|
| Uqbar Class Descriptor | 1 |
| Índice..... | 1 |
| Objetivo..... | 2 |
| Acerca de Uqbar..... | 2 |
| Download..... | 2 |
| Introducción a Uqbar Class Descriptor..... | 2 |
| Metáfora..... | 3 |
| Uso de la clase ClassDescriptor..... | 3 |
| Interface ClassVisitor..... | 4 |
| Detectando casos particulares..... | 5 |
| Escribiendo métodos para annotations de la clase..... | 6 |
| Escribiendo métodos para atributos..... | 7 |
| Escribiendo métodos para annotations de los atributos..... | 7 |
| Escribiendo métodos para los métodos..... | 8 |
| Escribiendo métodos para annotations de un método..... | 9 |
| Escribiendo métodos para annotations de los parámetros de un método..... | 9 |
| Escribiendo métodos para un Constructor..... | 10 |
| Escribiendo métodos para annotations del Constructor..... | 10 |
| Escribiendo métodos para annotations de los parámetros del constructor..... | 11 |
| Escribiendo métodos para los tipos de la clase..... | 12 |
| Escribiendo métodos para valores de las annotations..... | 12 |
| Ignorando métodos..... | 13 |
| Polimorfismo de las annotations..... | 13 |

Objetivo

- El presente documento tiene por objetivo Explicar el uso del framework Uqbar Class Descriptor

Acerca de Uqbar

Uqbar es un grupo de desarrolladores surgido de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires, dentro de la cátedra Técnicas Avanzadas de Programación. El objetivo de Uqbar es desarrollar herramientas de desarrollo que faciliten el trabajo. Los fundadores de Uqbar son, en orden alfabético, Cancinos Claudio, Fernandes Guillermo, Fernandes Javier, Gassman Leonardo, Passerini Nicolás, Picasso Juan Pablo. Al momento de escribir este documento, Uqbar tiene el apoyo de las cátedras de la UTN-FRBA: Técnicas Avanzadas de Programación, Paradigmas de Programación, Arquitecturas de Proyectos IT. También cuenta con el apoyo de Seminarios Athena <http://www.seminariosathena.com.ar/> y de Andina Software SRL <http://www.andinasoftware.com.ar/>

Download

Sitio en sourceforge: <http://sourceforge.net/projects/classdescriptor/>

Repositorio maven 2: <http://andina.no-ip.biz:8080/artifactory/libs-releases>

```
<dependency>
  <groupId>uqbar</groupId>
  <artifactId>uqbar-class-descriptor</artifactId>
  <version>1.1.0</version>
</dependency>
```

Introducción a Uqbar Class Descriptor

A la hora de desarrollar frameworks y componentes de arquitectura con java, es muy común la necesidad de conocer las características de los objetos usando reflection. La forma natural de reflection en java tiene ciertas particularidades que atentan contra la usabilidad, principalmente la ausencia del concepto *tell don't ask*. Para detectar ciertas características de una clase, se pregunta acerca de la clase si cumple con tal condición y entonces se toma una decisión en consecuencia. En problemas simples esto no trae inconvenientes, simplemente con algún que otro *if* se soluciona. Sin embargo atacando problemas de arquitectura, dónde muchos de los componentes que se construyen dependen directamente de las características de las clases, la legibilidad del código se vuelve mala. En arquitecturas que utilizan el concepto de behavioural complete y

declaratividad, este problema toma dimensiones aún mayores. La buena división de responsabilidades se ve comprometida porque se suelen tomar todas las acciones detrás del mismo *if*, mezclando en el mismo método acciones de distinto índole.

El uso intensivo de excepciones chequeadas por parte del API de reflection de java fuerza a que el verdadero código útil quede rodeado de un gran número de líneas para el manejo de las mismas, que pocas veces aportan valor.

Viendo estos problemas, se desarrolló Uqbar Class Descriptor, un componente para tomar acciones según alguna característica particular de una clase. La idea central del mecanismo es *inversion of control* y *tell dont ask*. En lugar de tener que preguntar por si una clase cumple con una condición, se escribe el código que se quiere ejecutar cuando eso ocurra y el Class Descriptor lo invocará automáticamente cuando corresponda.

Otro punto importante de este framework es la forma de indicar los casos que se quiere procesar con cada porción de código. Se sigue una convención de código para escribir los métodos, por lo tanto la interfaz del objeto escrito por el usuario no es fija y puede ser extendida tanto como se necesite.

Una consecuencia que trae, es la posibilidad de usar Annotations de distintos tipos en forma polimórfica, pues se puede escribir un método que será invocado para aquellas annotations (que independientemente de su tipo) tengan algún campo en particular.

Metáfora

La metáfora se basa en el patrón visitor. El director es el Class Descriptor, un componente del framework que es invocado con una clase para inspeccionar y un visitor escrito por el programador, que tiene las acciones a realizar para cada caso que se desee procesar. El director sólo invoca los métodos del visitor que corresponda.

Uso de la clase ClassDescriptor

```
new ClassDescriptor().describe(ClassForDescription.class, visitor);
```

La clase ClassDescriptor es el director del visitor. Esta construida de tal forma que es fácil extender de ella para sobrescribir la forma en que se describe una clase.

Algunos de los casos por el cual uno quisiera escribir una subclase pueden ser:

- Cambiar el orden
- Visitar sólo los métodos declarados en la clase y no todos los públicos,
- Visitar todos los fields y no sólo los declarados en la clase
- Informar acerca de los tipos que implementa indirectamente por su padre.

La forma en que se describe una clase es la siguiente:

1. Superclase
2. Interfaces implementadas (`objectClass.getInterfaces()`)
3. Annotations de clase
4. Valores de las annotations de clase
5. Constructores
 - a. Constructor
 - b. Annotations del constructor
 - c. Valores de las annotations del constructor

-
- d. Annotations de los parámetros del constructor
 - 6. Atributos declarados en la clase (`objectClass.getDeclaredFields()`)
 - a. Atributo
 - b. Annotations del atributo
 - c. Valores de las annotations del atributo
 - 7. Métodos públicos (`objectClass.getMethods()`)
 - a. Método
 - b. Annotations del método
 - c. Valores de los annotations del método
 - d. Annotations de los parámetros del método

Interface **ClassVisitor**

Esta es la interface base para escribir un visitor. Si el usuario escribe una clase que implemente dicha interface, y le pasa un Objeto de este tipo al `ClassDescriptor`, Todos los métodos implementados serán invocados sin problemas. Sin embargo, si se presta atención a la firma del método *describe* de `ClassVisitor`, el tipo del visitor es `Object` y no `ClassVisitor`. Esto se debe a que el usuario puede escribir sólo un visitor con los métodos que le interese, sin necesidad de implementar o extender de ningún otro lado. La única condición que debería cumplir, es que los métodos del visitor tengan la misma firma que los declarados en esa interface. Aquellos programadores que se sientan más cómodos en la robustez del tipado del lenguaje, pueden escribir sus visitors implementándola.

`void` `descriptionStart(Class clazz);`

Se invoca al iniciar el proceso de descripción

`void` `descriptionFinish(Class clazz);`

Se invoca al finalizar el proceso de descripción

`void` `classAnnotation(Class clazz, Annotation annotation);`

Se invoca para informar que la clase tiene esa annotation

`void` `field(Class clazz, Field field);`

Se invoca para informar que la clase tiene ese field

`void` `fieldAnnotation(Class type, Field field, Annotation annotation);`

Se invoca para informar que la clase tiene esa annotation en ese field

`void` `method(Class type, Method method);`

Se invoca para informar que la clase tiene ese método

`void` `methodAnnotation(Class type, Method method, Annotation annotation);`

Se invoca para informar que la clase tiene esa Annotation en ese método

`void` `parameterAnnotation(Class type, Method method, int index, Annotation annotation);`

Se invoca para informar sobre la annotation en un parámetro de un método

```
void parameterAnnotation(Class type, Constructor constructor, int index, Annotation annotation);
```

Se invoca para informar sobre la annotation en un parámetro de un constructor

```
void constructor(Class type, Constructor constructor);
```

Se invoca para informar que una clase tiene ese constructor

```
void constructorAnnotation(Class type, Constructor element, Annotation annotation);
```

Se invoca para informar acerca de una annotation en un constructor

```
void superClass(Class clazz, Class superClass);
```

Se indica la superclase de la clase

```
void type(Class type, Class superType);
```

Se indica que la clase tiene ese tipo

```
void annotationValue(Class clazz, AnnotatedElement element, Annotation annotation, String name, Object value, boolean isDefault);
```

Se invoca para informar sobre el valor de un campo de una annotation, indica también si el valor es un valor por default o fue sobrescrito por programador

Detectando casos particulares

Al escribir un visitor siguiendo la interface ClassVisitor, aún se tiene el problema de detectar casos particulares. Por ejemplo, al recibir una annotation, en el visitor se debería escribir un if para saber el tipo de la annotation y así tomar una acción u otra. Uqbar Class Descriptor soluciona ese problema permitiendo multidispatching. Los métodos anteriormente descriptos son llamados sólo si no se escribió al menos un método más específico para dicho caso. La forma de sobrescribir el método default para un caso particular, depende del método que se esté sobrescribiendo.

Es válido que algún visitor no implemente algún método default

A continuación desarrollaremos todas las variantes. Los casos escritos son una simplificación de los casos que conforman la batería de test de la versión 1.1.0 (disponibles en sourceforge). Todos los casos se basan en detectar ciertas características sobre una clase de prueba.

Innovación:

```
new ClassDescriptor().describe(ClassForDescription.class, visitor);
```

Clase a describir:

```
@AnnotationTest1
@AnnotationTest2
public class ClassForDescription implements Cloneable, Serializable {

    @AnnotationTest1
    private boolean atributel;

    @AnnotationTest2(value1 = "yeah")
    @AnnotationTest1(value1 = "otra cosa", value2 = 1)
```

```
private boolean attribute2;

@AnnotationTest2
private boolean attribute3;

@AnnotationTest1
public ClassForDescription() {
}

@AnnotationTest2
@AnnotationTest1
public ClassForDescription(@AnnotationTest1 boolean param1) {
}

@AnnotationTest2
public ClassForDescription(@AnnotationTest2 boolean param1,
                           @AnnotationTest2 boolean param2) {
}

@AnnotationTest1
public void method1(@AnnotationTest1 @AnnotationTest2 boolean param1,
                   @AnnotationTest2 Integer param2) {
}

@AnnotationTest1(value2 = 2)
@AnnotationTest2
public Integer method2(@AnnotationTest1 @AnnotationTest2
                      @AnnotationTest3 int param1) {
    return new Integer(param1);
}

@AnnotationTest2
public Integer method3() {
    return this.method4(0);
}

// Esto no se tiene en cuenta para ClassVisitorTest porque es un
// visitor que solo trabaja contra la interfaz publica
private Integer method4(@AnnotationTest1 @AnnotationTest2 int param1){
    return 0;
}
}
```

Escribiendo métodos para annotations de la clase

```
public class ClassAnnotationVisitor {

    //metodo default
    public void classAnnotation(Class clazz, Annotation annotation) {
    }

    //metodo por annotation
```

```
public void classAnnotation(Class clazz, AnnotationTest2 annotation){  
}  
  
}
```

Para escribir un método para una annotation particular, el mismo debe tener como primer parámetro un objeto de tipo Class (clase que se describe) y como segundo parámetro un Objeto del tipo de la annotation que se quiere particularizar. En este caso se invocará una vez al método default (classAnnotation con un class y una Annotation) con la annotation AnnotationTest1. Y una vez se invocará al método classAnnotation que recibe un Class y AnnotationTest2.

Escribiendo métodos para atributos

```
public class FieldVisitor {  
  
    //metodo default  
    public void field(Class clazz, Field field) {  
    }  
  
    //metodo por field  
    @Attribute("attribute1")  
    public void fieldAttribute1(Class clazz, Field field) {  
    }  
  
}
```

Para escribir un método para un field en particular, el método debe tener los mismos parámetros que el método default (field), pero estar anotado con la annotation Attribute, e indicarle cual es el atributo para el cual se invoca el método. En este caso, se invocará dos veces al método field (uno con el attribute2, y otro con el attribute3) y una sola vez al método fieldAttribute1, con el field attribute1.

Escribiendo métodos para annotations de los atributos

```
public class FieldAnnotationVisitor {  
  
    //metodo default  
    public void fieldAnnotation(Class type, Field field,  
                                Annotation annotation) {  
  
    }  
  
    //sobreescritura por nombre y annotation  
    @Attribute("attribute2")  
    public void fieldAnnotationAttribute2Annotation1  
        (Class type, Field field, AnnotationTest1 annotation) {  
  
    }  
  
}
```

```
//sobreescritura por nombre
@Attribute("attribute2")
public void fieldAnnotationAttribute2(Class type, Field field,
                                     Annotation annotation) {
}

//sobreescritura por annotation
public void fieldAnnotation2(Class type, Field field,
                             AnnotationTest2 annotation) {
}
}
```

El método `fieldAnnotation` puede ser sobrescrito para un atributo en particular, para una annotation particular, o para la combinación de ambos.

En este caso, el método default (`fieldAnnotation`) será llamado sólo una vez, para el `attribute1` con la annotation `AnnotationTest1`, porque es el único caso que no matchea contra los otros métodos.

El método `fieldAnnotationAttribute2Annotation1`, es invocado sólo una vez, para tratar el caso de la `AnnotationTest1` con el cual se anotó el `attribute2`. Pues se Sobrescribió tanto por atributo como por annotation. Para lo cual se usó la annotation `Attribute` y entre los parámetros del método figura la annotation de tipo `AnnotationTest1`.

El método `fieldAnnotationAttribute2` se le ha indicado que es para el `attribute2`, por lo tanto será invocado 2 veces, una vez con cada annotation. Note la independencia entre este método y el anterior. Ambos métodos serán invocados con el `attribute2` y la `AnnotationTest1`

El tercer caso es la sobreescritura por annotation. Esto se logra escribiendo un método cuyos parámetros sean del tipo `Class`, `Field` y la annotation particular. Entonces el método `fieldAnnotation2` será llamado 2 veces, una con el `attribute2` y otro con el `attribute3`.

Escribiendo métodos para los métodos

```
public class MethodVisitor {

    //metodo default
    public void method(Class type, Method method) {
    }

    //sobreescritura para un metodo sin parametros
    @Message(name = "method3")
    public void methodSpecific(Class clazz, Method method) {
    }

    //sobreescritura para un metodo con parametros
    @Message(name = "method1",
             parameters = {boolean.class, Integer.class})
    public void methodSpecific2(Class clazz, Method method) {
    }
}
```

```
}  
}
```

La sobrescritura por método es utilizando la annotation Mesagge. En este caso el methodSpecific se ejecutará para el method3 y el methodSpecific2 se ejecutará para el method1.

El method2 será tratado por el método default (method), al igual que todos los métodos públicos de Object. El method4 al ser privado será ignorado. Si otro comportamiento se requiere, entonces debe ser utilizado una subclase de ClassVisitor.

Escribiendo métodos para annotations de un método

```
//metodo default  
public void methodAnnotation(Class type, Method method,  
                             Annotation annotation) {  
}  
  
//sobreescritura por nombre y annotation  
@Message(name ="method2", parameters = {int.class})  
public void methodAnnotationMethod2Annotation1  
    (Class type,Method method, AnnotationTest1 annotation) {  
}  
  
//sobreescritura por method  
@Message(name ="method2", parameters = {int.class})  
public void methodAnnotationMethod2(Class type, Method method,  
                                   Annotation annotation) {  
  
//sobreescritura por annotation  
public void methodAnnotation2(Class type, Method method,  
                              AnnotationTest2 annotation) {  
}
```

A igual que los fields, la sobrescritura de las annotations de un método puede ser por método, por annotation y combinandas. Para indicar un método, debe utilizarse la annotation Message, y respetar los tipos del método default. Para sobrescribir por Annotation, debe cambiarse el tipo de la Annotation por el tipo particular. Y combinando ambas cosas se logra la sobrescritura por método y annotation. En este caso se invocará

Escribiendo métodos para annotations de los parámetros de un método

```
public class MethodParameterAnnotationVisitor {  
  
    //metodo por default  
    public void parameterAnnotation(Class type, Method method, int index,  
                                   Annotation annotation) {  
    }  
    //metodo sobrescrito para una annotation
```

```
public void annotationTest3(Class type, Method method, int index,
                          AnnotationTest3 annotation) {
}

//metodo sobrescrito para un method
@Message(name = "method2", parameters={int.class})
public void method2(Class type, Method method, int index,
                  Annotation annotation) {
}

//metodo sobrescrito para un method y una annotation
@Message(name = "method2", parameters={int.class})
public void method2Annotation2(Class type, Method method,
                              int index, AnnotationTest2 annotation){
}
}
```

Los niveles de sobreescritura son iguales a los del método, es decir por method, por annotation o ambos. En esta versión no se soporta la sobreescritura por cada parámetro en particular

Escribiendo métodos para un Constructor

```
public class ConstructorVisitor {

    //metodo por default
    public void constructor(Class type, Constructor constructor) {
    }
    //metodo por constructor
    @Types({boolean.class})
    public void specificConstructor(Class type, Constructor
    }
}
```

La forma de sobrescribir un método para un constructor es usando la annotation Types, indicando cuales son los tipos de los parámetros de ese Constructor. En este caso el Constructor con boolean lo trata el método specificConstructor, mientras que los otros dos Constructores son tratados por constructor

Escribiendo métodos para annotations del Constructor

```
public class ConstructorAnnotationInvokerTestCase {

    // metodo default
    public void constructorAnnotation
        (Class type, Constructor constructor, Annotation annotation) {
    }

    // sobreescritura por nombre y annotation
    @Types( { boolean.class })
    public void constructorAnnotationConstructor2Annotation1
        (Class type, Constructor constructor, AnnotationTest1 annotation){
    }
}
```

```

}

// sobrescritura por nombre
@Types( { boolean.class })
public void constructorAnnotationConstructor2
    (Class type, Constructor constructor, Annotation annotation) {
    }

// sobrescritura por annotation
public void constructorAnnotation2(Class type, Constructor
}

}

```

La sobrescritura para los Constructores sigue la misma filosofía que la de los métodos y fields, se puede sobrescribir para un Constructor usando la Annotation Types y manteniendo los tipos de los parámetros. Para una annotation en particular escribiendo un método con el tipo o combinando ambas

Escribiendo métodos para annotations de los parámetros del constructor

```

public class ConstructorParameterAnnotationVisitor {

    //metodo default
    public void parameterAnnotation(Class type, Constructor constructor,
        int index, Annotation annotation) {
    }

    //sobrescribir por constructor
    @Types({boolean.class, boolean.class})
    public void parameterAnnotationByConstructor (Class type,
        Constructor constructor, int index, Annotation annotation){
    }

    //sobrescribir por Annotation
    public void parameterAnnotationByAnnotation(Class type,
        Constructor constructor, int index, AnnotationTest2 annotation) {
    }

    //sobrescribir por constructor y annotation
    @Types({boolean.class, boolean.class})
    public void parameterAnnotationByConstructorAndAnnotation
        (Class type, Constructor constructor, int index,
        AnnotationTest2 annotation) {
    }

}

```

Igual a las annotations de los parámetros de los métodos, se permite sobrescribir por Annotation, por constructor, o por ambos. Aún no se porta sobrescribir por un

parámetro en particular. La sobrescritura se logra utilizando la annotation Types o escribiendo un método con el tipo particular de la annotation.

Escribiendo métodos para los tipos de la clase

```
public class TypeVisitor {  
  
    //metodo para la superclase  
    public void superClass(Class clazz, Class superClass) {  
    }  
    //metodo default para las interfaces  
    public void type(Class type, Class superType) {  
    }  
    //metodo para un tipo en particular  
    @Type(Cloneable.class)  
    public void typeSpecific(Class type, Class superType) {  
    }  
}
```

El método superClass no tiene sentido ser sobrescrito, pues siempre se puede llamar una sola vez al no existir la herencia múltiple.

El método SpecificType es invocado una vez, pues la clase es Cloneable.

El método default (type) solo se llama para la interface Serializable.

Escribiendo métodos para valores de las annotations

Cualquier método de la forma

```
public void methodName(Class clazz, AnnotatedElement element,  
Annotation annotation, String name, Object value, boolean isDefault) {
```

Puede ser usado para sobrescribir el llamado al valor de un atributo de una annotation.

Las formas de sobrescribirlo son:

- Usando un tipo particular de AnnotatedElement.
- Usando el tipo particular en el parámetro value.
- Usando algunas de las annotations mencionadas anteriormente: Attribute, Message, Type, Types
- Invocando un método sólo en aquellas annotations que tienen un valor que no es el default.

En ese caso, la firma del método no debe incluir el último parámetro boolean:

```
public void methodName(Class clazz, AnnotatedElement element,  
Annotation annotation, String name, Object value) {
```

- Usando la annotation AnnotationAttribute

Ejemplo:

```
@AnnotationAttribute(name = "value1", type=String.class)
```

En este caso, el método será invocado para aquellas annotations que tengan el campo value1 de tipo String.

Cómo limitación, las annotations de parámetros de constructores y métodos no pueden sobrescribirse de esta forma. Deben ser procesadas dentro de los métodos que procesan los parámetros.

Ignorando métodos

```
public class MethodAnnotationVisitor {

    public void methodAnnotationTest(Class type, Method method,
                                     AnnotationTest1 annotation) {
        if(annotationTest.value()) {
            this.methodHelper1(type, method, annotation);
        }
        else {
            this.methodHelper2(type, method, annotation);
        }
    }

    @Ignore
    public void methodHelper1(Class type, Method method,
                              AnnotationTest1 annotation) {
    }
    @Ignore
    public void methodHelper2(Class type, Method method,
                              AnnotationTest1 annotation) {
    }
}
```

En caso de querer escribir un método en el visitor que coincida los parámetros con alguno de los métodos por default, pero no se tiene la intención de que el ClassDescriptor lo tenga en cuenta, se lo puede anotar con Ignore. En este caso ninguno de los metodos helper será llamado directamente por el ClassDescriptor.

Polimorfismo de las annotations

Uno de los mayores problemas que tiene el mecanismo de las annotations de java es que no tienen polimorfismo. A veces por cuestión de comodidad, se quiere anotar un elemento con una u otra annotation, pero ambas tienen un atributo en común, entonces es común ver código como el siguiente.

```
public void method(Class clazz) {
    if(clazz.isAnnotationPresent(AnnotationTest1.class)) {
        this.process(clazz.getAnnotation(AnnotationTest1.class).att());
    }
    else if (clazz.isAnnotationPresent(AnnotationTest2.class)) {
        this.process(clazz.getAnnotation(AnnotationTest2.class).att());
    }
}

public void process(Object value) {
}
```

La annotation `AnnotationAttribute` nos ayuda a emprolijar el código, otorgando la flexibilidad de ignorar el Tipo de la Annotation que contiene el valor

```
public class AnnotationVisitor {  
  
    @AnnotationAttribute(name = "att", type=String.class)  
    public void method(Class clazz, AnnotatedElement element,  
        Annotation annotation, String name, Object value,  
        boolean isDefault){  
  
        this.process(value);  
  
    }  
  
    public void process(Object value) {  
    }  
}
```