2010 International Conference on Availability, Reliability and Security

Investigating the Limitations of Java Annotations for Input Validation

Federico Mancini, Dag Hovland and Khalid A. Mughal
Department of Informatics
University of Bergen
Bergen, Norway
{federico.mancini,dag.hovland,khalid.mughal}@ii.uib.no

***Abstract*—Recently Java annotations have received a lot of attention as a possible way to simplify the usage of various frameworks, ranging from persistence and verification to security. In this paper we discuss our experiences in implementing an annotation framework for input validation purposes. We investigate the advantages and more importantly their limitations in the design of validation tests. We conclude that annotations are a good choice for specifying common validation tests. However, the limitations of annotations have an impact on creating and using generic tests and tests involving multiple properties.**

I. Introduction

Java annotations [1] allow meta-information to be attached to a program in a standard and structured way, such that it is possible to automate its processing. The *reflection* facility in Java makes it possible to retrieve annotations at run-time, and trigger special actions accordingly. This approach allows many aspects of an application to be specified in the application code, eliminating the need for external configuration files (e.g. XML) or inserting extra code in the application. It is not surprising that many frameworks have adopted annotations [2]–[5].

In this paper we focus on the use of annotations to define validation tests for Java object properties and report on our experience in the implementation of a framework for this purpose [6]. Our aim is to give an overview of possible solutions to technical and logical problems that arise with the use of annotations in this context. To begin with, we review systematically some typical issues and standard solutions related to the use of annotations in a framework. For the remainder of the paper we discuss some new challenges that we had to face when providing our framework with new features that were not considered before. In particular, we will show and justify our approach to these problems by critically comparing different solutions we considered. It will become gradually clearer which the current limitations of annotations are, and how far they can be pushed for content validation purposes.

The main discriminating factor we use to distinguish between good and bad solutions is the influence a technical solution has on the usability and flexibility of the framework. We also give higher priority to the extensibility of the framework, i.e., the possibility for the user to define new validation tests in a flexible way and easily reuse the existing ones.

We assume the reader has some familiarity with the basic concepts of annotations and reflection in Java 6 [7], but we start with a short description of our framework and a running example. This should be sufficient to follow the rest of the paper.

II. Working example

In simple terms, content validation in Java means to check that the properties of an object satisfy certain constraints. Annotations can be used to link validation tests to properties. For example, given an object representing the content of a web form for money transfer, we want to make sure that the property representing the amount to be transferred

978-0-7695-3965-2/10 $26.00 © 2010 IEEE
DOI 10.1109/ARES.2010.29

```
public class WebForm{
  private int Amount;
  ...
  @IntRange(min=1,max=100000)
  public int getAmount(){
    return this.Amount;
  } }
```

Fig. 1: *Example of validation annotation.*

is an integer in a certain range. In our framework, this can be done by applying an annotation called `@IntRange` to the method returning such an amount. The corresponding code is shown in Figure 1.

At run-time the annotation and the corresponding return value of the method it tags will be extracted by means of reflection. The value will then be passed for validation to the actual test represented by the annotation. Let us point out that in practice it can be possible to annotate directly a field of an object as they do in [3]:

```
@IntRange(min=1,max=100000)
private int Amount;
```

However we decided not to offer this possibility for a simple reason: if the field is `private`, the framework must first change its visibility to public by means of reflection, before to be able to retrieve its value. We consider it a bad practice to allow an external framework to tamper with the visibility of the application components.

Our framework also offers validation of interdependent properties. In other words, it is possible to validate multiple properties that can be considered valid only if they satisfy some constraint simultaneously. In order to do this, we introduced *cross-annotations*. Cross-annotations can be used to tag two or more properties of the same class, so that they will be validated together. It is possible to give an example by extending the example in Figure 1. If the amount were split into two fields in the web form, i.e., Euro and Cent, and two corresponding properties in the `WebForm` object, then we would need some more elaborated validation. In fact, if the range to check is now between 0.01 and 100 000.00, it would not be enough to check separately the range

```
public class WebForm{
  private int AmountEuro;
  private int AmountCent;
  ...
  @IntRange(min=0,max=100000)
  @CrossRange
  public int getAmountEuro()
   {return this.AmountEuro;}

  @IntRange{min=0,max=99}
  @CrossRange
  public int getAmountCent()
   {return this.AmountCent;}
}
```

Fig. 2: *Example of cross-validation.*

of each field, since 100 000.99 would not be a valid total amount. A further check that involves both properties simultaneously would be necessary. To do this we can define an annotation `@CrossRange`, which can be applied to both these properties as shown in Figure 2. The test corresponding to this annotation checks that both values are greater than 0 and if one is exactly 100 000, then the other one is exactly 0.

To avoid confusion, we call the annotations that are used to validate single properties, as `@IntRange` in Figure 1, *property-annotations*.

III. Lack of inheritance

The first issue when creating annotations for use with a specific framework is: how to distinguish the annotations used by the framework from other annotations at runtime? The problem is that an annotation cannot be extended by other annotations, i.e., there is no inheritance (or subtyping) relationship between annotations.

To resolve this problem there are mainly two approaches:

1) Use a list of predefined annotations, and determine at runtime whether an annotation belongs to this list.
2) Define a meta-annotation that can be used as a marker.

The first approach is convenient if the set of annotations used by the framework is fixed and not supposed to be extended by the user, as in the case

of Struts validation annotations [8]. The latter is convenient when we want to let the user define custom annotations that can extend those defined by the framework, as in our framework and the Hibernate Validator [3], which is based on [9].

We actually use two markers: `@Validation` to define property-annotations and `@CrossValidation` to define cross-annotations. Hence the declaration of `@IntRange` and `@CrossRange` would look like:

```
@Validation
public @interface IntRange{
  int min();
  int max();
}

@CrossValidation
public @interface CrossRange{}
```

IV. User defined annotations

Many problems discussed in this paper arise from the fact that we want to give the user the means to define custom annotations. In fact, this implies that the annotations must have some standard structure so that the framework can process them without knowing their specific usage. One such problem was discussed in the previous section. However, there are more issues that have to be addressed.

For instance, how to define the classes containing the actual validation tests in a standard way? and how to map them to the corresponding annotations? The first problem has an easy solution since we are considering standard Java classes: it is possible to define a common interface they must adhere to:

```
public interface IPropertyTester
 <A extends Annotation,I> {
  public boolean runTest(A an, I o)
  throws ValidationException;
}
```

A class implementing this interface is represented by an annotation of type `A`, that is, in turn, applied to methods returning objects of type `I`:

```
public static class IntRangeTester
 implements IPropertyTester
                <IntRange, Integer>{
  public boolean runTest(IntRange r,
                         Integer v){
   return(
    v >= r.min() && v <= r.max());
} }
```

This interface allows the framework to handle user-defined annotations in a straightforward way, by simply invoking the `runTest()` method and passing the instance of the annotation of type `A` and the return value.

For the problem of mapping tests to annotations, an approach proposed in [9] is to have the tester class as an element of the marker annotation:

```
@Validation(IntRangeTester.class)
public @interface IntRange {
 int min();
 int max();
}
```

This solution implicitly guarantees at compile time that the class associated with the annotation is indeed of the right type, since the element of `@Validation` must be of type `Class<? extends IPropertyTester>`.

In our framework we have tested a different solution. We define the tester class as an inner class of the annotation associated with it. The advantage is that validation annotations are self-contained, and one might even remove the marker annotation and use instead the presence of the inner class to recognize validation annotations. The disadvantage is the loss of compile-time check on the type of the inner class.

Something that cannot be checked at compile-time with any of these approaches, is whether a tester class is mapped to the right annotation, and whether an annotation is applied to the right type of property.

V. Composition

A powerful feature of annotations is the possibility of using meta-annotations, i.e., it is possible to annotate an annotation declaration. The use of a marker annotation like `@Validation` is an example. A more interesting application of this feature is the composition of validation annotations with each other in order to create more elaborated constraints without writing new test classes. For example, assume we have an annotation `@EvenNumber` that

checks whether a number is even. We can compose it with `@IntRange` as follows:

```
@Validation
@IntRange(min=0,max=100)
@EvenNumber
public @interface EvenRange{}
```

The test represented by this new composite annotation is the conjunction of the tests it is composed of. In reality no new validation test is defined: the same result could be obtained by annotating a method with all meta-annotations individually. This is why we extended composition by allowing also boolean operators as meta-annotations. By exploiting the recursive nature of composition, it is possible to create complex boolean expressions starting only from a small number of existing annotations, and therefore really new validation constraints.

As an example of how practical boolean composition can be, we show how to easily create a `@MultipleRange` annotation. We want to check whether an integer value is in a certain set of ranges, not just one. For example, an input is valid if it belongs to one of the following ranges: $[1-10]$, $[20-30]$ or $[60-65]$. We can compose three `@Range` annotations with the special meta-annotation `@BoolOp(OR)`. However, we need to be careful, as the following straightforward implementation would not compile:

```
@Validation
@BoolOp(OR)
@Range(min=1,max=10)
@Range(min=20,max=30)
@Range(min=60,max=65)
public @interface MultipleRange{}
```

In fact, it is not possible to use multiple instances of the same annotations on the same element. Here encapsulation can be helpful. If we first encapsulate each instance of the `@Range` annotation in a new annotation, we can correctly declare:

```
@Validation
@BoolOp(OR)
@Range1_10
@Range20_30
@Range(min=60,max=65)
public @interface MultipleRange{}
```

An alternative solution to multiple annotation instances is provided in [9]. They define a specific annotation which take an array of annotations as element:

```
@Validation
public @interface MultipleRange
 { Range[] list() default {}; }
```

The annotation above can then be used in the following way:

```
@MultipleRange(list=
{@Range(min=1,max=10),
@Range(min=20,max=30),
@Range(min=60,max=65)})
```

The drawback with this solution is that only the conjunction of the tests is possible, and a different annotation container is needed for each annotation type. However, it is not difficult to add a boolean operator as second element, thereby providing more flexibility and making this a preferable solution to encapsulation.

A. *Parameter passing*

The possibility of composing annotations has unfortunately a very strong limitation. Each meta-annotation must have well-defined element values at compile time. In other words, given that we have two annotations `@AtLeast(n)` and `@AtMost(m)`, we cannot define the annotation `@Range(n,m)` as follows:

```
@Validation
@AtLeast(n=Range.min())
@AtMost(m=Range.max())
public @interface Range {
   int min();
   int max();
}
```

It is not possible to pass the value of the elements of an annotation to its meta-annotations. Therefore, we can only create `@Range` annotations for fixed values of `n` and `m`. This might still be useful when there are many properties that need to satisfy a specific range, since the programmer can reuse the same annotation without having to specify the parameters for every instance.

Unfortunately, there does not seem to be any easy workaround for this problem.

VI. Generic elements

Another obstacle in the design of generic validation tests by using annotations, is the limited number of types that can be declared as elements: primitives, `enum` types, `Class`, annotation types, `String` and all corresponding array types.

This prevents the mapping of generic tests to annotations. Ideally it would be easy to create a generic range annotation: For example, the following implementation of a generic range annotation:

```
public @interface
Range <T extends Comparable> {
  Class<T> type() default Object;
  T min();
  T max();
}
```

which represents the generic test:

```
rangeTest(Range r, T o){
 int low=o.compareTo(r.min());
 int up=o.compareTo(r.max());
 return (low>=0 && up<=0);
}
```

Unfortunately, this is not possible as annotation declarations cannot have type parameters.

We identified two possible partial solutions to this problem. The first is to define a specific range annotation for each type we want to test. This is a safe, but not very flexible approach, and quite expensive in terms of the number of annotations to define.

A more generic approach, which is also adopted in [3], exploits the fact that most types representing numerical values have a constructor which accepts a string representation of the value. We can thus define an annotation `@Range(min,max)` where both `min` and `max` are of type `String`. This requires a careful implementation to ensure type safety at runtime, and can easily be misused by the user. A possible improvement is to add an element to the annotation (possibly an `enum`) which allows the user to specify the type of property which is currently being validated.

VII. Cross-annotations

The limitations of the annotation facility became even more apparent when designing cross-annotations in our framework [6]. Keeping the tests as general and reusable as possible became more and more difficult.

In order to access the set of properties that should be validated together, each property has to be marked by the same cross-annotation. The values of the properties marked by the same annotation are then collected into a list, which the user can easily manipulate in a custom validation test. As all values are treated equally, it is usually not possible to distinguish them, i.e., to know which value represents which property.

The advantage is that it is easier for the user to manipulate the list of values, because the structure of this list is independent from the specific object structure. The price for (re)usability in this case, is that mainly only two types of tests can be designed, given the list of return values to validate:

1) Check that a certain amount of these values satisfy some validation constraint, e.g., none, all, at least n and so forth.
2) Check that some combination of the values satisfy the validation constraint, e.g., their sum, product, concatenation, etc.

These types of tests are a useful extension of property-annotations, but they might be insufficient if an application requires a more complex and specific validation logic. For instance, if a property must be validated in a different way according to the value of other properties. This kind of *conditional* validation is not expressible naturally by annotations, and some possible solution would probably involve a text reference to the properties involved. However we discourage all solutions involving a string reference to anything in the code, as any typo might easily cause unnecessary run-time errors.

An alternative solution could be to supply each cross-annotation with a standard element that can be used to give an identifier to each property, or simply store the name of the property. However, we feel that this would completely kill reusability since the implementation of the validation test would depend on the specific association of identifiers to properties of an object, or on the property names.

VIII. Related Work

The experiences reported in this paper came from developing the framework *SHIP Validator*, which is described in [6], and available at [10]. Early discussions of the use of annotations for validation can be found in Holmgren [11] and in Hookom [12]. The ideas in [12] are elaborated in JSR 303 [9], on which the Hibernate Validator [13] is based. Struts 2 [2] also provides validation through annotations. It offers a limited set of standard annotations, with no possibility of creating custom tests.

When it comes to running the actual validation, we are close to the solutions proposed in [9], [12], which allow complete decoupling between validation and application code. In contrast, the solution in Holmgren involves inserting extra code inside the method to be validated.

IX. Conclusions

On one hand annotations lend themselves pretty naturally to link specific validation tests to properties in the code. They are easy to use and create, aid clarity in the code, and provide good type safety. On the other hand, more generic tests are difficult to map to annotations and although this is sometimes feasible and increase reusability, it also require more processing from the framework side to guarantee correctness and avoid run-time errors. In general, there are many things that cannot be guaranteed at compile-time when using annotations. Mostly because of the use of reflection itself, but also because it is difficult to verify that user-defined annotations comply with all the framework specific.

We should keep in mind that annotations are supposed to be just a way to add meta-information to the code, not a programming language inside Java. Therefore, it must not come as a surprise that their expressive power shows some limitations for input validation, especially when more complicated and interdependent constraints must be represented.

What we have tried to show here, is a possible compromise between what is naturally expressible using annotations and what can be achieved by pushing them with the help of more elaborated preprocessing on the framework side. As a general guideline we encourage simpler annotations and tests whose correctness can be guaranteed as much as possible at compile-time, rather than more powerful features that depend on very insecure expedients as string reference to code, and may easily cause run-time errors.

Our conclusion is that annotations can be a very good solution to easily integrate standard input validation in Java applications, without overloading the user and the framework with overly complex implementation. For most validation purposes, our framework is more than adequate. For more complex validation, specific solutions must be devised and some flexibility must be given up. In this case it is probably best to simply code the validation logic directly in the application.

References

[1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Fourth Edition*. Addison-Wesley, 2006.

[2] "Struts," May 2009. [Online]. Available: http://struts.apache.org

[3] "Hibernate validator," Hibernate, September 2009. [Online]. Available: https://www.hibernate.org/412.html

[4] S. W. Chan, "Security annotations and authorization in GlassFish," September 2009. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2EE/security_annotation/

[5] R. Biswas and E. Ort, "The java persistence api," September 2009. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2EE/jpa/

[6] D. Hovland, F. Mancini, and K. A. Mughal, "The ship validator: an annotation-based content-validation framework for java applications," Department of Informatics, University of Bergen, Tech. Rep. 389, September 2009.

[7] "Java 6," Sun, September 2009. [Online]. Available: http://java.sun.com/javase/

[8] "Struts validation annotations," september 2009. [Online]. Available: http://struts.apache.org/2.0.14/docs/annotations.html

[9] E. Bernard and S. Peterson, "JSR 303: Bean validation," Bean Validation Expert Group, March 2009. [Online]. Available: http://jcp.org/aboutJava/communityprocess/pfd/jsr303/index.html

[10] D. Hovland, F. Mancini, and K. A. Mughal, "Ship validator," August 2009. [Online]. Available: http://shipvalidator.sourceforge.net

[11] A. Holmgren, "Using annotations to add validity constraints to javabeans properties," Sun, March 2005. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2SE/constraints/annotations.html

[12] J. Hookom, "Validating objects through metadata," O'Reilly, January 2005. [Online]. Available: http://www.onjava.com/lpt/a/5572

[13] "Hibernate," May 2009. [Online]. Available: https://www.hibernate.org/