REPORTS IN INFORMATICS

ISSN 0333-3590

The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications

Dag Hovland, Federico Mancini and Khalid A. Mughal

REPORT NO 389 **September 2009**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/pdf/2009-389.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.uib.no/ii/en/research/reports-in-informatics.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

1

The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications

Dag Hovland, Federico Mancini and Khalid A. Mughal

Abstract

In this paper we investigate the use of Java annotations for software security purposes. In particular, we implement a framework for content validation where the validation tests are specified by annotations. This approach allows to tag which properties to validate directly in the application code and eliminates the need for external XML configuration files. Furthermore, the testing code is still kept separate from the application code, hence facilitating the creation and reuse of custom tests. The main novelty of this framework consists in the possibility of defining tests for the validation of multiple and interdependent properties. The flexibility and reusability of tests are also improved by allowing composition and boolean expressions.

1 Introduction

The OWASP Top Ten Project [1] lists the lack of proper input validation as the most prevalent cause of critical software vulnerabilities. For this reason, it is important to check that all input satisfies the criteria under which it is safe to execute the program. As an example, take a Java program performing integer division. Integer division by 0 is an illegal operation, resulting in a runtime exception. Hence the value of the divisor should always be validated.

Carefully designing the application can alleviate problems caused by incorrect input. However, this alone will not prevent problems that might arise when a bad input is either passed on to other subsystems like databases, or manipulated and returned to the user.

Standard *input validation* mechanisms should make sure that all input is validated for length, type, syntax, and business rules before accepting the data to be displayed, stored or used [1]. This task can be repetitive and tedious for a programmer, and this is is the primary motive for implementing frameworks for input validation (Commons Validator [2], Struts 2 [3], Hibernate [4], and Heimdall [5]). Such frameworks make it easier to maintain and execute the testing code by decoupling the application logic from the validation logic.

For object-oriented languages like Java, the challenge is to validate specific properties of an object representing the input, without writing validation code in the object itself. Historically, XML configuration files have been used to achieve this separation of concerns, by explicitly storing the names of the properties

*Email: `Dag.Hovland,Federico.Mancini,khalid.mughal}@ii.uib.no`,

Department of Informatics
University of Bergen
PB. 7803
N-5020 Bergen
Norway

to be tested and that of the tests to be performed. At runtime, *reflection* [6] or Servlet filters (listener or interceptors) [3] are then used to actually run the tests on the target methods.

An alternate solution, based on *annotations*, which were introduced in Java 5.0 [6], has gradually emerged. Approaches for input validation based on this new technology are described in [7, 8, 9], and employed, for instance, by Struts 2 [3] and Hibernate [4].

Our approach is inspired by Heimdall [5], but adopts annotations instead of XML configuration, and provides more extensive and powerful tools for the creation of custom validation tests. The reasons to prefer annotations over XML configuration files have been well motivated in [8], and here we show in practice how far annotations can be pushed for input validation purposes. Although some technical solutions we use are also found in [7, 8, 9], we offer a simpler and more powerful way of creating custom tests, with focus on reusability. Furthermore, we propose a way of defining validation constraints over multiple properties of an object simultaneously, rather than just single properties. This allows the user to validate the relationship between interdependent properties, which, to our knowledge, is not possible with any other validation framework based on annotations or XML. For this purpose we distinguish between *property-tests* and *cross-tests*. A property-test is used for the validation of a single object property (for instance, JavaBean properties accessible through getter-methods), whereas a cross-test is concerned with constraints involving multiple properties.

The next section shows a simple example of how annotations can be used for validation. This running example is gradually extended to show more advanced features of our framework. A more formal description of how new annotations can be created and used is given in Section 3. The second part of this article contains the implementation details of the current version of the framework. Finally we compare our work to the other framework we mentioned previously in this section, and draw some conclusions. The latest version of its implementation is available from [10].

2 A Running Example

In this section we introduce a running example used throughout the paper, and show how annotations can be used to define tests on single properties of an object.

We will use the web form for international money transfers from a hypothetical Internet bank (see Figure 1). *IBAN* (International Bank Account Number) is the standard for identifying bank accounts internationally. Some countries have not adopted this standard, and for money transfer to these countries, a special *clearing code* is needed in combination with the normal account number of the beneficiary. *BIC* (Bank Identifier Code), also known as SWIFT, is needed to identify the beneficiary's bank uniquely.

We assume that the object representing the form is created in Java, and that each field in the web form is represented by a property of this object. Fields where the user does not enter a value, are in this example represented by the `null` value. A partial implementation of this Java object is shown in Figure 1. Here every annotation represents a test to be run on the return value of the method it is applied to. In our framework, annotations representing tests are called *validation-annotations*. This categorization is further split into *property-annotations*, which represent property-tests, and *cross-annotations*, which represent cross-tests. All the annotations in Figure 1 are property-annotations, i.e., they involve checking a single property.

We use property-tests to check whether basic formatting rules are respected. For example, the annotation `@IntRange(min=0,max=10000)` indicates whether the value of `amountEuro` is non-negative and not greater than 10000. The property-annotation `@IntRange(min=0,max=99)` represents a test to check whether `amountCents` is between 0 and 99. The property-annotation `@ValidateBIC` represents a property-test for BIC codes, and `@Required` means that the field cannot be left empty.

IBAN

BIC BICCODE

Account

Clearing-code AB1232342

Amount € 10000 . 10 c

Pay international bill

Figure 1: *Web form for international bank transfer.*

EXAMPLE OF ANNOTATED CODE

```
@ValidateBIC
@Required
public String getBIC() {
 return BIC;
}

@IntRange(min=0,max=10000)
public Integer getAmountEuro() {
 return amountEuro;
}

@IntRange(min=0,max=99)
public Integer getAmountCents() {
 return amountCents;
}
```

Table 1: *Example code using the property-annotations to test the input from the web form in Figure 1.*

<table>
<tr>
<th>INTERFACE OF THE INNER CLASS OF A PROPERTY ANNOTATION</th>
<th>DECLARATION OF @IntRange</th>
</tr>
<tr>
<td>

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

</td>
<td>

```
@Validation
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE,
         ElementType.METHOD})
public @interface IntRange <IntRange,
                            Integer> {
 int min();
 int max();
 public static class Tester
 implements IPropertyTester{
  public boolean runTest(IntRange r,
                         Integer v) {
   return(v >= r.min() && v <= r.max());
  } } }
```

</td>
</tr>
</table>

Table 2: *Example of a correct property-annotation declaration.*

The annotations only specify what tests should be run on each value. To actually run the tests, the `WebForm` object must be passed to a validator. The validator inspects the object through reflection, extracts the annotations and the return values from the getter-methods, and invokes the corresponding tests. This process is discussed in details in Section 4.

3 Validation annotations and tests

In this section we discuss the reasons for distinguishing between property-tests and cross-tests, and provide details of how they are implemented and used.

3.1 Property-annotations and Property-tests

Declaring a property-annotation is fairly straightforward. As an example we use the declaration of the annotation `@IntRange` shown in Figure 2.

A fundamental part of the declaration is the meta-annotation `@Validation`, which works as a marker. Without it, our framework would not be able to distinguish a property-annotation from other annotations. There are other solutions to this problem, but such *marker-annotation*s are a standard way to compensate for the lack of inheritance in annotations [8, 7, 9].

The `@Retention(RetentionPolicy.RUNTIME)` meta-annotation must be present such that the property-annotation is accessible at runtime. The annotation `@Target` has the usual meaning, but we use both `ElementType.ANNOTATION_TYPE` besides `ElementType.METHOD` to limit the use of some special annotations. The annotation declaration itself is fairly standard and can be annotated with any number of other annotations.

Finally, we require a public *inner class* which must contain the code of the property-test associated with this property-annotation. This class must implement the interface `IPropertyTester` shown in Table 2

in order to ensure that it provides the implementation of the method `runTest()`, which is invoked by the framework to run the test. Another possible approach for associating a test to an annotation is explained in [7].

The test corresponding to the property-annotation `@IntRange` is defined in the inner class `Tester`, as shown in Figure 2. The method `runTest()` is called by reflection and takes as parameters an instance of the annotation and the object to test (that is, the return value of the method). We allow only one inner class implementing `IPropertyTester` in the annotation declaration.

3.1.1 Handling `null` values

Many validation frameworks provide an annotation `@Required` which indicates that a certain property should not be `null` [3, 4, 5]. However, no annotation seems to be provided to specify when a property *can* be `null`.

To understand why this might be useful, let us assume that we allowed the field BIC in Figure 1 to be left empty by the user, i.e., the method `getBIC()` in Table 1 was annotated with `@NotRequired` instead. This means that if return value of `getBIC()` is `null`, no `NullPointerException` shpuld be thrown during the validation process. To achieve this, either the test represented by `@ValidateBIC` must be able to correctly handle a `null` value in this situation, or the framework should prevent any test to be run when `BIC` has the value `null`. In the first case the burden of treating this special case is left to the programmer, who must consider the possibility that any test he or she designs might be run on a `null` value. However, tests are supposed to be reusable and cannot account for all possible ways of treating a `null` value in different situations. The same problem arises when, in the absence of a `@Required` annotation, the framework should decide how to interpret a `null` value, at the risk of masking a possible error or causing one.

To avoid these problems, we provide the `@NotRequired` annotation, which can be used to specify that a `null` return value is valid, and that in this case no other tests should be run on the value. If neither a `@Required` nor a `@NotRequired` annotation is specified, the framework will simply run the other tests on the method, even if the return value is `null`. Therefore, by default, our framework does not give any special treatment to `null` values, and the programmer can design reusable tests, by handling a `null` value in an independent way. In this setting, any `NullPointerException` will properly signal a programming error.

3.2 Cross-annotations

Recall the specifications of international bank transfers mentioned in Section 2. All transfers require the BIC code of the receiving bank, and in addition either the IBAN or both clearing code and the account number. This means that there is a mutual dependency between some fields of the web form. Therefore, in order to check such constraints in the corresponding object, it is not enough to consider the return values of the involved methods independently. For this purpose we introduce a new type of validation-annotation which we have called cross-annotations. These allow a programmer to create tests involving multiple properties of an object, i.e., cross-tests.

In Figure 3 we extend the example in Figure 1 to show how it is possible to annotate the `WebForm` object in order to enforce the constraints mentioned earlier. Each cross-test is represented by a cross-annotation, which is applied to all methods whose return values are involved in the test. All annotations in the example are cross-annotations with the exception of `@Required` and `@NotRequired`. The property-annotations from Figure 1 are not shown in order to keep the example simple.

EXAMPLE OF USAGE OF CROSS-ANNOTATIONS

```
@Required
public String getBIC() { return BIC; }

@ExactlyOneNull
@NotRequired
public String getIBAN() { return IBAN; }

@ExactlyOneNull
@AllOrNoneNull
@NotRequired
public String getAccount() {
 return account;
}

@AllOrNoneNull
@NotRequired
public String getClearingCode() {
 return clearingCode;
}
```

Table 3: *Code annotated with cross-annotations.*

The cross-test represented by `@ExactlyOneNull`, which is applied to the return values of the methods `getIBAN()` and `getClearingCode()`, ensures that exactly one of them has not been filled in the web form. Furthermore, the cross-test represented by `@AllOrNoneNull` makes sure that either all or none of the methods marked with it return `null`. Thus, we are able to check that either the IBAN is used, or both the account number and the clearing-code are specified, but not all three.

Cross-annotations can be declared in almost the same way as property-annotations, as shown in Table 4. Only the marker-annotation, `@CrossValidation`, and the interface of the inner test class, shown in Figure 4, are different.

As can be seen in Figure 4, a cross-test takes as parameter the corresponding cross-annotation and all the return values involved in the test as a single `List`. This means that the return values are not differentiated according to what method they come from, hence limiting the type of cross-tests that can be developed in the current framework. These limitations are discussed in the next section.

3.3 Boolean composition

Another novelty of our approach is that we can combine validation-annotations with boolean operators in order to create new validation-annotations. These *composed* annotations can be created by declaring a new validation-annotation which is annotated with the validation-annotations we want to compose. In addition, the special meta-annotation `@BoolTest` can also be used in the composition. Its single element is of type `public enum BoolType{OR, AND, ALLFALSE}`, with the usual semantics. By default, specifying a list of annotations without the `@BoolTest` annotation represents the conjunction of the corresponding

INTERFACE FOR THE INNER CLASS OF A CROSS-ANNOTATION

```
public interface ICrossTester<A extends Annotation, V> {
 public boolean runTest(A a, List<V> v)
                              throws ValidationException;}
```

EXAMPLE OF CROSS-ANNOTATION

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})
@CrossValidation
public @interface AllOrNoneNull {
 public static class Tester implements
 ICrossTester<AllOrNoneNull,String> {
  public boolean runTest(AllOrNoneNull c, List<String> v) {
   ... } }
```

Table 4: *Example of a correct cross-annotation declaration.*

DECLARATION OF `@ValidateBIC`

```
@Validation
@BoolTest(BoolType.AND)
@PatMatch("\\w{8}|\\w{11}")
@AdditionalTest
public @interface ValidateBIC{}
```

Table 5: *Definition of the annotation* `@ValidateBIC`*, used in Figure 1.*

tests, thus `BoolType.AND` is not strictly necessary.

Figure 5 shows the declaration of the annotation `@ValidateBIC` which we first introduced in Figure 1. This annotation is created by composing `@PatMatch("\\w{8}|\\w{11}")`, which is a common annotation for string-matching tests, and the one represented by `@AdditionalTest`, which represents some other possible test that we do not specify here. Since the annotation `@BoolTest(BoolType.AND)` is also specified, the test represented by `@ValidateBIC` will succeed only if *both* the tests represented by the two other property-annotations succeed.

Boolean composition can also be applied with cross-annotations. For example, given the web form above, we want to check that the overall amount transferred is greater than 0.00, but not greater than 10000.00, we can use the cross-annotation `@AmountCheck` as shown in Table 6, since the fields for Euros and cents are represented as different properties in the `WebForm` object.

In Table 6, we see that `@AmountCheck` is a composition of two other cross-annotations: `@SumMin(1)` and `@MaxAmount`. The first represents a test checking that the sum of `amountEuro` and `amountCents` is greater than 0. The second is in turn a composed cross-annotation, which by checking that either one of the two values is smaller than 1, or both are smaller than 10000, can guarantee that the total amount they represent together is at most 10000. This can be represented as the following formula: $(\text{Euros} + \text{Cents} \geq 1) \wedge ((\text{Euros} < 1 \vee \text{Cents} < 1) \vee (\text{Euros} < 10000 \wedge \text{Cents} < 10000))$. Besides, it is clear that the order

<table>
<tr><td rowspan="4">LOGICAL STRUCTURE OF <code>@AmountCheck</code>

AND
SUM   OR
>1   ALL   ATLEAST(1)
<10000   <1</td><td>DECLARATION OF <code>@MaxAmount</code></td></tr>
<tr><td><code>@CrossValidation</code>
<code>@BoolTest(BoolType.OR)</code>
<code>@OneLessThan(1)</code>
<code>@AllLessThan(10000)</code>
<code>public @interface MaxAmount {}</code></td></tr>
<tr><td>DECLARATION OF <code>@AmountCheck</code></td></tr>
<tr><td><code>@CrossValidation</code>
<code>@BoolTest(BoolType.AND)</code>
<code>@SumMin(1)</code>
<code>@MaxAmount</code>
<code>public @interface AmountCheck {}</code></td></tr>
<tr><td colspan="2">USAGE OF <code>@AmountCheck</code> IN THE CODE</td></tr>
<tr><td><code>@AmountCheck</code>
<code>public Integer getAmountEuro() {</code>
<code> return amountEuro;}</code></td><td><code>@AmountCheck</code>
<code>public Integer getAmountCents(){</code>
<code> return amountCents;}</code></td></tr>
</table>

Table 6: *Example of composition with cross-annotations.*

of Cents and Euros in the formula does not matter. In each subexpression, the same property must hold for both. This means that we can always express this constraint by our operators as shown always in Table 6.

Now that we introduced composition, it might become clearer why we imposed the limitations discussed in the previous section on cross-tests. If we allowed elements in cross-annotations which could be associated to a particular return value, for instance to identify the method they came from, composition would become more involved.

However, it is allowed to use element parameters to configure the test represented by the annotation, as in `@SumMin`. Furthermore, since such a parameter should be the same for each instance of the cross-annotation appearing in the object, it is good practice to encapsulate the annotation with the specific element value into a new cross-annotation without any elements.

Using composition to encapsulate an annotation with specific argument values into one without parameters, can also ease maintenance or/and refactoring. Also, boolean composition makes it particularly easy to create validation tests based on white- and blacklisting.

Table 7: *Examples of each of the four types of tests.*

	Basic Tests	Composed Tests
Property-Tests	`@IntRange`	`@ValidateBIC`
Cross-Tests	`@AllOrNoneNull`	`@AmountCheck`

3.4 Recursive validation

In some cases, the return value of a method can be an object that needs special validation itself. In this situation, the kind of tests that we have presented until now, might be too generic for an efficient validation. On the other hand, we would like to avoid ad-hoc validation tests as much as possible, to preserve reusability. For this purpose we define the special annotation `@Valid`, which validates an object according to the tests defined in its own class declaration. In other words, we recursively run a validator on a return value, before testing the rest of the object that is currently being validated.

A simple example might be to replace the methods `getAmountEuro()` and `getAmountCents()` in the class `WebForm`, with a method `getTotalAmount()` which returns an object of type `Amount` and was annotated with `@Valid` as shown in Table 8.

NEW METHOD OF THE CLASS `WebForm`	NEW CLASS TO VALIDATE RECURSIVELY

`@Valid`
`public Amount getTotalAmount {`
`  return this.totalAmount;`
`}`
	`public class Amount{`
	`  ....`
	`  @AmountCheck`
	`  public Integer getAmountEuro() {`
	`    return amountEuro;}`
	
	`  @AmountCheck`
	`  public Integer getAmountCents(){`
	`    return amountCents;}}`

Table 8: *Example of recursive validation.*

3.5 Composing property-tests into cross-tests

As seen above, cross-tests are given a list of values of the same type, and the test cannot distinguish between them. Many cross-tests therefore consist either in counting how many of the values pass a given test, or in applying some operation to all the values (e.g., a sum) and check whether the result passes a test. It seems natural to reuse property-annotations to create these tests.

The annotations `@CrossProperty` and `@CrossOperator` can be used to construct cross-tests from property-tests. The semantics of each annotation is as follows:

- `@CrossProperty`: it takes one of the operators shown in Table 9 and an integer. These two parameters together define how many return values must satisfy the given property-tests. For the operators `ALL` and `NONE`, however, no integer is required.
- `@CrossOperator`: it takes a class defining an associative operator as argument (e.g., sum). This operator is applied to the list of return values associated with the cross-annotation, and the resulting value is checked against the given property-tests.

Table 10 contains examples of how `@CrossProperty` and `@CrossOperator` can be used to give an alternative implementation of the cross-annotations: `@AllNull`, `@OneLessThan(1)` and `@SumMin` seen in Table 6. Unfortunately, the cross-tests composed from property-test cannot take arguments, so the

Table 9: *Operators that can be used to compose property-annotations into cross annotations*

Annotation	Argument
`@CrossProperty`	{`ALL,NONE,ATLEAST,ATMOST,EXACTLY`}, `int n`
`@CrossOperator`	`Class <? extends ICrossOperator >`

new test is called `@OneLessThan1`. By using this alternative implementation, the annotation `@AmountCheck` defined in Table 6 can be created without writing any inner class containing the actual test.

DECLARATION OF `@AllNull`

`@Retention(RetentionPolicy.RUNTIME)`
`@CrossValidation`
`@AllProperty`
`@IsNull`
`public @interface AllNull {}`

DECLARATION OF `@OneLessThan1`

`@CrossValidation`
`@Retention(RetentionPolicy.RUNTIME)`
`@CrossProperty(operator=PropertyOperator.ATLEAST,n=1)`
`@IntUpperBound(0)`
`public @interface OneLessThan1 {}`

INTERFACE FOR CLASSES USED AS ARGUMENTS OF `@CrossOperator`

`public interface ICrossOperator <A> {`
`  public A op(A l, A r);`
`}`

DECLARATION OF THE CLASS `Sum`

`class Sum implements ICrossOperator<Integer>{`
`  public Integer op(Integer l, Integer r){`
`   return l+r;`
`  }`
`}`

DECLARATION OF `@SumMin`

`@Retention(RetentionPolicy.RUNTIME)`
`@CrossValidation`
`@CrossOperator(Sum.class)`
`@IntLowerBound(1)`
`public @interface SumMin {}`

Table 10: *The code showing how to reimplement some cross-annotations composing `@AmountCheck`.*

4 Implementation details

4.1 Packages

The hierarchy of packages comprising the framework is shown in Figure 2.

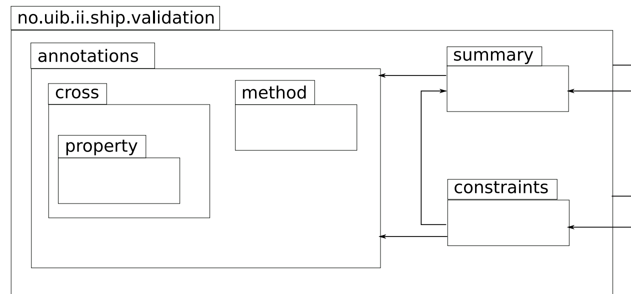


Figure 2: *Package hierarchy.*

4.1.1 Package `validation`

This is the main package which contains the following classes and interfaces:

- `IValidatorFactory`: interface used to define a factory creating `Validator` objects.
- `ValidatorFactory`: the (currently) unique factory implementing `IValidatorFactory`.
- `IValidator`: the interface defining the objects returned by an `IValidatorFactory`.
- `Validator`: the class of the validator objects used by the application.
- `ValidationException`: the class defining validation exceptions that can occur in the framework.

All other packages are subpackages of the `validation` package.

4.1.2 Package `constraints`

The classes contained in this subpackage implement the tools needed to extract validation annotations from the application code. They create the corresponding annotation objects that associate tests and return values, run the actual validations tests and create a summary of the outcome. The main class of the package is `AnnotationObject`. It is an abstract class that defines a common behaviour for most objects in this package by giving a standard implementation for the majority of its methods. The reason is that the main steps in the validation algorithm are the same for all types of annotations, hence we only need to plug-in specialized implementations of particular methods for each type. In other words we use the *Strategy Pattern* [11]. In particular, for each type of validation-annotation, i.e., property-annotations, cross-annotations, and cross-annotations created from property-annotations, we need specialized methods for creating the tree structure of the boolean composition and to run the corresponding tests. Figure 3 shows the class diagram of the package.

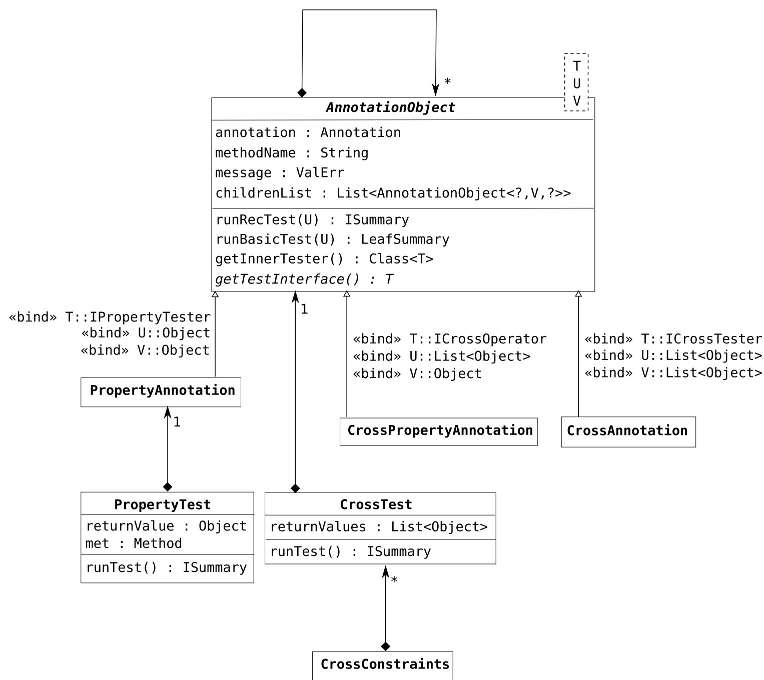


Figure 3: *Relation between some of the classes in the package no.uib.ii.ship.validation.constraints.*

Every class that extends that superclass `AnnotationObject`, that is, the three subclasses `PropertyAnnotation`, `CrossAnnotation` and `CrossPropertyAnnotation`, contains a field to store the instance of the annotation they represent and a list of the `AnnotationObject`s encapsulated by this annotation. The type parameters `T`, `U` and `V`, represent respectively: the interface implemented by the inner class of the annotation boxed in the object, e.g., `IPropertyTester` for `PropertyAnnotation` objects; and the type of the parameters accepted by the `runTest()` method of such an inner class. This allows us to retrieve the inner class of the annotations in a generic way by reflection, so that the method `getInnerTester()` needs to be implemented only in the abstract class `AnnotationObject` and can be used by all its subclasses.

The other methods defined in the class `AnnotationObject` need to be specialized by the subclasses, and can be divided into two types: those that create the list of children of an annotation and those that run the tests.

The methods `runRecTest()` and `runBasicTest()` are used to actually test the return values. In particular, `runRecTest()` is only used recursively on the annotation tree and to build the summary tree according to the boolean composition of the partial tests, while `runBasicTest()` is the method that actually invokes the inner test of an annotation on the given return value to check whether it passes the test or not.

The other two classes found in this package, `PropertyTest` and `CrossTest`, have the following tasks in common:

1. Collect the validation annotations on a method;

2. Create an AnnotationObject which will contain the tree of all these annotations (where the AnnotationObject at the root does not represent any real annotation and its annotation field is, therefore, set to null);

3. Retrieve the return value of the method by reflection;

4. Pass the return value to the AnnotationObject when invoking the runTest() method.

The difference between the two classes is that, given an object to validate, the Validator creates a new PropertyTest object for each method to validate, but only one CrossTest to store all cross-tests. The reason is simply that we do not know all the methods involved in a cross-test until all methods of the object have been checked by reflection.

An example of an IPropertyTest object is given by the object diagram in Figure 4. The object represents the property-annotations on the method getBIC() shown in Figure 1. The recursive structure of the property-annotations is preserved by the tree structure of the object. As a side remark, this is the IPropertyTest object that is created when the return value of the getBIC() method is not null. Otherwise, the annotation @Required would be the only one appearing in the object.

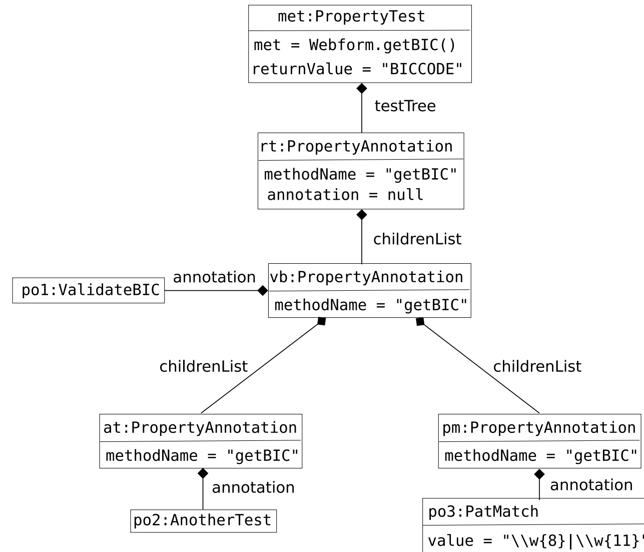


Figure 4: *Example of an IPropertyTest object.*

4.1.3 Package **summary**

This package contains the classes used to create a summary of the validation test results. The summary is structured as a tree that mimics the recursive boolean composition of the tests, and every class in this package represents a different type of node in this tree, specialized to store the outcome of a specific type of test.

The *composite pattern*[11] is used to implement the functionality of this package. All classes implement the interface ISummary, since they all need to be able to print out and return the test results they store.

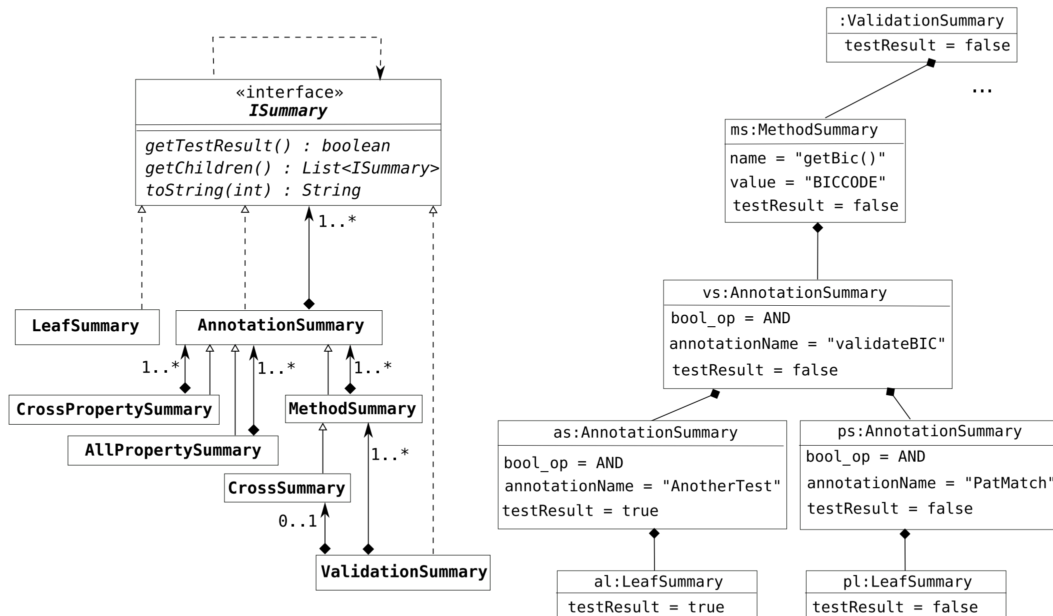


Figure 5: *Class diagrams of the package* `summary`*, and example of* `MethodSummary` *object.*

However some of them, i.e., the leaves of the tree which are of type `LeafSummary`, do not need to list or add children to themselves, so that the corresponding methods are left empty even though they are defined in the interface. All others classes are extensions of the class `AnnotationSummary`. This approach allows us to create and query a summary in a uniform manner.

The correspondence between a class of this package and the kind of test result it contains is quite straightforward. A `ValidationSummary` object contains the summary for a whole object that was validated, and it consists of:

1. A `MethodSummary` object for each method annotated with property-annotations.

2. One `CrossSummary` object which contains the results of all cross-tests. It has the same structure as a `MethodSummary`, but with the difference that a `CrossSummary` object does not represent any method, and therefore the related field is initialized differently.

The summary can also contain other `ValidationSummary` objects in case of recursive validation, i.e., if a method is annotated with `@Valid` (see Section 3.4).

A class diagram of the package is shown in Figure 5, together with an object diagram representing the results of the validation test on the return value of the `getBic()` method with the input shown in Figure 1.

4.1.4 Package `annotations`

The package `annotations` contains all annotations that can be used to create new validation-annotations, and the interfaces that the inner class of each annotation type must implement. These annotations are listed in Table 11. The subpackages contain the "ready-to-use" validation-annotations provided with the framework. They are organized in property-annotations (Table 12 and 13), cross-annotations (Table 14) and

cross-annotations created by composing property annotations. Since these last ones differ from the annotations in Table 14 only in their implementation, and the fact that they cannot take parameters, we do not list them here.

no.ii.uib.ship.validation.annotations			
	ANNOTATION	ELEMENTS	USAGE
	`@Validation`	–	See Section3.1
	`@CrossValidation`	–	See Section 3.2
COMP	`@BoolTest`	`BoolType value`	See Section 3.3
	`@CrossOperator`	`Class` <? extends `ICrossOperator` >	See Section3.5
	`@CrossProperty`	`PropertyOperator operator`	See Section3.5
	`@AllProperty`	`Boolean value`	See Section3.5

Table 11: *Annotations used as markers and to compose other annotations.*

no.ii.uib.ship.validation.annotations.method			
	ANNOTATION	ELEMENTS	USAGE
NULL	`@IsNull`	–	Represents a test that succeeds only if the return value it is applied to is `null`
	`@Required`	–	See Section 3.1.1
	`@NotRequired`	–	See Section 3.1.1
	`@NullTest`	`boolean value`	Represents a test that succeeds only if the return value it is applied to is not `null` or if it is `null` and `value==true`
XML	`@SchemaNode`	`String file` `String url`	Checks an XML document against a XML Schema gave as a node, which can be found either at the location specified in `file` or at the URL specified in `url`
	`@SchemaString`	`String file` `String url`	As the previous one, but the XML schema is given as a string
	`@PatMatch`	`String value`	To match a `String` against the given regular expression
	`@Valid`	–	See Section 8

Table 12: *Annotations used for single methods, i.e., property-annotations.*

no.ii.uib.ship.validation.annotations.method.range			
	ANNOTATION	ELEMENTS	USAGE
RANGE ANNOTATIONS	@DoubleLowerBound	double value	Specialized annotations for the types double, int and String, used to check whether a return value satisfies a given lower or upper bound, or is within a given range.
	@DoubleRange	double min double max	
	@DoubleUpperBound	double value	
	@IntLowerBound	int value	
	@IntRange	int min int max	
	@IntUpperBound	int value	
	@StringLowerBound	String value	
	@StringRange	String min String max	
	@StringUpperBound	String value	
	@StringLengthRange	int min int max	Checks whether a String has a length within the range [min,max]

Table 13: *Annotations used for single methods, i.e., property-annotations.*

no.ii.uib.ship.validation.annotations.cross			
	ANNOTATION	ELEMENTS	USAGE
RANGE	@OneLessThan	int value	Checks whether at least one of the involved return values is smaller or equal than value
	@AllLessThan	int value	Checks whether all the involved return values are less or equal than value
	@OneAtLeast	int value	Checks whether at least one of the involved return values is equal or greater than value
	@AllAtLeast	int value	Checks whether all the involved return values are equal or greater than value
NULL	@AllOrNoneNull	–	Checks whether either all the involved return values are null or none of them is
	@ExactlyNNull	int value	Checks whether exactly n of the involved return values are null
SUM	@SumMin	int value	Checks that the sum of the involved return values is not smaller than value.
	@SumRange	int min int max	Checks that the sum of the involved return values is within the integer range [min,max].

Table 14: *Annotations used on multiple methods, i.e., cross-annotations.*

4.1.5 Package **test**

This package provides examples of how the framework can be used, including the working example in this paper.

4.2 Sequence diagrams

In this section we describe in detail how the framework works in practice, using sequence diagrams.

4.2.1 Usage example

Figure 6 describes the execution of the code that the user has to put in his/her application in order to use the framework. For example, the execution of the following code can be traced in Figure 6:

```
InputObj o = new Webform(null,"BICCODE",null,"AB1232342", 10000, 10);
IValidatorFactory vf = new ValidatorFactory();
IValidator v = vf.getValidator();
ValidationSummary vs = val.validate(o);
boolean success=vs.getTestResults();
if (!success)
   System.out.println(vs.toString());
```

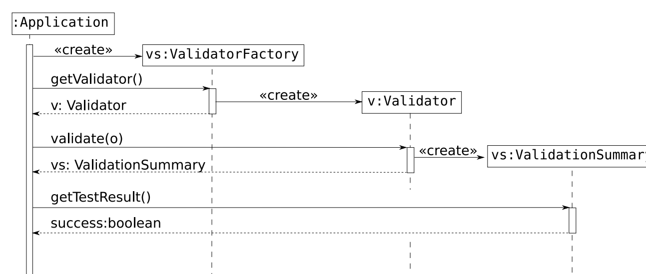


Figure 6: *Sequence diagram for the main steps of the validation process, where* `o` *is the object to be tested.*

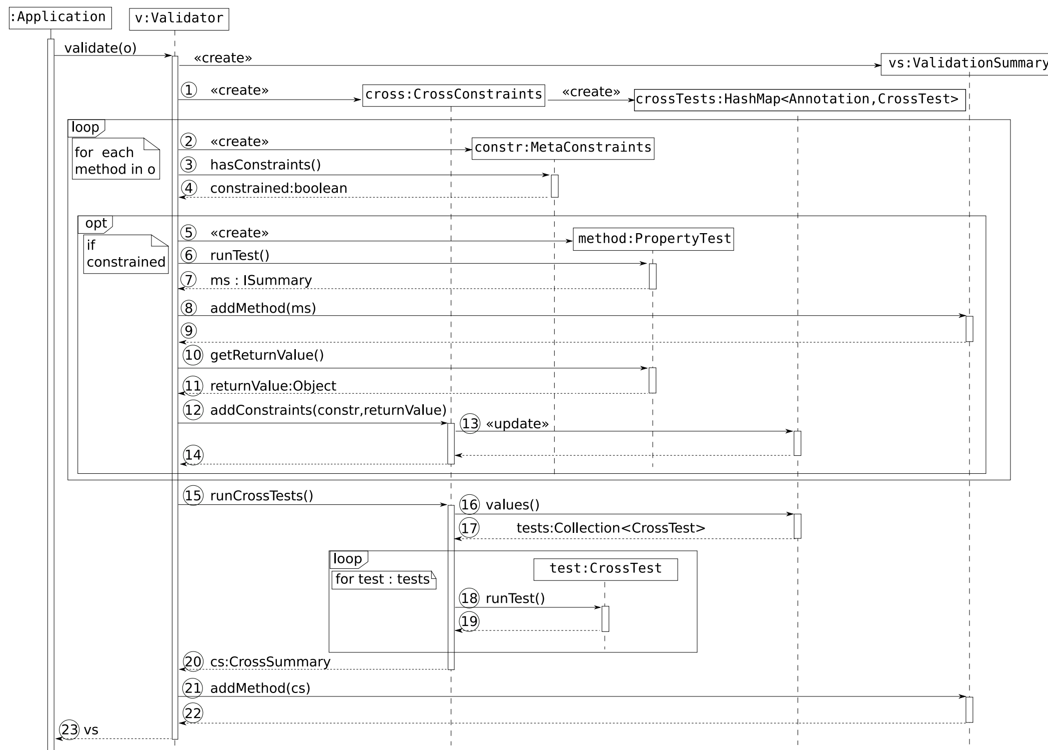
The methods returning values of properties to be tested, in this case the methods of the class `Webform` (which can be found in the package `test`), must be annotated by the programmer as shown in the example in Section 2. An important precondition assumed by the framework, is that all properties of an object to be validated, are available through no-arg methods, e.g. getter methods in JavaBeans. This is not an important restriction, especially for objects representing an input. If existing classes allow access to input only through methods with parameters, it should be easy to add a no-arg method, which then supplies default argument(s) to the former method.

The result of the validation is returned as an object of the type `ValidationSummary`, which can then be queried by the application.

4.2.2 The validation process

Details of the call to the method `validate()` of the `Validator` object, on an object `o`, are shown in Figure 7. The validator goes through all the methods of the object `o`, and by reflection finds all annotations

that are marked for validation. Then, for each method with such annotations, all local tests are recursively constructed as a tree, and run on the return value and a summary of the results is created. The cross-tests are collected and stored in a hash map as we go through the methods, and are run after all methods have been locally tested.



1. The `CrossConstraints` object keeps track of the cross-tests that should be run on values in the object `o`. Each cross-annotation in `o` is stored as a key in the hash map `crossTests`, together with a `CrossTest` object containing the list of return values associated with it.

Steps 2 to 14 are repeated for each method in the class of the object o.

2-4. `MetaConstraints` is used to extract and store the validation- and cross-annotations on the method. Only if there are any such constraints the value of `constrained` is true, and Steps 5 to 14 executed.

5-9. The `PropertyTest` takes care of retrieving the return value, running the tests on this value, and constructing the corresponding `MethodSummary`. The latter is added to the `ValidationSummary` by the `Validator`.

10-14. The `Validator` retrieves the return value of the method to pass it to the `CrossConstraints`, which, in turn, updates the list of values in the `crossTests` map for each cross-annotation which appears on the method.

15-20. Calling the method `runCrossTests()` causes each `CrossTest` object in the hash map `crossTests` to run the corresponding tests on its list of return values, in a similar way as `PropertyTest` does. Finally `CrossConstraints` creates a `CrossSummary` as described in Section 4.1.3.

21-22. The `CrossSummary` is added as a normal `MethodSummary` to the `ValidationSummary`.

Figure 7: *Sequence diagram for a call to validate properties of the object* `o`.

The real testing and creation of a summary is shown in the sequence diagram of Figure 8. This diagram shows how the the return value of the method `getBic()` is validated against the test represented by the annotation `@ValidateBIC`, and how the corresponding summary is created. The names of the objects are the same as those in the object diagrams in Figure 4 and Figure 5.

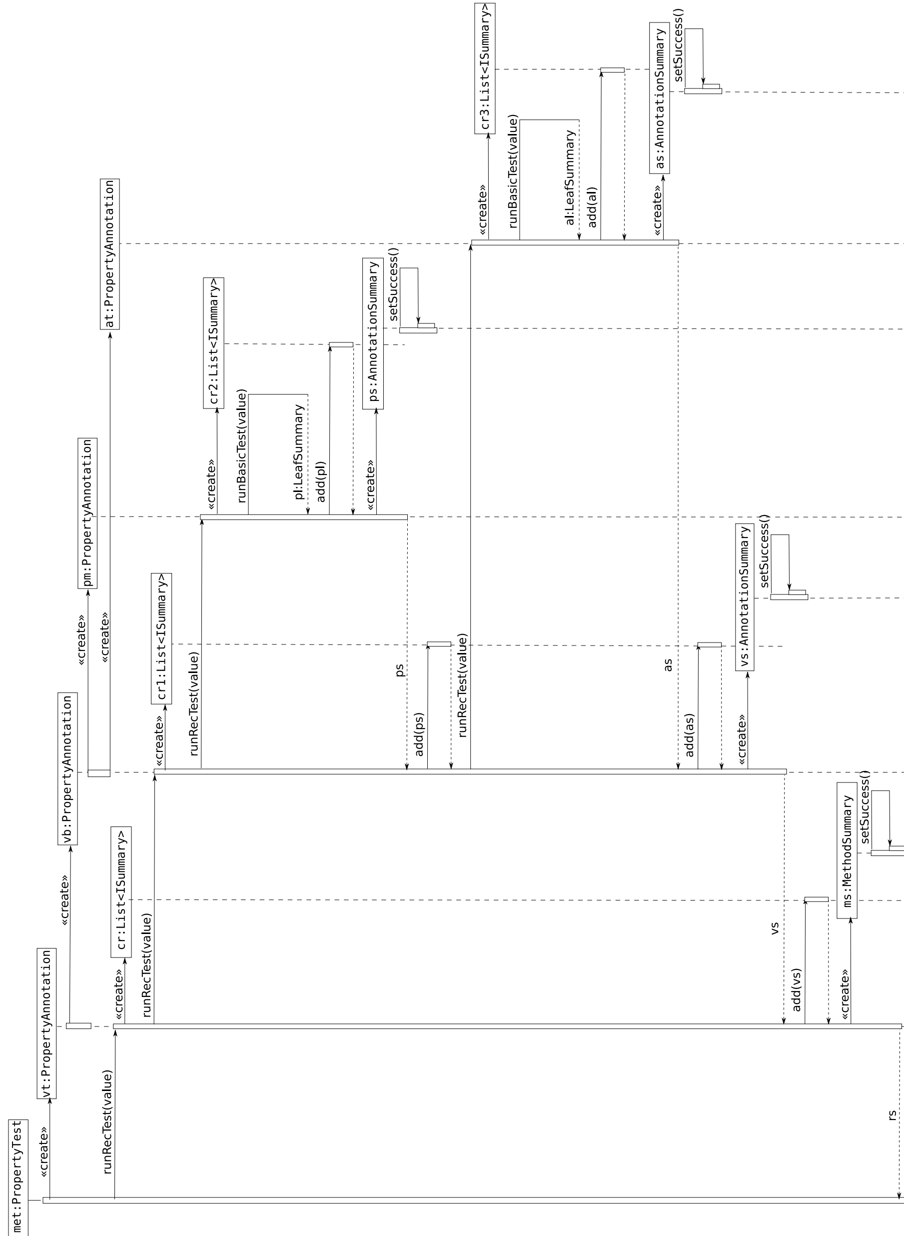


Figure 8: *Sequence diagram showing the actual validation of the `getBIC()` method.*

4.2.3 The validation summary

Table 15 shows the printout of the summary generated by the validator for the input given in Figure 1.

To illustrate the similarity of the structure of the summary with that of an `AnnotationObject`, the tree structure of the annotation `@AmountCheck` is included in the same table, where the boxes represent the `AnnotationSummary`s containing the result of corresponding partial tests.

This tree structure is also graphically illustrated in the summary printout, but here only the tests that caused the validation to fail are shown. Notice that because of the boolean operators, the meaning of failure is dependent not only on a single test, but on entire subtrees of the summary. For this reason a different message is printed according to which boolean operator was involved in the test. For instance, we can see that `@AmountCheck` failed because *all* of `@OneLessThen(1)` and `@AllLessThen(10000)` failed. If at least one had succeded, then also `@MaxAmount` would have succeded because of the `OR` operator, and therefore the whole `@AmountCheck` test would have passed. In case of an `AND` operator, we just print which tests failed. In case of an `OR` operator, we print that at least one of the tests should have succeeded in order for the test to pass. Finally, in case of the `ALLFALSE` operator, we print that all the tests should have failed, in order for the test to pass. We do this recursively on the tests, and the user can also decide the recursive depth of the summary, in order to obtain the desired level of details.

`AnnotationSummary` OBJECT CONTAINING THE RESULT OF THE `@AmountCheck` TEST
@AmountCheck (FAIL) AND @SumMin(1) (PASS) @MaxAmount (FAIL) OR @OneLessThan(1) (FAIL) @AllLessThan(10000) (FAIL) INPUT Euro=10000 Cents=10
PRINTOUT OF THE COMPLETE SUMMARY

```
The value "BICCODE" returned by "getBIC()"
has not passed the following property-test:
-Test: @ValidateBIC() because the following test(s) failed:
 |--Test: @PatMatch(value=\w{8}|\w{11})
===========================================
The following cross-tests have failed:
-Test: @AmountCheck() because the following test(s) failed:    |
 |--Test: @MaxAmount() because AT LEAST ONE                     |
    of the following test(s) should have succeded:             |
    |--Test: @OneLessThan(value=1)                              |
    |--Test: @AllLessThan(value=10000)                          |
-Test: @AllOrNoneNull()
-Test: @ExactlyOneNull() because ALL the following test(s) failed:
 |--Test: @ExactlyNNull(value=1)
===========================================
```

Table 15: *Validation of the form given in Figure 1, by the tests defined in Tables 1, and Tables 3 and 6.*

5 Related work

There are other ways of tackling the input validation problem, but they are fundamentally different from our approach. For instance, there are static analysis tools [12] which provide support for tainting [13, 14] or tools that provide specific solutions for particular input validation vulnerabilities like the AntiSamy project for XSS [15].

Our framework is designed to allow the user to easily define and integrate custom validation tests in the application. Replacing the classical XML configuration files with annotations retains most of the advantages of having an external configuration file, like decoupling of validation logic from the application logic, and reusable tests. In addition, methods and classes do not need to be referred by string references any more, which is very error prone and requires additional maintenance. One disadvantage of switching to annotations, might be that runtime changes are not possible anymore. However, being forced to recompile after making changes helps to ensure the type safety of the application.

Additional advantages of using annotations instead of XML configuration files are discussed in Holmgren [8] and Hookom [7]. Both papers also include some technical solutions for using annotations to validate object properties, but only provide some basic illustrative code, rather than a fully functional framework. However, it seems like the ideas in [7] are the starting point for the Hibernate validator and the work in [9].

Many basic technical solutions we use are very similar to those provided by Hookom and Hibernate. For example, using special meta-annotations as markers to allow the creation of custom validation-annotations and the way of associating tests to annotations. However, it must be said that these are standard solutions when annotations are involved.

When it comes to running the actual validation, we are close to the solutions proposed in [7, 9], 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. Although this approach allows tests on methods without return values, i.e., setter methods or methods with parameters, it makes the test code and the application code more interdependent, which is what we have tried to avoid.

Composition is also proposed in [9], although the full scope of boolean operators does not seem to have been addressed. What is called a multi-valued constraint in [9], i.e., applying the same annotation with different element values to the same method, can easily be achieved in our framework by encapsulating each instance in another annotation as in Figure 2.

Struts 2 [3] also provides validation through annotations. It offers a limited set of standard annotations, with no possibility of creating custom tests. As new annotations cannot be created, composition is not possible and the only way to add custom tests is to use a `@CustomValidator` annotation which takes as argument the name of the test. This is then associated to the corresponding class in an XML configuration file. In other words, despite the use of annotations, classes are still referenced to by string names.

Most importantly, none of the mentioned related work seems to consider the possibility of validating multiple properties. We consider our cross-annotations a natural extension of validation-annotations which can add expressive power to the validation-tests that the user can design. Besides, we manage to keep most of the technicalities involved in cross-validation hidden inside the framework, so that there is almost no difference between property- and cross-annotations from a user point of view, and usability is not compromised.

Finally, most of these frameworks are mainly designed to work with JavaBeans, and make strong assumptions about the type of applications that can utilize them. Our framework, as Heimdall, does not assume much about the application, and should be easy to integrate with any Java project.

6 Conclusion and future works

We have implemented a flexible content-validation framework [10] based on Java annotations, which can easily be integrated into existing applications. The main idea in the design of this framework has been that it should be easy to create libraries of custom validation-annotations, and that these tests should be highly reusable. We have tried to provide simple, yet powerful means doing this, for example, using boolean composition. Besides, we have pushed the limits of annotations by allowing constraints involving interdependent properties, which have not been addressed in any work we are aware of.

For future work, we intend to extend the library of predefined annotations by creating new tests aimed at specific input validation vulnerabilities and improve the validation summary to support specific queries.

References

[1] "OWASP Top Ten project," May 2009. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project 1

[2] "Commons validator," Apache, May 2009. [Online]. Available: http://commons.apache.org/validator/ 1

[3] "Struts," May 2009. [Online]. Available: http://struts.apache.org 1, 3.1.1, 5

[4] "Hibernate," May 2009. [Online]. Available: https://www.hibernate.org/ 1, 3.1.1

[5] L.-H. Netland, Y. Espelid, and K. A. Mughal, "A reflection-based framework for content validation," in *ARES*. IEEE Computer Society, 2007, pp. 697–706. 1, 3.1.1

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

[7] J. Hookom, "Validating objects through metadata," O'Reilly, January 2005. [Online]. Available: http://www.onjava.com/lpt/a/5572 1, 3.1, 5

[8] 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 1, 3.1, 5

[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 1, 3.1, 5

[10] D. Hovland, F. Mancini, and K. A. Mughal, "Ship validator," August 2009. [Online]. Available: http://shipvalidator.sourceforge.net 1, 6

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 4.1.2, 4.1.3

[12] B. Chess and J. West, *Secure programming with static analysis*. Addison-Wesley Professional, 2007. 5

[13] W. Pugh, "Jsr 305: Annotations for software defect detection," September 2006. [Online]. Available: http://jcp.org/en/jsr/detail?id=305 5

[14] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *ACSAC*. IEEE Computer Society, 2005, pp. 303–311. 5

[15] "OWASP AntiSamy project," OWASP, May 2009. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project 5