Design Patterns for the Implementation of Constraints on JavaBeans

Holger Knublauch Martin Sedlmayr Thomas Rose

Research Institute for Applied Knowledge Processing (FAW)
Helmholtzstr. 16, 89081 Ulm, Germany
{knublauch|sedlmayr|rose}@faw.uni-ulm.de

Abstract. Although constraints are useful to express semantic restrictions on object structures, Java provides little support for their implementation and evaluation. We argue that constraints should be specified explicitly, and propose a set of design patterns for the representation of constraints on JavaBeans. Object-oriented reflection is used to evaluate these constraints at run-time. We show how this technology can be used to support data and knowledge acquisition, and system evaluation.

1 Introduction

Constraints are meta-data about the semantic restrictions that are assumed to be satisfied in a given set of objects. For example, constraints can restrict the range of an object's integer attribute or the maximum cardinality of an array. Constraints are useful to document and communicate the semantics of classes, to validate the correctness of object states at run-time, to repair inconsistent data structures, and for various types of reasoning.

Although constraints are assumed to hold on almost every object model, software developers rarely specify them explicitly. At the most, semantic information is modeled in the design models of a software product. For example, UML provides the Object Constraint Language (OCL) [9], which can be used to declare formal constraints in class diagrams. These constraints are usually lost in the resulting implementation, because languages such as Java provide little support to express them explicitly.

In this paper, we show that it is both feasible and beneficial to implement constraints explicitly in Java. We propose standard conventions for representing constraints on Java data structures, based on design patterns similar to those from the JavaBeans specification [7]. Our simple design patterns specify templates of methods such as `getXXXMaxValue` for a property named `XXX`. These templates can be queried at run-time by means of object-oriented reflection.

This document is organized as follows. Section 2 provides some background

on the types of constraints typically needed in class structures. Section 3 briefly discusses the JavaBeans approach to specifying meta-data by means of design patterns. Section 4 introduces design patterns for the representation of constraints. Section 5 describes utility classes and tools that we have developed in support of these design patterns. Finally, we discuss benefits and limitations of our approach and draw some conclusions.

2 Constraints in object-oriented systems

In object-oriented programs, constraints describe the valid states of objects, i.e. the admissible values of attributes in any given situation. The following data structure, that will serve as the running example throughout this document, contains an informal description of some typical constraints.

```
abstract class Person {
    int age;            // Must be >= 0
    Person[] children;  // Max. cardinality 0, if spouse is null
    Person spouse;      // Must not be one of the children
}
class MalePerson extends Person {} // Spouse must be FemalePerson
class FemalePerson extends Person {} // Spouse must be MalePerson
```

One of the most basic purposes of specifying constraints is to document and communicate the semantics of classes between developers and programs. Components with well-defined interfaces are much easier and safer to (re)use.

In order to maintain constraint satisfaction, programmers try to ensure that any execution of their systems leads to consistent data structures only. Constraint satisfaction is assumed when the system's run-time objects are modified. For example, when instances are created or edited by human users, intelligent user interfaces must reject invalid input and advice the user to edit missing or wrong input. This is especially important in knowledge-based systems, where knowledge has to be acquired into a class structure with well-defined semantics (an *ontology*) [6]. Here, constraints can help to clarify meanings and to detect inconsistencies in a knowledge-base.

Some applications have to cope with objects that were instantiated externally, for example when objects are communicated across a network, such as in multi-agent systems [3]. Here, autonomous software agents communicate with each other by exchanging objects with well-defined, restricted meanings. Objects that fail to satisfy these semantic restrictions have to be rejected or adapted.

Finally, data structures with explicit semantics allow to apply generic reasoning and deduction methods on them. For example, the information that the value of a `Person`'s `spouse` attribute must be of the opposite sex, can be used to find potential spouses from a list of objects.

The following types of constraints on class attributes cover typical requirements from knowledge-based and multi-agent systems. They are similar to those

defined in the Open Knowledge Base Connectivity (OKBC) specification [1], which is a standard for exchanging knowledge and meta-data.

- *Numeric range* (Minimum and maximum values)
- *Valid and invalid values* (The set of potential or illegal values)
- *Admissible value types* (The set of classes for potential values)
- *Default values* (If other values are unknown)
- *Cardinalities of arrays* (Minimum and maximum sizes)
- *Duplicates* (Whether an array may contain duplicates)
- *Ordering* (Whether the array elements must be ordered)

3 JavaBeans and design patterns

In the context of meta-data such as constraints, it is straight-forward to explore the types of meta-data supported by Java. In object-oriented systems, meta-data provides information about the classes of objects, and their attributes and methods. *Reflection* is used to access this information at run-time. The Java Reflection API contains – among others – methods to identify the names and types of class attributes. This allows to implement generic algorithms that operate on any class structure and dynamically explore the properties of these classes.

The JavaBeans specification defines coding conventions that support the development of such generic algorithms. The coding conventions define how classes should expose their meta-data, so that reflection can be applied best. The main purpose of the JavaBeans specification is to provide a uniform interface for reusable classes, that can be connected and analyzed with generic tools, such as visual GUI builders. A JavaBeans instance (a *bean*) provides meta-data about its features, which are the attributes describing its state, the events it fires, and the methods it offers for other components to call. The attributes, which are called *properties*, can be either of a primitive type, references to other objects, or one-dimensional arrays of these types (the *indexed properties*). A JavaBeans class exposes its list of properties by declaring certain methods, that respect well-defined design patterns. For example, the following two methods declare that the class has a property "age" of type `int`, which can be read and written.

```
public int getAge()              // e.g.  { return age; }
public void setAge(int value)    // e.g.  { age = value; }
```

Properties are called *bound* if they fire a `PropertyChangeEvent`, whenever their value has changed. Thus, beans do not only provide static meta-data, but also pro-actively notify the environment they are embedded in. This attractive meta-data support makes JavaBeans a candidate implementation platform for

semantically rich data structures. However, the types of meta-data provided by the JavaBeans standard are quite limited. There is little information provided apart from the names, types and access rights of the properties. In the following sections we will argue that other useful meta-data – especially constraints – can (and should) be supplied by Java classes, by applying design patterns very similar to those from JavaBeans.

4 Design patterns for constraints on JavaBeans

The following subsections specify new design patterns for representing various types of constraints on the properties of JavaBeans classes. The patterns are templates (coding conventions) for public methods that deliver meta-data about the constraints that are to be satisfied by the given object. Reflection can be used to locate the methods that obey the templates. If no such methods can be located, the Java default of no constraints is assumed to hold. If a located method is declared `static`, then the constraint is assumed to hold on all instances of the class.

In the following, the abbreviation `XXX` stands for the property name in its capitalized form, like in JavaBeans, so that `getXXX` means `getAge` for the `age` property. `<type>` denotes the (simple) property type.

4.1 Minimum and maximum values of numeric properties

For properties of numeric types, the following methods can be used to declare minimum and maximum values.

```
<type> getXXXMinValue()
<type> getXXXMaxValue()
```

The following example specifies that the minimum `age` of any `Person` is 0.

```
public static int getAgeMinValue() {
    return 0;
}
```

4.2 Valid and invalid property values

The following pattern allows to explicitly specify the list of admissible values for a property.

```
<type>[] getXXXValidValues()
```

In some cases, it might be easier to define the set of valid values by excluding invalid values. This can be achieved with the following pattern.

```
<type>[] getXXXInvalidValues()
```

The following example declares that a `Person`'s `spouse` must not be one of his or her `children`.

```
public Person[] getSpouseInvalidValues() {
    return children;
}
```

4.3 Admissible types of non-primitive properties

Sometimes the admissible values of non-primitive properties must be of certain types only. The following patterns allow to define valid or invalid classes explicitly.

```
Class[] getXXXValidClasses()
Class[] getXXXInvalidClasses()
```

The resulting array must consist of subclasses of the (simple) property type. All other (sub)classes are excluded. For example, the following code from the `MalePerson` class specifies that the `spouse` must be a `FemalePerson`.

```
public Class[] getSpouseValidClasses() {
    return new Class[] { FemalePerson.class };
}
```

4.4 Default values

This pattern allows to specify a property's default value. This may be useful to fill in missing user input, to initialize a data structure, or to return it to admissible values if a constraint has been violated.

```
<type> getXXXDefaultValue()
```

4.5 Cardinality of indexed properties

In order to specify the minimum and maximum cardinalities of an indexed property, the following methods can be declared.

```
int getXXXMinCardinality()
int getXXXMaxCardinality()
```

Both values must be non-negative and the minimum must be less or equal to the maximum value. For example, the following method from the `Person` class declares that a `Person` with no `spouse` may have no `children`.

```
public int getChildrenMaxCardinality() {
    return (spouse == null) ? 0 : Integer.MAX_VALUE;
}
```

4.6 Duplicates in indexed properties

If the following method returns `true` for a given indexed property, then all its array elements must be different, with regard to the `equals` method.

```
boolean isXXXDuplicateFree()
```

4.7 Ordering of comparable, indexed property values

The following template is only applicable for an indexed property whose array elements implement the `java.lang.Comparable` interface. If the given method returns `true`, then all elements must be ordered, i.e. if `next` is the successor of `element`, then `element.compareTo(next)` must deliver -1 or 0.

```
boolean isXXXOrdered()
```

5 Supporting classes and tools

Whereas the preceding section specified how constraints can be declared in Java classes, this section will introduce classes and tools that simplify and standardize their access and use. These "KBeans" classes are available from our webpage [2].

5.1 Design pattern access

Similar to the JavaBeans introspection classes, which extract meta-data on beans using reflection, we have implemented standard support methods for the design patterns from section 4. For example, the following static method obtains the minimum value of a numeric property, if the responsible pattern from subsection 4.1 has been implemented in the class, and `null` otherwise.

```
Number getPropertyMinValue(Object bean, String propertyName)
```

5.2 Constraint classes

Based on these standard access methods, we have implemented a hierarchy of classes that encapsulate constraint descriptions with methods to evaluate them (figure 1).

`Constraint` objects have a reference to a bean and an `isSatisfied` method to check whether the constraint is currently satisfied by the bean. The class `PropertyConstraint` represents constraints on a specific bean property and contains an `isValidValue` method, which can be used to test potential or current property values individually. For indexed properties, the values are the complete arrays, not the array elements. In order to check single array elements, the `IndexedPropertyConstraint` class adds an `isValidIndexedValue` method.

Various subclasses of these base classes support the standard constraints from section 4. For example, the `RangeConstraint` class describes and tests numeric minimum and maximum values, and the `OrderedIndexedPropertyConstraint` class tests whether the elements of an indexed property are ordered.

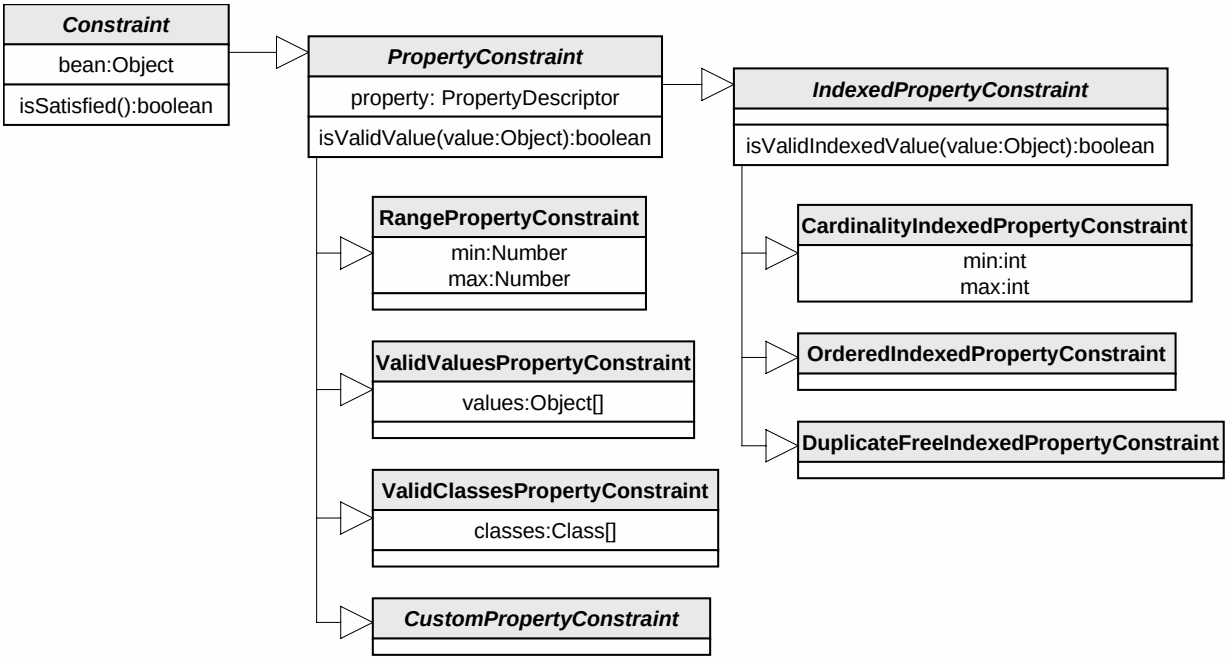

Fig 1. A simplified model of the KBeans constraint classes.

A utility class can create these standard `Constraint` objects. Additionally, it checks for implementations of the following design pattern, which can be used to specify custom constraints that can not be expressed in terms of the standard constraint types.

```
CustomPropertyConstraint[] getXXXCustomConstraints()
```

5.3 Automated constraint checking

Applying the `Constraint` classes, we have implemented utility classes to automate constraint checking. The `ConstraintManager` class checks constraints whenever the value of any property in a given data structure has changed. Here, the fact that each change on a bound JavaBeans property produces a `PropertyChangeEvent` is used for two purposes. First, the `ConstraintManager` only needs a reference to a "root" object, from where it traces the links to any other object that is accessible from it. The `ConstraintManager` registers itself as a `PropertyChangeListener` on these objects and is thus able to autonomously register or unregister as a listener of the objects that are added to or removed from the data structure. Second, any property change can cause an execution of the constraint checking. If the list of constraint violations has changed, `ConstraintEvents` are sent to all interested `ConstraintListeners`.

Figure 2 illustrates a scenario, in which a `ConstraintManager` observes a given data structure of `Person` objects and reports constraint violations to a user interface object.

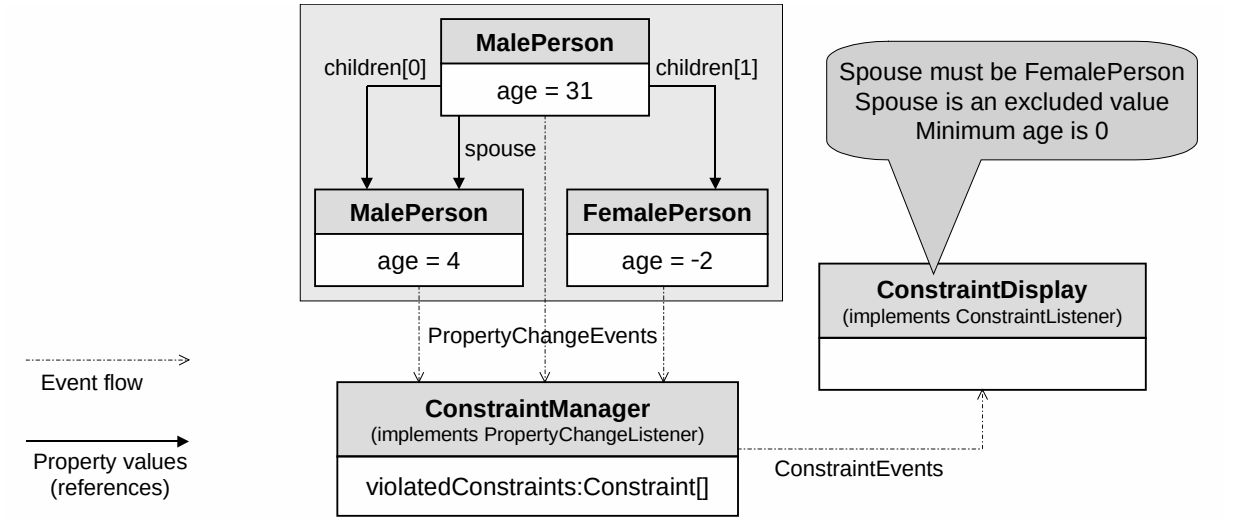

Fig 2. A `ConstraintManager` that pro-actively notifies a display component about constraint violations in a JavaBeans data structure.

5.4 Tool support for data and knowledge acquisition

Utilizing the constraint checking mechanism described above, we have developed a tool called `KBeansShell` that serves as a graphical editor of any structure of JavaBeans instances (figure 3). Besides being a valuable programming and debugging tool, the shell has already proven to be very useful as a knowledge acquisition tool for various knowledge-based systems [5]. It allows to display and edit beans as a tree or a graph.

The shell uses a `ConstraintManager` to detect constraint violations. The current violations are listed in an extra window, and a double-click on a constraint violation moves the cursor to the tree node that contains the invalid property value. Thus, the user is pro-actively guided through the data or knowledge acquisition process.

The open architecture of the `KBeansShell` allows to include application-specific components where needed. For example, it obeys the JavaBeans standard, which allows to define custom editing components for classes or properties (`Customizers` and `PropertyEditors`). Such components can check the constraints to actively restrict user input to only admissible values, e.g. as a combobox showing the values from the "valid values" design pattern. Intelligent user interface components could even automatically replace missing data with default values.

The shell can also be used to examine the run-time data structure in an executing application. For example, in a knowledge-based patient monitor for anesthesia [5], the developers can open a `KBeansShell` to browse or even change the objects in the system's knowledge-base. Thus, in support of rapid prototyping, constraint violations can be easily detected, reducing debugging overhead.

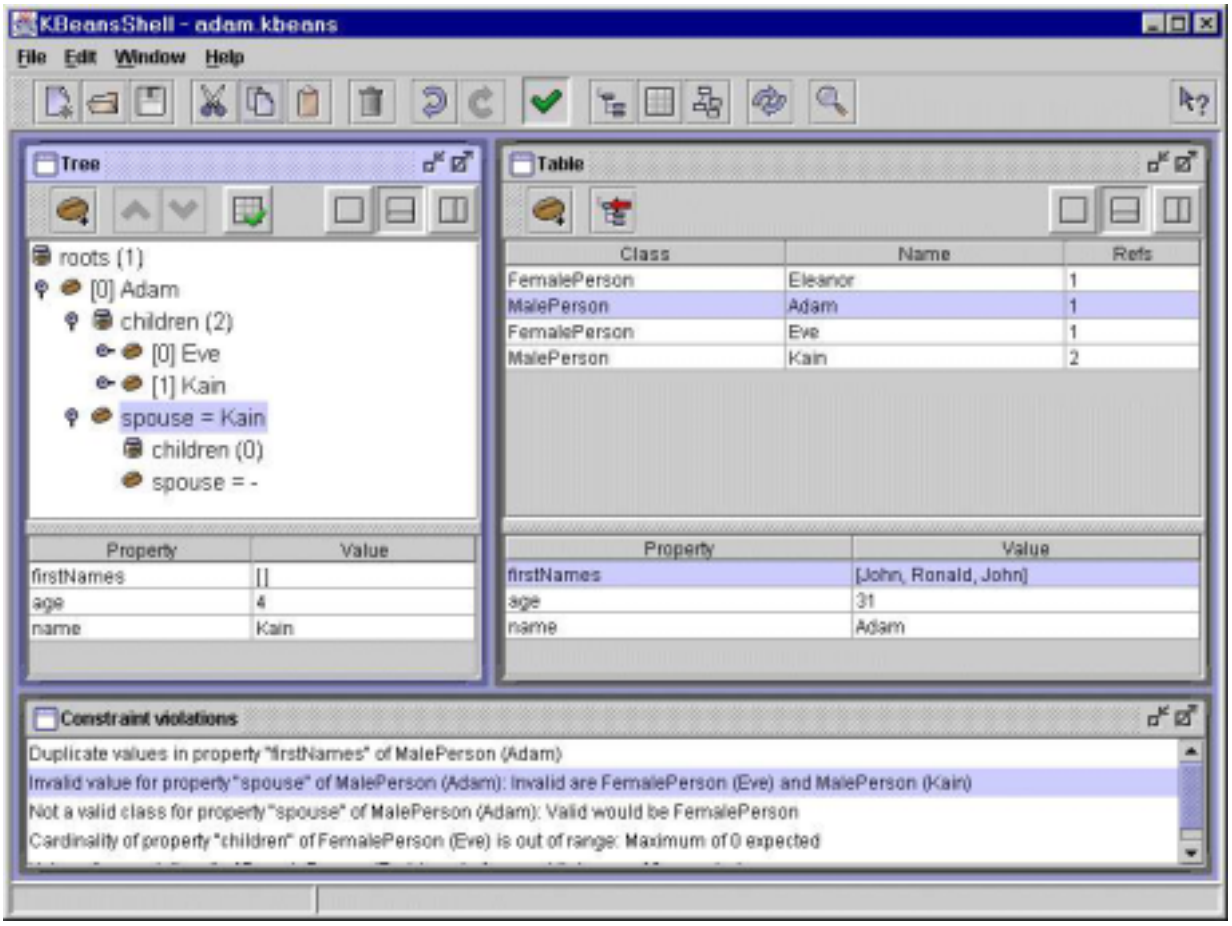

Fig 3. The `KBeansShell`, a generic graphical editor for JavaBeans instances.

6 Discussion

In this document, we have presented a simple but powerful approach for implementing constraints on JavaBeans. Its simplicity lies in the use of clear design patterns, that are easy to apply, comprehend, and evaluate. The patterns are based on pure Java code and do not rely on external formats or declarations. Especially, the meta-data remain with the class they describe, so that component exchange is facilitated. Since each of the patterns is optional, and their omission in a class means to apply Java defaults, any existing JavaBeans data structure is backwards compatible to our format. The constraint methods do not have any impact on the system's performance, because they only provide optional meta-data. These meta-data can be used by debugging components, such as the `ConstraintManager`, until the system is sufficiently evaluated.

The names of the methods describing the constraints were chosen to be intuitive. At the same time they are compatible to the JavaBeans specification, i.e. the constraints themselves can be regarded as JavaBeans properties of the objects (e.g. `ageMinValue` is a meta-data property for `age`).

The small core set of well-defined standard constraints supports compatibility with related standards and tools. However, our approach is extensible for other constraint types beyond those presented here. First, individual design patterns

can be introduced. Second, the `CustomPropertyConstraint` class can be subclassed by constraint checkers of arbitrary complexity.

Embedding constraint expressions into the source code allows to exploit the full scope of Java functions and operators, and a very efficient run-time performance. The expressiveness of Java allows to evaluate constraints for each instance individually. Additionally, the use of standard Java means that any Java software development tool can be used for constraint definition.

However, the flexibility of Java expressions is also the cause of some open issues, that we hope to cover in future work. Especially, there is yet no support to automatically generate the Java expressions from constraints in other specification formats, such as OCL or OKBC. In support of reverse or round-trip engineering [5], we currently investigate in a bi-directional mapping between parts of UML/OCL and our format. For example, it is straight-forward to translate between cardinality expressions in UML relationships and the corresponding design patterns.

A limitation of our approach is that it is only applicable to JavaBeans classes. The fact that JavaBeans properties must be declared `public` might lead to privacy violations. However, this limitation does not concern knowledge-bases and agent interfaces, the basic goal of which is to declare explicit, sharable class structures, that are public by their very nature.

Our approach is related to recent work in the context of adding assertions to Java, for example in support of *Design by Contract* [4, 8]. Similar to this, the JavaBeans standard includes the `VetoableChangeEvent` concept, in which a property's `setXXX` method can reject potential values. However, these approaches only allow to *test* constraint satisfaction, but provide little meta-data that could be used constructively, for example to guide user input.

7 Conclusions

The approach to implementing constraints on JavaBeans by means of the simple design patterns presented in this paper allows to enrich Java classes with valuable meta-data. The main strength of our approach is to make these semantic meta-data explicit and executable, so that generic classes and tools can efficiently use them. This includes generic classes for monitoring the state of an object structure, for implementing intelligent user interfaces, for data and knowledge acquisition, and various types of reasoning. These classes can help to reduce development overhead and to improve software quality.

References

[1] V. Chaudhri, A. Farquhar, R. Fikes, P. Karp, and J. Rice. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proc. of the AAAI-98*, Madison, WI, 1998.

[2] FAW Ulm. KBeans Homepage. `http://www.faw.uni-ulm.de/kbeans`, 2000.

[3] N. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Int. Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[4] M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design By Contract. In *Proc. of the Second International Conference on Metalevel Architectures and Reflection (Reflection'99)*, Saint-Malo, France, 1999.

[5] H. Knublauch and T. Rose. Round-Trip Engineering of Ontologies for Knowledge-Based Systems. In *Proc. of the Twelfth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Chicago, IL, 2000.

[6] R. Studer, D. Fensel, S. Decker, and V.R. Benjamins. Knowledge Engineering: Survey and Future Directions. In *Proc. of the 5th German Conf. on Knowledge-based Systems*, Würzburg, Germany, 1999.

[7] Sun Microsystems. JavaBeans Specification. `http://java.sun.com/beans`, 1997.

[8] Sun Microsystems. A Simple Assertion Facility. `http://www.javasoft.com/aboutJava/communityprocess/review/jsr041`, 2000.

[9] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addision-Wesley, 1998.