KBeans Specification

Semantic Transparency for Components and Domain Models

Technical Report FAW-TR-01001, FAW Ulm

HOLGER KNUBLAUCH
Research Institute for Applied Knowledge Processing (FAW)
Helmholtzstr. 16, 89081 Ulm, Germany
Holger.Knublauch@faw.uni-ulm.de

Semantic transparency allows to communicate the intended meaning of domain models and the valid application scenarios of components between humans and machines. Object-oriented languages like Java provide little support for the implementation of semantic transparency, because they lack primitives for the declaration of semantic metadata on classes. KBeans is a natural extension of the JavaBeans specification which allows to declare and process facets, which provide metadata on the semantics of object properties. Facets are implemented by fields and methods which meet predefined naming conventions. Object-oriented reflection is used to detect and access these fields and methods at runtime. The metadata provided by the facets can be used to prevent, detect, and repair invalid object states. KBeans supports a catalog of facet types which embraces related standards such as XML Schema and OKBC. The benefits of KBeans are demonstrated in case studies from the domains of component reuse, ontology sharing, software testing, and knowledge acquisition.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Object-oriented Programming; D.2.13 [**Software Engineering**]: Reusable Software; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms: Design, Documentation, Languages, Reliability, Verification
Additional Key Words and Phrases: Component reuse, Constraints, JavaBeans, Ontologies, Semantic Transparency

1. INTRODUCTION: THE NEED FOR METADATA

Two central visions of research in object-orientation are to support reusable components and smooth transitions between informal domain entities and executable classes [Booch et al. 1999]. We argue that object-oriented programming languages like Java have not yet exhausted their full potential to support this vision.

The basic idea of component-based development [Pree 1997] is to build applications by plugging together reusable building blocks (*components*), which encapsulate their state and expose their services through well-defined interfaces. These interfaces enable developers to link and configure components by means of visual programming tools and GUI builders. However, the presence of externally developed components within a system introduces new challenges for software engineering activities, particularly due to hidden dependencies among components, reduced testability, and difficulties in program understanding. The main reason for these

Version 1.0.0 (02.07.2001) – For updates visit http://www.faw.uni-ulm.de/kbeans

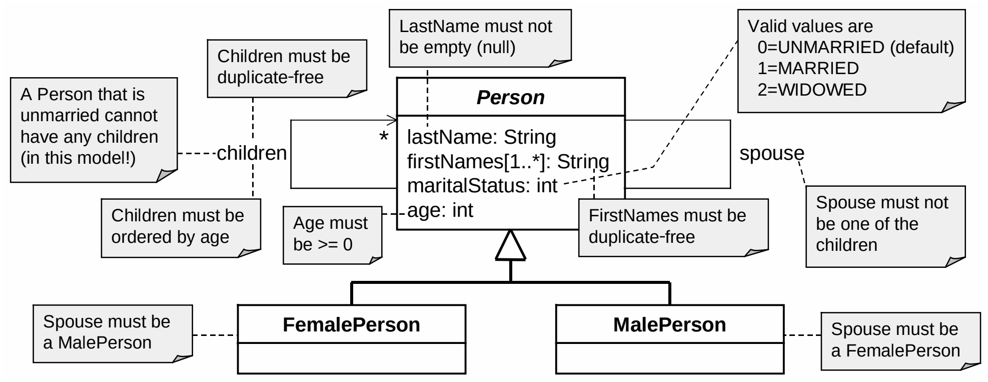

LastName must not be empty (null)
Children must be duplicate-free
Valid values are 0=UNMARRIED (default) 1=MARRIED 2=WIDOWED
A Person that is unmarried cannot have any children (in this model!)
Person
*
children
lastName: String
firstNames[1..*]: String
maritalStatus: int
age: int
spouse
Spouse must not be one of the children
Children must be ordered by age
Age must be >= 0
FirstNames must be duplicate-free
Spouse must be a MalePerson
FemalePerson
MalePerson
Spouse must be a FemalePerson

Fig. 1. A simple domain model and its intended semantics (UML notation).

problems is the lack of information about components that are not internally developed [Orso et al. 2000].

Another challenge in object-oriented projects is to find a smooth bidirectional mapping between informal domain entities and a class model that maintains the domain's semantics. As illustrated in figure 1, even simple domain models can be based on a considerable amount of semantical assumptions. These assumptions are important, because they constrain the space of valid application scenarios, and thus allow to use and reuse models safely and efficiently. However, when class models such the example presented in figure 1 are implemented in an imperative language like Java, their original semantics are usually obscured behind non-transparent programming primitives. The constraint that a person's age must not be negative might be implemented by `if`-clauses that reject negative values before they are assigned to the `age` attribute (e.g., in the `setAge` method). The constraint that a person's marital status must be one of three predefined constants might be implemented by three appropriate radio buttons. Thus, the semantics are typically implemented *implicitly*, especially by making sure that any execution path through the program will result in valid object states. The complexity of potential execution paths often leads to systems that are error-prone, hard to understand and maintain, and almost impossible to reverse engineer [Demeyer et al. 1999].

The origin of the problems of components and domain models described above is that their developers have implemented semantical assumptions that are neither transparent to other developers nor to the software environment in which the components and domain models are embedded in. As a result, object-oriented classes (including components) are frequently instantiated in ways that violate these assumptions.

Semantic transparency [Cover 1998] means that machines and humans are presented with models that are both *unambiguous* (having a precise, predictably interpreted meaning) and *meaningfully correct* (simultaneously satisfying a number of integrity constraints). A computer's notion of "meaning" is restricted to *values* that can be related to other values that have a meaning to humans. Therefore, semantic transparency for machines can be reduced to formal information about the elements of an object model and their relationships. Thus, a key to implementing semantically transparent objects is to provide information (or *metadata*), especially about the valid object states. The goal of this document is to show that it is both feasible and beneficial to formally specify metadata on Java objects, so that their semantics are transparent. With these metadata, machines and humans are able to *prevent*, *detect*, and eventually *repair* object states that violate the semantics intended by their developers.

The path to our solution is as follows. Section 2 analyzes how non-imperative (i.e., declarative) languages, in particular schema and knowledge representation languages, implement semantic transparency. These languages allow to declare *facets*, which provide metadata about attributes, such as their minimum value. The *KBeans* (Knowledge Beans) approach presented in this document enriches Java standards to allow for the declaration of such facets. The enabling technology for this is object-oriented *reflection* [Maes 1988], which can be used to extract metadata about classes, attributes, and methods at runtime, so that *generic* algorithms and tools can process them. The Java component model *JavaBeans* [Sun Microsystems 1997] is based on reflection and provides several generic services to process attributes and their values. We argue that reflection and JavaBeans (both are briefly introduced in section 3) lay an attractive foundation for implementing facets.

After the two background sections, the main part of this document is organized as follows. Section 4 introduces KBeans, a flexible extension of JavaBeans to express facets of predefined types. Section 5 contributes details on the single facet types, which can be used to transparently declare semantics. Section 6 describes various application scenarios and case studies with KBeans from the domains of component reuse, software testing, and knowledge acquisition. Section 7 compares KBeans with related approaches for enriching Java with semantics and constraints. The paper is concluded with a summary in section 8.

2. FACETS TO PROVIDE METADATA FOR SEMANTIC TRANSPARENCY

The space of potential metadata about objects is vast (cf. [Meyer 1997; Orso et al. 2000]): We could define invariants about the possible states of objects, the valid values of arguments that are passed to a method, pre- and post-conditions of methods, the performance of an algorithm, or even details on the internal computations within a method, down to every single command. However, not all of these types of metadata are equally important. In order to reduce complexity, we will focus on metadata that can support to reuse, configure, and interface components and to instantiate domain models. In particular, we will focus on metadata about the values of class attributes (JavaBeans properties). We will show later that these metadata can significantly improve semantic transparency of reusable objects, because they constrain the ways the objects can be accessed and configured by their environment without violating the semantics intended by their authors.

Given this focus on attribute values, how do other languages than Java tackle the problem of providing semantic metadata? Semantic transparency is an increasingly popular topic of research in those domains that deal with communicating data and knowledge bases between distributed systems. Therefore, the following two

Table I. The constraining facets of XML Schema.

Facet name	**Value type**	**Facet type**	**Semantics**
`length`	`string`, list	`int` (≥0)	(Exact) string or list length.
`minLength`*	`string`, list	`int` (≥0)	Minimum string or list length.
`maxLength`*	`string`, list	`int` (≥0)	Maximum string or list length.
`pattern`	`<type>`	reg. expr.	Lexical space of the values.
`enumeration`	`<type>`	`<type>[]`	Enumeration of valid values.
`whitespace`	`<type>`	(3 options)	Preserve, replace, or collapse whitespaces.
`maxInclusive`	`<ordered>`	`<type>`	Inclusive upper bound (≤).
`maxExclusive`	`<ordered>`	`<type>`	Exclusive upper bound (<).
`minInclusive`	`<ordered>`	`<type>`	Inclusive lower bound (≥).
`minExclusive`	`<ordered>`	`<type>`	Exclusive lower bound (>).
`totalDigits`	`<decimal>`	`int` (>0)	Maximum number of digits.
`fractionDigits`	`<decimal>`	`int` (≥0)	Maximum length of the fractional part.

*Similar to `minLength` and `maxLength`, which constrain simple types, XML Schema supports `minOccurs` and `maxOccurs` declarations for complex types.

subsections will briefly review two of the currently most widely used data and knowledge representation languages.

2.1 Facets in the Document Description Language XML Schema

XML Schema [World Wide Web Consortium 2001] is an XML-based specification language for describing the syntax and semantics of a class of XML documents. Being an official standard recommendation of the World Wide Web Consortium, it is a richer alternative to the obsolete DTD format of XML.

XML Schema documents can be used to define simple elements (comparable to the primitive types of Java such as `int` and `boolean`) and complex elements (which can be built by combining other elements, comparable to classes in Java). New simple element types are defined by deriving them from existing simple types, e.g. by building lists of other types. In particular, it is possible to derive a new simple type by restricting an existing simple type. The legal range of values for the new type is a subset of the existing type's range of values. Restrictions (or constraints) are defined by means of so-called *facets*. Every facet has a name that is associated to predefined semantics, and a value of a type that can be uniquely derived from the simple element type that is constrained by the facet. For example, the facet named `minExclusive` defines the exclusive lower bound (using >) of a numerical element. The value of the `minExclusive` facet must be of the same type as the simple element, e.g. `int` for `int` elements. In order to cover the most frequently needed types of metadata, XML Schema provides a set of standard facet types, which is summarized in table I.

2.2 Facets in the Frame-Based Ontology Language OKBC

Ontologies [Gruber 1993] are formal, shared specifications of domain concepts and the relationships between them. The goal of defining ontologies is to provide information about the semantics of particular information items, so that they can be shared and processed as "knowledge" between agents and humans. Most ontology description languages allow to represent knowledge in terms of concepts (classes), relations, functions, axioms, and instances (cf. [Corcho and Pérez 2000]). These

Table II. The constraining facets of OKBC.

Facet name	Facet type	Semantics
`:VALUE-TYPE`	class	Type restriction on the values of the slot.
`:INVERSE`	slot	Any value is also a value of the specified slot.
`:CARDINALITY`	`int` ($\geq$0)	Exact number of values.
`:MAXIMUM-CARDINALITY`	`int` ($\geq$0)	Maximum number of values.
`:MINIMUM-CARDINALITY`	`int` ($\geq$0)	Minimum number of values.
`:SAME-VALUES`	slot chain	Values are equal to values of specified slot.
`:NOT-SAME-VALUES`	slot chain	Values are not equal to values of specified slot.
`:SUBSET-OF-VALUES`	slot chain	Values are a subset of specified slot.
`:NUMERIC-MINIMUM`	number	Inclusive lower bound of value ($\geq$).
`:NUMERIC-MAXIMUM`	number	Inclusive upper bound of value ($\leq$).
`:SOME-VALUES`	`<type>[]`	Enumeration of valid values.
`:COLLECTION-TYPE`	{`set`, `list`, `bag`}	Values are treated as set, list, or bag.

aspects are frequently structured into *frames* (e.g., [Chaudhri et al. 1998]), which can be compared to class hierarchies in an object-oriented sense.

The *Open Knowledge Base Connectivity (OKBC)* [Chaudhri et al. 1998] provides a set of representational constructs commonly found in frame-based knowledge representation systems. In a nutshell, OKBC ontologies define *classes* that have *slots*, which are comparable to object-oriented attributes. Similar to XML Schema, slots can be annotated with facets, especially to constrain the valid values of the slots. The standard facet types of OKBC are summarized in table II.

Although XML Schema and OKBC have fundamental differences, e.g. in their support for class inheritance [Klein et al. 2000], they share the idea of attaching facets to value types. Note that these facets only *declare values*, but the applications are free to process these values individually. For example, metadata provided by facets can be used to check correctness, to repair invalid values (e.g., by replacing an illegal value with its inclusive lower bound), and even to perform reasoning by solving constraint satisfaction problems.

3. METADATA IN JAVA: REFLECTION AND JAVABEANS

How can we implement and access facets in Java programs? One approach (e.g., [McLaughlin 2000]) is to include external formats such as XML Schema or OKBC documents into the software architecture, and rely on the facet services provided by parsers and components for those languages. However, such a mixture of Java and external files complicates software maintenance and contradicts to the paradigm of *traveling light* [Beck 1999] from evolutionary software processes, in which as much of the design decisions as possible are documented in the source code, and in which reverse engineering is used to extract the design models into a format such as UML. Furthermore, it requires an infrastructure to access the objects from the external files as Java objects. Access to data from XML or ontology documents is traditionally implemented by translating them into an intermediate layer of objects, such as DOM trees. This is very flexible, as it allows to access documents the schemas of which are unknown at build-time. However, it complicates access to the data contained in the documents and means an artificial mixture of modeling levels, because entities that are naturally considered as classes are treated as instances.

Unless we want to rely on external specifications written in declarative languages despite these difficulties or extend the core of the Java language itself with new commands to declare facets, we need to cope with the types of metadata access that are already supported by Java. Object-oriented *reflection* [Maes 1988] provides metadata about objects and their classes. The Java Reflection API contains – among others – methods to identify the names and types of class attributes and methods. This allows to implement generic algorithms that operate on *any* class structure by dynamically exploring the properties of these classes.

The *JavaBeans* specification [Sun Microsystems 1997] defines coding conventions that support the development of such generic algorithms. The main purpose of the JavaBeans specification is to provide a uniform interface for reusable classes (precisely, components), that can be analyzed and mutually connected with generic tools, such as visual GUI builders. A JavaBeans instance (a *bean*) provides metadata 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 arrays of these types (the *indexed properties*). A JavaBeans class exposes its list of properties by declaring certain methods, that follow pre-defined naming conventions or *design patterns*. For example, the following two *getter* and *setter* methods declare that the class has a property called "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 metadata, but can also pro-actively notify their environment about changes. This attractive metadata support makes JavaBeans a candidate implementation platform for semantically rich data structures. However, the types of metadata 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 metadata – especially constraints – can (and should) be supplied by Java classes, by implementing facets with design patterns very similar to those from JavaBeans.

4. THE KBEANS APPROACH: ADDING FACETS TO JAVABEANS

This section presents the KBeans approach to define and evaluate facets that declare metadata about JavaBeans properties. An example based on figure 1 will help to convey the main idea. In order to declare the constraints that the (inclusive) minimum age of a person is 0, and that a person's spouse may not be one of his or her children, we add the following members to the `Person` class.

```
public final static int ageMinInclusive = 0;
public Person[] getSpouseInvalidValues() { return children; }
```

As illustrated in figure 2, these fields and methods can be detected and evaluated by a reflection-based facet engine.

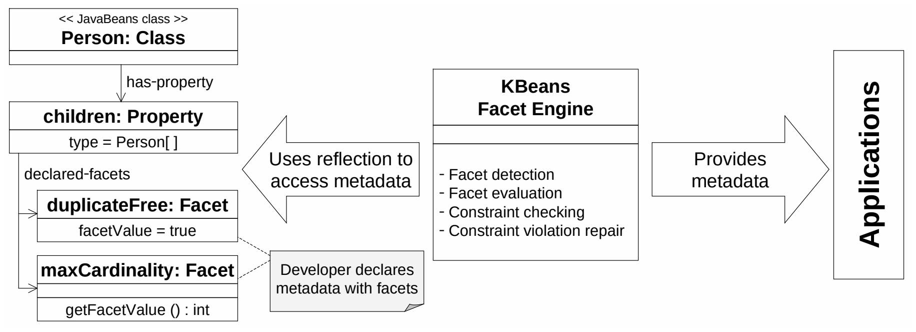


Fig. 2. The main idea of KBeans is to attach facets to JavaBeans properties, which can be accessed by means of reflection.

The following semi-formal definitions will lay an ontological foundation. Here, *property* is a property of a given JavaBean *bean* that is decorated with a facet.

facet:. (*bean* × *property* × *facet type*) → *facet value*

facet value:. A primitive Java value, an `Object`, or `null`.

facet type:. *facet name* → *type mapping*

facet name:. A Java identifier which is unique among the facet types that are installed in the Java Virtual Machine.

property type:. The `Class` of *property*.

type mapping:. *property type* → *facet value type*

facet value type:. The `Class` of the facet value or `null`.

constraint type:. A *facet type* with a *validation method*.

validation method:. (*value* × *facet value*) → `boolean`

In other words, a *facet* declares one Java value for a given property of a given bean. If this value is `null`, then the facet is treated as *not* declared. The class and semantics of the facet value are defined by the *facet type*. Each facet type has a unique name and a *type mapping* function, that derives the type of the facet value, given the type of the property. The result of the type mapping can be `null` to indicate that the facet type cannot be applied to the given property type. A *constraint type* is a facet type with a *validation method* which can check whether a given value would be a valid property value, given the bean's current facet value.

In order to add facets to a given JavaBeans class, KBeans supports two types of facet declarations:

—*Dynamic facet declarations* are methods that match the signature defined in subsection 4.1. These methods compute facet values dynamically, i.e. they may return different results for every bean with every method invokation.

—*Static facet declarations* are fields that match the signature defined in subsection 4.2. These fields store a constant facet value for all beans of the class.

4.1 Dynamic Facet Declarations

Let's assume we wish to declare a facet of the facet type with the name *facetName* for a property called *propertyName* of the `Class C`. The facet is declared dynamically, if `C` has a method of the following signature

```
public <facetValueType> get<PropertyName><FacetName>()
```

where *facetValueType* is the result of the facet's type mapping for the given property type, and both *propertyName* and *facetName* are capitalized in conformity with the JavaBeans naming rules. The method's prefix can be `is` instead of `get`, if the *facetValueType* is `boolean`. The current facet value is defined by the result of the method call. In the example from the beginning of this section, the property name is `spouse`, the facet name is `invalidValues` and the facet value type is `Person[]`, because the type of `spouse` is `Person`. The facet value is the array of the values of the `children` attribute.

Since the method signature above is compatible to the naming conventions of JavaBeans *getter* methods [Sun Microsystems 1997], the facets themselves can be accessed and formally defined as JavaBeans properties:

Definition (Dynamic Facet Declaration). A *dynamic facet declaration* for a property is a readable JavaBeans property with the facet's value type and the name `propertyName + capitalize(facetName)`.

The fact that facets for a property are themselves properties has several consequences. We can read and (if a *setter* method is supplied) write the facet values just like other property values. This includes that we can use existing bean-based tools and methods such as GUI builders and persistence mechanisms to modify them. Furthermore, we can recursively define meta facets that constrain other facet values.

Another interesting issue with facets declared as methods is *inheritance*. *Design-by-Contract* [Meyer 1997] defines that class invariants must be inherited from superclasses, so that subclasses can only make the invariant more restrictive. Similarly, the component-based development methodology Catalysis [D'Souza and Wills 1998] recommends that every component should be open to extensions but closed to modification. In the case of constraining KBeans facets, this means that a method overloading an inherited facet method must return a value at least as restrictive as the result of the `super` methods. If we funneled all facet access through specific utility classes (such as the `FacetIntrospector` described in subsection 4.4), we could implement a mechanism that automatically invokes the super methods and delivers a list of facet values for a given facet type. However, in conformity with the usual semantics of inheritance in Java, where programmers can freely modify an object's behavior by overloading, we decided to put the responsibility for overloading facets into the hands of the programmer. In those cases where overloading is strictly not permitted, developers can define the facet-declaring methods as `final`.

Finally, it is important to note that facet methods – similar to *getter* methods in general – must be free of side-effects. In particular, they must not modify any variable beyond their local scope. The reason for this is that the facet methods can not make any assumptions about when they are invoked.

The extreme flexibility of dynamic facet declarations, which is due to their potential to perform arbitrary computations within their method bodies, is at the same time a blessing and a curse. Facet access is expensive, because the methods have to be called each time a facet is being evaluated, and for every object individually. The methods do not provide any information about whether their return value has changed since the last call. Furthermore – and in many cases most importantly – their facet value is the result of a "black box" procedure that reduces its semantic transparency. It is impossible to formally reason or make any assumptions about the facet value itself at runtime. This also makes it barely possible to export the semantics to other formats such as XML Schema and OKBC. Covering the case that such a stronger notion of semantic transparency is needed, we introduce the concept of static facet declarations in the following subsection.

4.2 Static Facet Declarations

Let's (again) assume we wish to declare a facet of the facet type with the name *facetName* for a property called *propertyName* of the `Class C`. The facet is declared statically, if `C` has a field of the following signature

```
public final static <facetValueType> <propertyName><FacetName>
```

A rather executable definition follows.

Definition (Static Facet Declaration). Let `field` be the result of the call of `getField(propertyName + capitalize(facetName))` for the `Class` that owns a property. `field` is a *static facet declaration* for the property, if it is `public`, `final`, and `static` and has the facet's value type.

The (constant) value of this field is regarded as the facet value for all instances of the class. Similar to their dynamic counterparts, static constraint declarations that are "overloaded" by a namesake field in a subclass should ensure that the overloaded values are more restrictive than the values of the `super` fields.

4.3 Coding Conventions

Beside having a meaning in KBeans, both dynamic and static facet declarations can also be treated as normal fields and methods. Thus, the distinction between "normal" fields and methods and those that declare facets is rather blurred. However, semantic transparency requires that other developers and programs are supplied with clear, unambiguous facet declarations. Otherwise, programmers might not recognize existing facet declarations, or produce incorrect declarations by accident or due to misspelling methods. Furthermore, it can become relatively hard to perform *refactorings* [Fowler 1999] on properties, such as renaming, because facet declarations have to be changed together with the property. A solution to these problems could be to extend the Java specification by a new modifier keyword (e.g., `facet`) that would make the declaration of facets explicit. Since such an extension will not be available in the near future, we need to cope with an intermediate compromise. Most of the problems described above can be reduced by an automated facet declaration checker, which uses reflection to detect apparent facet declarations that are misspelled or have illegal value types. Apart from this tool, we propose several coding conventions that can make facet declarations more readable and reduce oversights.

The facet names should convey their semantics and make their role as a facet declaration clear. For example, the purpose of a method called `getLastNameMaxLength` is clearly accessible to readers of both source code and corresponding UML diagrams. Since all facet declarations contain a concatenation of property and facet names, they are easy to relate to their role. In order to make facets even more apparent, the names of the facet types could include predefined keywords, e.g. end with `_FACET`.

On source code level, it is common practice to conglomerate methods that cover similar aspects. In order to enable programmers to quickly browse a JavaBeans property and its facets, we propose to place getter and setter methods and the facet declarations above each other. The facet values should be precisely described with a commentary block, so that the semantics are transparent to developers from the automatically generated JavaDoc API documentations alone. Using the predefined @`see` tag of JavaDoc, properties should provide cross-references to the facets that are defined on them, as shown in the following method definition. This also allows to automatically detect oversights, because JavaDoc will report misspelled facet declarations.

```
/**
 * Sets the values of the children property.
 * @param values the new children
 * @see #getChildrenMaxCardinality
 * @see #isChildrenDuplicateFree
 * @see #isChildrenOrdered
 */
public void setChildren(Person[] values) { ... }
```

A technical weakness of the KBeans approach is that programmers could choose to add both static and dynamic facet declarations to the same class. Like facet overloading, such situations will normally occur infrequently, but they are technically possible. Our compromise here is to define that if both types are detected for a class, then the dynamic declaration has priority over the static one.

Note that all of these problems are due to the fact that we do cope with what is available in the Java language specification. If we extended Java with commands to make facets explicit, we could detect all of the problems already at compile-time.

4.4 Accessing and Processing Facets

Figure 3 displays a UML diagram of Java classes that can be used to access and process facets efficiently. An implementation of this *facet engine* is available from the KBeans Homepage [FAW Ulm 2000].

Facet types are defined by means of classes that implement the `FacetType` interface. Each `FacetType` has methods that can be used to check whether a facet declaration of the given type is admissible for a given property type and to compute the type mapping. If a facet type class also implements the `ConstraintType` interface, it must define a method to check whether a given value would satisfy the semantics behind the constraint. For example, we have defined a class `MinInclusiveCon-`

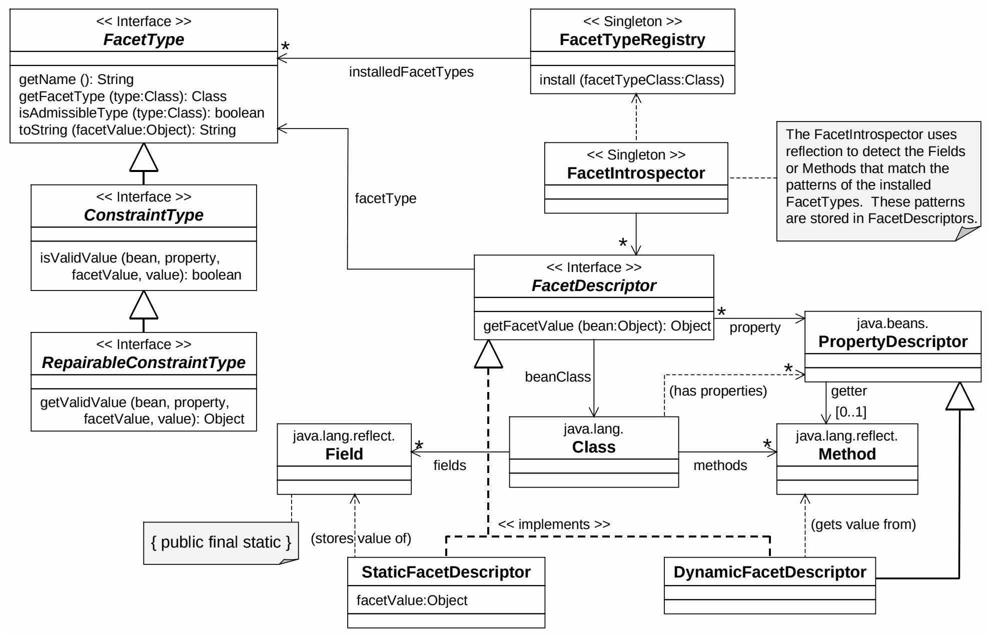


Fig. 3. The core classes of the KBeans facet engine.

`straintType`, which implements `ConstraintType` and which can check whether a given numeric value is greater than or equal to (`>=`) the inclusive lower bound defined by the facet value. This class furthermore implements the `Repairable-ConstraintType` interface to provide a "repair" method that derives a valid value from an invalid one.

The `FacetType` classes must be centrally registered in the `FacetTypeRegistry`, which maintains a list of facet types so that reflection-based classes can find out what to look for. KBeans supports a number of core facet types (see section 5), which are installed by default. However, applications are free to add individual new facet types, e.g. in static initializer blocks that are automatically executed when the JavaBeans class that uses the facet type is accessed for the first time.

The main functionality of the facet engine is accessible through the `FacetIntro-spector`. This static utility class uses reflection to detect the fields and methods that match the patterns of the installed static and dynamic facet types. The `Fa-cetIntrospector` associates the JavaBeans `PropertyDescriptors` with `FacetDe-scriptors`, which have links to the respective facet fields and methods and thus store the result of the reflection process for future access.

5. A CATALOG OF KBEANS FACETS

This section presents the catalog of the standard facet types installed in the `Facet-TypeRegistry` of KBeans. The goal of this catalog is to cover the most frequently needed types of metadata to support semantic transparency, while remaining as small and simple as possible. Since the XML Schema and OKBC languages have similar goals (cf. section 2), we decided to include all of their standard constraint types into KBeans, apart from a few exceptions which do not make sense in the context of Java.

The inclusion of these standards allows to use the KBeans approach to smoothly integrate semantically rich data structures from other languages into Java programs. *Data binding* [Reinhold 2001] involves compiling (XML) schemas or ontologies into Java classes, which handle the translation between external documents and objects. Existing data binding frameworks for XML Schema (e.g., *Enhydra Zeus* [Enhydra 2001] and *Castor* [Castor Project 2001]) generate Java source code – in particular, JavaBeans classes. The static facet declarations from KBeans can be used to improve data binding and even round-trip engineering [Knublauch and Rose 2000] of ontologies, XML Schema files and JavaBeans classes.

The KBeans constraint types furthermore embrace some elements of UML class diagrams [Booch et al. 1999], such as cardinalities and inverse relationships. These elements are seldom stated explicitly in Java code, so that reverse engineering from Java to UML is complicated.

A summary of this catalog is shown in table III. The references to Java types in the table and the rest of this document are as follows.

`<type>`. *any* type

`<simple>`. any non-indexed type

type`[]`. the indexed variant of *type*

`<primitive>`. any primitive type, or `String`

`<object>`. any non-primitive type, except `String`

`<numeric>`. {`byte`, `double`, `float`, `int`, `long`, `short`}, or the respective wrapper types (`Integer`, etc.)

`<comparable>`. a `<numeric>`, or a class implementing `java.lang.Comparable`

The facet names follow the usual naming conventions of Java, so that some XML Schema and OKBC constraint types have been renamed. The three XML Schema facets not implemented as KBeans standards are `whitespace` (which is only relevant for XML parsing), `totalDigits` (which can be expressed in terms of `maxLength` and `maxExclusive`), and `length` (which is the conjunction of `min-Length` and `maxLength`). Similarly, the OKBC facet `:CARDINALITY` is not implemented in KBeans, since it is simply an abbreviation of `:MIN-CARDINALITY` and `:MAX-CARDINALITY`. Note that it is possible to dynamically add these facet types to KBeans, by adding appropriate `FacetType` classes to the `FacetRegistry` from figure 3.

In the next subsections, each of the KBeans standard facet types is defined by the following schema.

`facet name`

Property types: The property types that the facet can be declared for. The types are represented by placeholders such as `<type>`.

Table III. The standard facet types of KBeans.

Facet Type	OKBC equivalent	XML Schema equivalent	Property types	Facet value type
defaultValue	-	-	<type>	<type>
maxLength	-	maxLength/totalDigits	<type>	int ($\geq$0)
minLength	-	minLength	<type>	int ($\geq$0)
fractionDigits	-	fractionDigits	float*, double*	int ($\geq$0)
maxCardinality	:MAXIMUM-CARDINALITY	maxLength/maxOccurs	<simple>[]	int ($\geq$0)
minCardinality	:MINIMUM-CARDINALITY	minLength/minOccurs	<simple>[]	int ($\geq$0)
validClasses	:VALUE-TYPE	-	<object>*	Class[]
invalidClasses	-	-	<object>*	Class[]
maxInclusive	:NUMERIC-MAXIMUM	maxInclusive	<comparable>*	<comparable>
minInclusive	:NUMERIC-MINIMUM	minInclusive	<comparable>*	<comparable>
maxExclusive	-	maxExclusive	<comparable>*	<comparable>
minExclusive	-	minExclusive	<comparable>*	<comparable>
validValues	:SOME-VALUES	enumeration	<type>	<simple>[]
requiredValues	-	-	<simple>[]	<simple>[]
invalidValues	-	-	<type>	<simple>[]
notNull	-	-	<type>	boolean
inverseProperty	:INVERSE	-	<object>*	String
equalProperty	:SAME-VALUES	-	<type>	String[]
unequalProperty	:NOT-SAME-VALUES	-	<type>	String[]
validValuesProperty	:SUBSET-OF-VALUES	-	<type>	String[]
requiredValuesProperty	-	-	<type>[]	String[]
invalidValuesProperty	-	-	<type>	String[]
pattern	-	pattern	<type>	String
ordered	-	-	<comparable>[]	boolean
duplicateFree	:COLLECTION-TYPE	-	<type>[]	boolean
validValuesNames	-	-	<type>	String[]

*Facet type is applicable to both simple and indexed type.

Facet value type: The type of the facet value. This value depends on the property type, and identical placeholders in both property and facet value types represent the same type.

Description: A textual description of the facet's semantics and typical application scenarios.

Validation method: If the facet type is a constraint type: Pseudo-code of a boolean expression or a mathematical expression (in *italics*) to describe the validation method. Note that for the sake of simplicity, not all cases (such as `null` values) are covered here.

Example: A brief example illustrating the facet. Most of these examples refer to the `Person` class from figure 1.

5.1 Default Value

`defaultValue`

Property types: `<type>`

Facet value type: `<type>`

Description: Specifies the default value of the property. This value should be equal to the property's initial value, and should not violate any of the property's constraints. Default values can be used to repair invalid values, and to implement "reset" functionality.

Example:

```
// The valid values of the maritalStatus property
public final static int UNMARRIED = 0;
public final static int MARRIED = 1;
public final static int WIDOWED = 2;

/**
 * The default maritalStatus of a Person is UNMARRIED.
 */
public final static int maritalStatusDefaultValue = UNMARRIED;
```

5.2 Length and Cardinality

`maxLength`, `minLength`

Property types: `<type>`

Facet value type: `int (0 <= minLength <= maxLength)`

Description: Specifies the maximum/minimum length of the result of the `toString` method of property values (`null` values are treated as empty strings). Most frequently, this will be used to constrain the length of `String` properties, or the number of digits of a number. A `maxLength` of `Integer.MAX_VALUE` is treated as infinite.

Validation method:

```
minLength <= ("" + propertyValue).length() <= maxLength
```

Example:

```
/**
 * The maximum length of a Person's last name is 17 chars.
 */
public final static int lastNameMaxLength = 17;
```

`fractionDigits`

Property types: `float, double, float[], double[]`

Facet value type: `int (fractionDigits >= 0)`

Description: Specifies the maximum number of digits of the fractional part of a floating point number.

Validation method:

```
double fraction = (propertyValue - (int)propertyValue);
("" + fraction).length() - 2 <= facetValue}
```

Example:

```
/**
 * A price can have at most two digits behind the dot.
 */
public final static int priceFractionDigits = 2;
```

`maxCardinality`, `minCardinality`

Property types: `<simple>[]`

Facet value type: `int (0 <= minCardinality <= maxCardinality)`

Description: Specifies the maximum/minimum cardinality of indexed property values (the cardinality of `null` arrays is 0). A `maxCardinality` of `Integer.MAX_VALUE` is treated as infinite.

Validation method:

```
minCardinality <= propertyValue.length <= maxCardinality
```

Example:

```
/**
 * A Person that is UNMARRIED cannot have any children.
 */
public int getChildrenMaxCardinality() {
    if(getMaritalStatus() == UNMARRIED) {
        return 0;
    }
    else {
        return Integer.MAX_VALUE;  // unlimited
    }
}
```

5.3 Type Restriction

`validClasses`

Property types: `<object>`, `<object>[]`

Facet value type: `Class[]`

Description: Constrains the type of valid property values. The facet values must be subclasses of the property type. This constraint is satisfied, if all values of the property are either instances of one of the types declared as facet values, or `null`.

Validation method:

```
forall v=propertyValues: facetValues.contains(v.getClass())
```

Example: (defined in `MalePerson` class)

```
/**
 * The spouse of a MalePerson must be a FemalePerson.
 */
public final static Class[] spouseValidClasses = {
    FemalePerson.class
};
```

`invalidClasses`

Property types: `<object>`, `<object>[]`

Facet value type: `Class[]`

Description: Constrains the type of valid property values. The facet values must be subclasses of the property type. This constraint is satisfied, if none of the values of the property is an instance of one of the types declared as facet values.

Validation method:

```
!(forall v=propertyValues: facetValues.contains(v.getClass()))
```

Example: (defined in `MalePerson` class)

```
/**
 * The spouse of a MalePerson cannot be a MalePerson.
 */
public final static Class[] spouseInvalidClasses = {
    MalePerson.class
};
```

5.4 Range by Comparison

`maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`

Property types: `<comparable>`, `<comparable>[]`

Facet value type: `<comparable>`

Description: Specifies the upper/lower bounds of the property value, with regard to `Comparable.compareTo`.

Validation method:

```
maxExclusive: propertyValue < facetValue
maxInclusive: propertyValue <= facetValue
minExclusive: propertyValue > facetValue
minInclusive: propertyValue >= facetValue
```

Example:

```
/**
 * A Person's (inclusive) minimum age is 0 (age >= 0).
 */
public final static int ageMinInclusive = 0;
```

5.5 Range by Enumeration

`validValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies all valid values of the property. This constraint is satisfied, if all property values are either equal to one of the facet values (using the `equals` method), or `null` if `null` is one of the facet values.

Validation method: $propertyValues \subseteq facetValues$

Example:

```
/**
 * The maritalStatus must be UNMARRIED, MARRIED, or WIDOWED
 */
public final static int[] maritalStatusValidValues = {
    UNMARRIED,
    MARRIED,
    WIDOWED
};
```

`requiredValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies a set of values that must be part of the set of property values. This constraint is satisfied, if all facet values are equal to one of the property values (using the `equals` method).

Validation method: $propertyValues \supseteq facetValues$

Example:

```
/**
 * The parents of the spouse must be among the guests.
 */
public Person[] getGuestsRequiredValues() {
    return (spouse == null)  ?  null  :  spouse.getParents();
}
```

`invalidValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies invalid values of the property. None of the property values may be equal to one of the facet values (using the `equals` method), or `null` if `null` is one of the facet values.

Validation method: $propertyValues \cap facetValues = \emptyset$

Example:

```
/**
 * A Person cannot be married to one of his/her children.
 */
public Person[] getSpouseInvalidValues() {
    return children;
}
```

`notNull`

Property types: `<type>`

Facet value type: `boolean`

Description: Declares whether `null` is a valid property value.

Validation method:

```
facetValue == false || propertyValue != null
```

Example:

```
/**
 * The lastName must not be null.
 */
public final static boolean lastNameNotNull = true;
```

5.6 Relationship between Properties

Whereas the above constraint types restrict property values by describing valid and invalid values directly, the following constraint types achieve this indirectly by declaring a relationship between the values of the property and those from another property. Since it is very complicating in Java to refer to other properties by objects (this would mean to return `PropertyDescriptors` as facet values), these constraint types declare properties by their names, i.e. as `Strings`. Beside simple references to other properties, they also use *property chains* (cf. the *slot chains* from OKBC). A property chain defines a list of property names `P[0]`, ..., `P[n]`, so that the values of the chain are the values of the `P[n]` property of the values of the `P[n-1]` property... of the values of the `P[0]` property, where `P[0]` is a property of the given class. For example, the values of the property chain `["spouse", "parents"]` are the parents of a person's spouse.

`inverseProperty`

Property types: `<object>`, `<object>[]`

Facet value type: `String` (name of a property of the class `<object>`)

Description: Declares the inverse property in the class that is referenced to by the property type. The property is specified by name. Satisfaction of this constraint means that if `ref` is a property value of the object `obj`, then `obj` is also a value of the inverse property of `ref`.

Validation method:

```
forall v=propertyValues: v.inverseProperty.contains(obj)
```

Example:

```
/**
 * The inverse property of children is parents.
 */
public final static String childrenInverseProperty="parents";
```

`equalProperty`

Property types: `<type>`

Facet value type: `String[]` (a property chain)

Description: Declares that the property values must be equal to the values of another property that is specified by a property chain. This constraint is satisfied if both sets of values are equal, whereby the single elements are compared with `equals`.

Validation method: $propertyValues = chainValueSet(facetValues)$

Example:

```
/**
 * A person's children are equal to the spouse's children.
 */
public final static String[] childrenEqualProperty = {
    "spouse", "children"
};
```

`unequalProperty`

Property types: `<type>`

Facet value type: `String[]` (a property chain)

Description: Declares that the sets of values of the given property and those of a specified property chain must not be completely equal.

Validation method: $propertyValues \neq chainValueSet(facetValues)$

Example:

```
/**
 * A person's parents must be different from the
 * spouse's parents.
 */
public final static String[] parentsUnequalProperty = {
    "spouse", "parents"
};
```

`validValuesProperty`, `requiredValuesProperty`, `invalidValuesProperty`

Property types: `<type>` (or `<type>[]` for `requiredValuesProperty`)

Facet value type: `String[]` (a property chain)

Description: Declares the valid/invalid/required property values by referring to the values of a specified property chain. The semantics are comparable to the related facet types `validValues`, `required-Values`, and `invalidValues`.

Validation methods:
`valid`...: $propertyValues \subseteq chainValueSet(facetValues)$
`required`...: $propertyValues \supseteq chainValueSet(facetValues)$
`invalid`...: $propertyValues \cap chainValueSet(facetValues) = \emptyset$

Example:

```
/**
 * A person cannot be married to one of his/her children.
 */
public final static String[] spouseInvalidValuesProperty = {
    "children"
};
```

5.7 Lexical Pattern Matching

`pattern`

Property types: `<type>`

Facet value type: `String`

Description: Declares a regular expression in the notation of Perl which is used by both the `java.util.regex` package of Java 1.4 and the XML Schema facet `pattern` [World Wide Web Consortium 2001]. The constraint is satisfied, if the `toString` representation of each of the property values matches this expression (whereby `null.toString()` is `""`).

Validation method:

```
forall v=propertyValue: Pattern.matches(facetValue, "" + v);
```

Example:

```
/**
 * The zipCode must match the pattern "[0-9]{5}(-[0-9]{4})?".
 */
public final static String zipCodePattern =
                             "[0-9]{5}(-[0-9]{4})?";
```

5.8 Collection Type

`ordered`

Property types: `<comparable>[]`

Facet value type: `boolean`

Description: Declares whether the property values must be in ascending order (using the `Comparable.compareTo` method).

Validation method:

```
!facetValue ||
forall i: propertyValue[i].compareTo(propertyValue[i+1] <= 0
```

Example:

```
/**
 * Children must be ordered by age (using Person.compareTo).
 */
public final static boolean childrenOrdered = true;
```

`duplicateFree`

Property types: `<simple>[]`

Facet value type: `boolean`

Description: Declares whether the property values may contain duplicates (using the `equals` method).

Validation method:

```
!facetValue ||
(forall i!=j: !propertyValue[i].equals(propertyValue[j])
```

Example:

```
/**
 * The list of firstNames must be duplicate-free.
 */
public final static boolean firstNamesDuplicateFree = true;
```

5.9 Textual Representation

`validValuesNames`

Property types: `<type>`

Facet value type: `String[]`

Description: Defines a textual representation of all the property value(s) declared with the `validValues` constraint. Each of the results of `validValues` must have one corresponding name element. For example, this facet can be used to define human-readable values of primitive constants for GUI elements (e.g., radio buttons and combo boxes). It can serve as a replacement for the missing support of enumeration types in Java.

Example:

```
public final static String[] maritalStatusValidValuesNames={
    "unmarried", // [0] = UNMARRIED
    "married",   // [1] = MARRIED
    "widowed"    // [2] = WIDOWED
};
```

---

6. APPLICATION SCENARIOS

KBeans has been successfully applied in several development projects [Knublauch et al. 1999; Knublauch et al. 2000], in particular from the domains of knowledge-based systems and multi-agent systems. In this section, we will demonstrate how KBeans technology can be used to *prevent*, *detect*, and *repair* invalid object states.

6.1 Interfacing Reusable Components

Graphical widgets such as Java's Swing classes are a very popular type of components. Comfortable off-the-shelf visual modeling tools allow to combine simple widgets like buttons and text fields to dialogs and other complex components. Their visual appearance and behavior can be easily modified by changing the components' properties. Visual modeling tools analyze the components' interface, detect the properties and their types, and provide respective editors generically. For example, the `JButton` class from the Swing library has a `verticalTextPosition` property, which specifies whether the button's text shall be aligned to the left, the right, or the center. However, although the `verticalTextPosition` property is of type `int`, only three values are valid for it (namely the predefined constants `LEFT`, `RIGHT`, and `CENTER`). Without metadata in the style of KBeans, visual editors are unable to reject invalid values which counteract the component builder's intended semantics. Adding the facets `defaultValue`, `validValues`, and `validValuesNames` to the `verticalTextPosition` property would significantly improve semantic transparency.

6.2 Ontology sharing

Beside relatively simple GUI components, applications can be composed of much more complex components. *Problem-Solving Methods (PSMs)* [Studer et al. 1998] are complex components that implement reasoning algorithms in knowledge-based systems (KBSs). A PSM provides an explicit specification of the knowledge types it requires, and the new knowledge it generates. Both input and output types are specified in *method ontologies*. Knowledge Engineering methods [Studer et al. 1998] suggest to specify those ontologies in formal languages, such as OKBC. However, an often overlooked fact with ontologies is that they are rarely disconnected from the application they are embedded in. In particular, there are strong links between domain ontologies and KBS modules such as user interfaces (the user interface alone typically accounts for 30–50 per cent of the program code of KBSs [Bobrow et al. 1986; Myers and Rosson 1992]), and between method ontologies and the (Java) modules that implement the reasoning algorithms. [Fensel et al. 1999] describes an approach for implementing PSMs, which are specified in a formal language called UPML, in Java. Similar to our own previous work [Knublauch and Rose 2000], they even propose to map ontologies to an ordinary class hierarchy (in a JavaBeans style). KBeans facets can be used to specify the semantics of these classes.

Agents [Jennings et al. 1998] are autonomous software components that are embedded in an environment with which they can interact flexibly. Agents provide a number of services and request services from other agents by exchanging messages, the content of which is based on ontologies, which are shared among multiple agents. An agreement on the semantics and constraints on the possible values of the ontological concepts is required to enable inter-agent communication. An agent receiving invalid messages can either reject them, or try to use metadata to "repair" them. We have implemented an efficient messaging approach based on the standard agent platform FIPA-OS [FIPA-OS 2001]. Here, agents share ontologies by using common JavaBeans classes. The single (distributed) agents can have individual implementations of these classes, i.e. they can add own methods or attributes, but agree on common constraints. (This can be implemented by pushing all the property access and facet code into abstract superclasses that can be overloaded with local classes.) Agents communicate by exchanging (FIPA) messages which contain JavaBeans objects that are converted to XML streams. On message receipt, the agents are able to validate the beans with the KBeans constraints.

Since the lack of smooth transitions between abstract knowledge models and an executable implementation is a major weakness of existing Knowledge Engineering approaches [Benjamins et al. 1997], efforts are made to bridge the gap between ontologies and object-oriented approaches (e.g., [Cranefield and Purvis 1999; Schreiber et al. 1999]). KBeans can significantly reduce this gap. The advantage of using KBeans for the implementation of PSMs and multi-agent systems compared to the use of formal ontology languages is that developers can *travel light* in the sense of Extreme Programming [Beck 1999], i.e. they can keep all structural information about the ontologies in Java classes. This allows to implement the interactions between ontologies and other modules very smoothly. Programmers can use the language they are used to and have all information in one and the same model, increasing development speed and reducing code duplication.

6.3 Constraint Checking

In the context of modern development processes, in particular light-weight methodologies such as Extreme Programming [Beck 1999], self-testing units are an essential means of producing quality software rapidly. The KBeans constraint types can be employed to monitor a given structure of JavaBeans instances to detect constraint violations at runtime. For that purpose, KBeans provides a `ConstraintManager` class, which can be plugged into an existing model of JavaBeans instances or de-activated, when the program is no longer in development stage.

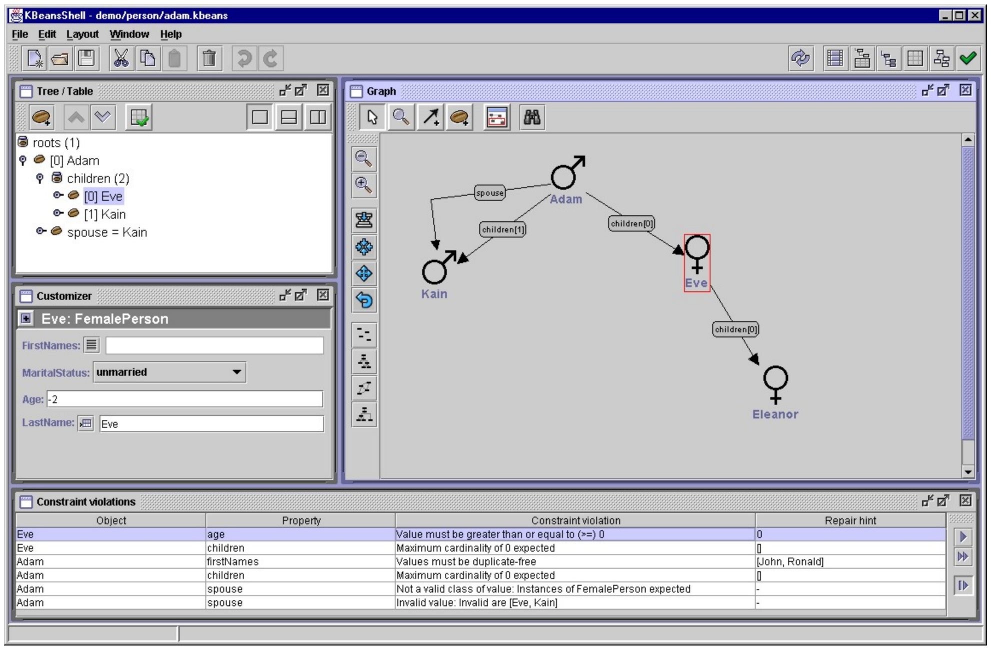
The `ConstraintManager` checks constraints whenever the value of any property in a given data structure has changed. Here, the fact that each change of a bound JavaBeans property produces a `PropertyChangeEvent` is used for two purposes. First, the `ConstraintManager` only needs a reference to a "root" object, from where it traces the links to all other objects that are 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 procedure. This is simply an invokation of the `isValidValue` methods of the `ConstraintTypes` that have been detected by the `FacetIntrospector`.

6.4 Data and Knowledge Acquisition

The metadata provided by KBeans facets can be employed by generic user interface components. These components could have labels or "tool-tip" texts that describe the expected values (e.g., "`Amount (10..100):`"), or change their background color if the user enters an invalid value. If a finite number of values is defined by a `validValues` facet, then a combo box or a group of radio-buttons can be used. If a `maxLength` facet is present, then a text field can automatically constrain the number of characters. All these components reduce development overhead, because programmers only need to explicate the object model's semantics, and the corresponding input dialogs or forms can be developed rapidly by visually plugging together generic components. The user interface code could even be generated automatically.

Driving the idea of generic components further, we have developed a tool called `KBeansShell` that serves as a graphical editor of any structure of JavaBeans instances (figure 4). With ontologies implemented as KBeans classes as described above, the shell has already proven to be very useful as a knowledge acquisition tool for various knowledge-based systems [Knublauch and Rose 2000]. It allows to display and edit beans by forms, trees, tables, or graphs, and provides generic mechanisms for persistence, clipboard commands, and undo. The open architecture of the `KBeansShell` allows to include application-specific components where needed. In particular, it obeys the JavaBeans standard, in which custom editing components for classes or properties (`Customizers` and `PropertyEditors`) can be defined.

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.



roots (1)
[0] Adam
children (2)
[0] Eve
[1] Kain
spouse = Kain

Eve: FemalePerson
FirstNames:
MaritalStatus: unmarried
Age: -2
LastName: Eve

Object	Property	Constraint violation	Repair hint
Eve	age	Value must be greater than or equal to (>=) 0	0
Eve	children	Maximum cardinality of 0 expected	[]
Adam	firstNames	Values must be duplicate-free	[John, Ronald]
Adam	children	Maximum cardinality of 0 expected	[]
Adam	spouse	Not a valid class of value: Instances of FemalePerson expected	-
Adam	spouse	Invalid value: Invalid are [Eve, Kain]	-

Fig. 4. A generic JavaBeans instance editor and knowledge acquisition tool.

The system even generates proposals for valid values, using the `RepairableConstraintType` interface and by identifying all objects in the knowledge base that would be valid values of a given property. Thus, the user is pro-actively guided through the data or knowledge acquisition process. Note that, in order to perform complex changes, the user is (at least temporarily) entitled to leave the object model or knowledge base in an inconsistent state.

The shell can also be used to examine the state of objects in an executing application. For example, in a knowledge-based patient monitor for anesthesia [Knublauch et al. 1999], 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 detected easily, reducing debugging overhead.

7. DISCUSSION

7.1 Related Work

The lack of an efficient constraint mechanism in Java has been a widely discussed issue for years. The huge demand for an appropriate Java extension has lead to the addition of a new `assert` keyword in Java 1.4 [Sun Microsystems 2000]. Here, an *assertion* is understood to be a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed. `assert` statements are useful to check constraint satisfaction at runtime, especially pre- and post-conditions in the sense of Design-by-Contract. The following code snippet shows how assertions – here in conjunction with KBeans constraints – can be used to check a pre-condition to prevent illegal value assignments.

```
public void setAge(int value) {
    assert FacetIntrospector.isValidValue(this, "age", value);
    this.age = value;
}
```

A major limitation of the `assert` statement is that it does not make explicit what type of contract it supports, e.g. there is no way to declare that a given boolean expression represents a post-condition. This problem is tackled by other approaches, such as *iContract* [Enseling 2001]. iContract uses a precompiler that converts code from predefined commentary blocks into executable Java code, as in the following example:

```
/**
 * @pre value >= 0
 */
public void setAge(int value) { ... }
```

This approach can increase the semantic transparency of classes and methods to humans, because the expressions from the commentary blocks will also appear in the API documentation. However, the need for an additional translation step and tools is a considerable overhead.

A different approach is taken by *jContractor* [Karaorman et al. 1999], where class invariants, and pre- and post-conditions can be specified by means of specific methods that match predefined design patterns very similar to the KBeans approach. For example, the following method declares the pre-condition of the `setAge` method.

```
protected boolean setAge_PreCondition(int value) {
    return value >= 0;
}
```

Calls to these boolean methods are inserted into the byte code by means of special class loaders and object factories.

All of these approaches are limited to constraint checking, i.e. they only evaluate boolean expressions, but do not provide any additional metadata about *what* they constrain. The only way to get more information about the constraints – and thus to increase semantic transparency for machines – would be to parse and analyze the source or byte code, which is in general far too complex to be feasible. Facet-based approaches such as KBeans can provide much more metadata, because they are declarative instead of imperative.

Instead of defining facets in special fields and methods like in KBeans, other approaches rely on runtime objects to represent facets. Okunev's [Okunev 2000] draft of a validation approach for JavaBeans properties makes use of the fact that the `PropertyDescriptors`, which are the result of the JavaBeans reflection process, can be decorated with arbitrary data items. Using the `setValue` method, metadata can be associated with predefined constraint types (e.g., `"maxLength"`). This approach is quite complicating to use, because the authors of constraints need to define additional `BeanInfo` classes, which are hard to read and (reverse) engineer. A similar approach is presented by McLaughlin [McLaughlin 2000], who employs

XML Schema files and data binding to express and read model semantics into Java objects. The constraints are translated into one `Constraint` object for each XML Schema type. The `Constraint` class has methods such as `getMinInclusive` that return the facet values defined in the schema. This approach, however, relies on external schema files, and is thus only an option if the object models are specified in XML Schema. Both approaches are also limited to static facet declarations, whereas dynamic KBeans facets can evaluate anything that can be expressed in Java.

The approach presented in this document integrates and extends our previous work from [Knublauch and Rose 2000] and [Knublauch et al. 2000], especially by relating it to XML Schema and OKBC, by the concept of static facet declarations, and by a more flexible facet engine.

7.2 Benefits and Limitations of KBeans

KBeans is a highly pragmatic, simple, and natural extension of JavaBeans. Its simplicity lies in the use of clear design patterns that are easy to apply, comprehend, and evaluate. The fields and methods implementing the patterns are based on pure Java code and do not rely on external formats or declarations. Particularly, the metadata that provide semantic transparency remain with the class they describe, so that component exchange is facilitated. The constraint methods do not have any negative impact on system performance, because they only provide optional metadata.

KBeans is fully compatible to the Java specification, i.e. no changes to the compiler are necessary to use it. Only a small library of core classes is required to implement facet access (e.g., the facet engine from subsection 4.4). Since each of the facet declarations is optional, and their omission in a class means to apply Java defaults, any existing JavaBeans data structure is backwards compatible to KBeans. Furthermore, KBeans can be flexibly extended by custom facet types. The space of potential application scenarios of facet metadata is vast.

Applying KBeans technology in projects only requires a minor adaptation of the programming style. Whereas programmers would usually implement the semantics of their models implicitly, they need to get used to making constraints and other metadata explicit. This is not necessarily an overhead, because all of the relevant metadata that can be explicated in facets are usually implemented somewhere in the code anyway. For example, the names and valid values of the `maritalStatus` property from figure 1 are typically implemented in the corresponding GUI elements. Thus, enriching components and shared object models by facets is mostly a question of reorganizing the classes. In general, developers who increase semantic transparency by explicitly embedding the intended semantics of their classes into design and implementation artifacts (i.e. UML diagrams and Java code) integrate the goal of component reuse early in the development process (cf. [Bennasri and Souveyet 2001]).

KBeans can be supported by tools both at runtime and during development. At runtime, reflection can be used to support visual builders and other generic programming tools. During development and modeling, KBeans facets appear just like other class members, i.e. they can be visualized in UML diagrams, edited with off-the-shelf IDEs, and they can even be engineered in a round-trip style. We have implemented a round-trip engineering tool [FAW Ulm 2000] that can be used as a plug-in for the widely used Java IDE Forte. It is an extension of the KBeansShell (figure 4), in which the user is able to edit JavaBeans objects that represent JavaBeans classes, properties, and facets. The source code is automatically synchronized. We have defined constraints on these meta-JavaBeans, e.g. to prevent illegal class names and cyclic inheritance, and use these constraints to halt code synchronization while the class model is invalid.

The major limitation of KBeans, when faced with the problem of semantic transparency in general, is its applicability to JavaBeans only. JavaBeans must fulfill various attributes, in particular, they must be public, and have a list of public properties. This limitation does not concern reusable components and domain models, the goal of which is to declare explicit, sharable class structures, that are public by their very nature. By the way, similar mechanisms like KBeans could be used to define pre-conditions of methods by declaring constraints on the valid values of method parameters (e.g., `setAgeParam0MinInclusive`). Also, the concepts of KBeans are not limited to Java, but can be transferred to other reflective languages such as C#.

As stated above, any approach to adding semantic transparency to object models is necessarily a compromise between what is available and what is desirable in Java. A weakness of KBeans is that the distinction between facet declarations and other fields and methods is relatively weak. We have proposed tool support and coding conventions which help to resolve this problem. Another issue is the great proliferation of class members, which can make classes hard to read, understand, and change. A pragmatic approach to reduce the complexity of the resulting UML diagrams is to filter all public, final, static fields. However, our opinion is that facet declarations should be considered as an integral part of any JavaBeans class, just like (other) public method declarations.

8. CONCLUSIONS

In this document, we have introduced a simple and natural extension of the JavaBeans standard called KBeans. The main purpose of KBeans is to support the development of domain models and components which are semantically transparent to both humans and developers. KBeans achieves this by supporting the declaration and evaluation of *facets*, which provide metadata about the valid values of JavaBeans properties. Facets are implemented by fields and methods which meet predefined naming conventions. Object-oriented reflection is used to detect and access these fields and methods at runtime. The metadata provided by the facets can be used to prevent, detect, and repair invalid object states. KBeans supports a catalog of facet types, which embraces related standards such as XML Schema and OKBC. We have demonstrated the benefits of KBeans in case studies from the domains of component reuse, ontology sharing, software testing, and knowledge acquisition.

ACKNOWLEDGMENTS

The author would like to thank Martin Sedlmayr and Thomas Rose for fruitful brain storming sessions.

REFERENCES

Beck, K. 1999. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA.

Benjamins, R., Fensel, D., Pierret-Golbreich, C., Motta, E., Studer, R., Wielinga, B., and Rousset, M.-C. 1997. Making knowledge engineering technology work. In *Proc. of the 9th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE-97).* Madrid, Spain.

Bennasri, S. and Souveyet, C. 2001. Component reuse and component-based methodologies. In *International Workshop on Mechanisms for Enterprise Integration: From Objects to Ontologies (MERIT 2001), at the ECAI 2001.* Budapest, Hungary.

Bobrow, D. G., Mittal, S., and Stefik, M. 1986. Expert systems: Perils and promise. *Communications of the ACM 29,* 9, 880–894.

Booch, G., Rumbaugh, J., and Jacobsen, I. 1999. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, MA.

Castor Project. 2001. Castor XML data binding. `http://www.castor.org`.

Chaudhri, V., Farquhar, A., Fikes, R., Karp, P., and Rice, J. 1998. OKBC: A programmatic foundation for knowledge base interoperability. In *Proc. of the AAAI-98.* Madison, WI.

Corcho, O. and Pérez, A. G. 2000. Evaluating knowledge representation and reasoning capabilities of ontology specification languages. In *Proc. of the ECAI'00 Workshop on Applications of Ontologies and Problem-Solving Methods.* Berlin, Germany.

Cover, R. 1998. XML and semantic transparency. `http://www.oasis-open.org/cover/xmlAndSemantics.html`.

Cranefield, S. and Purvis, M. 1999. UML as an ontology modeling language. In *Proc. of the IJCAI-99 Workshop on Intelligent Information Integration.* Stockholm, Sweden.

Demeyer, S., Ducasse, S., and Tichelaar, S. 1999. Why unified is not universal: UML shortcomings for coping with round-trip engineering. In *Proc. of the Second Int. Conf. on The Unified Modeling Language (UML'99)*, B. Rumpe, Ed. Fort Collins, CO.

D'Souza, D. and Wills, A. 1998. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA.

Enhydra. 2001. Zeus. `http://zeus.enhydra.org`.

Enseling, O. 2001. iContract: Design by contract in Java. JavaWorld (February), `http://www.javaworld.com/javaworld/jw-02-2001`.

FAW Ulm. 2000. KBeans homepage. `http://www.faw.uni-ulm.de/kbeans`.

Fensel, D., Motta, E., Benjamins, R., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M., Plaza, E., Schreiber, G., Studer, R., and Wielinga, B. 1999. The Unified Problem-Solving Method Development Language UPML. ESPRIT Projekt 27169 IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web, Deliverable 1.1, Chapter 1.

FIPA-OS. 2001. FIPA-OS homepage. `http://fipa-os.sourceforge.net/`.

Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code.* Addison Wesley Longman, Reading, MA.

Gruber, T. 1993. A translation approach to portable ontology specifications. *Knowledge Acquisition 5,* 2, 199–220.

Jennings, N., Sycara, K., and Wooldridge, M. 1998. A roadmap of agent research and development. *Int. Journal of Autonomous Agents and Multi-Agent Systems 1,* 1, 7–38.

Karaorman, M., Hölzle, U., and Bruno, J. 1999. jContractor: A reflective Java library to support design by contract. In *Proc. of the 2nd Int. Conf. on Metalevel Architectures and Reflection (Reflection'99).* Saint-Malo, France.

Klein, M., Fensel, D., van Harmelen, F., and Horrocks, I. 2000. The relation between ontologies and schema-languages: Translating OIL-specifications in XML-schema. In *Proc. of the*

Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence (ECAI 2000), R. Benjamins, A. Gomez-Perez, and N. Guarino, Eds. Berlin, Germany.

Knublauch, H. and Rose, T. 2000. Round-trip engineering of ontologies for knowledge-based systems. In *Proc. of the 12th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*. Chicago, IL.

Knublauch, H., Rose, T., and Sedlmayr, M. 2000. Towards a multi-agent system for pro-active information management in anesthesia. In *Proc. of the Agents-2000 Workshop on Autonomous Agents in Health Care*. Barcelona, Spain.

Knublauch, H., Sedlmayr, M., and Rose, T. 1999. Knowledge-based decision support in an anaesthesia information system. In *Proc. of the ESCTAIC Annual Meeting*. Glasgow, UK.

Knublauch, H., Sedlmayr, M., and Rose, T. 2000. Design patterns for the implementation of constraints on JavaBeans. In *Proc. of the NetObjectDays2000*. Erfurt, Germany.

Maes, P. 1988. Issues in computational reflection. In *Meta-Level Architectures and Reflection*, P. Maes and D. Nardi, Eds. Elsevier Science Publishers, Amsterdam, The Netherlands, 21–35.

McLaughlin, B. 2000. Validation with Java and XML schema. JavaWorld (September-December 2000) http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html.

Meyer, B. 1997. *Object-Oriented Software Construction, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ.

Myers, B. A. and Rosson, M. B. 1992. Survey on user interface programming. In *Proc. of the Conf. on Human Factors and Computing Systems (CHI)*. Monterey, CA.

Okunev, V. 2000. Validation with pure Java: Build a solid foundation for the data-validation framework with the core Java API. JavaWorld, December 2000, http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-valframe.html.

Orso, A., Harrold, M., and Rosenblum, D. 2000. Component metadata for software engineering tasks. In *Proc. of the 2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*. Davis, CA.

Pree, W. 1997. Component-based software development – a new paradigm in software engineering? *Software – Concepts and Tools 18,* 4, 169–174.

Reinhold, M. 2001. Java architecture for XML binding (JAXB). Java Community Process (JSR 31), Working draft http://java.sun.com/xml/jaxb.

Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., van de Velde, W., and Wielinga, B. 1999. *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT-Press, Cambridge, MA.

Studer, R., Fensel, D., and Benjamins, R. 1998. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering 25*, 161–197.

Sun Microsystems. 1997. JavaBeans specification. http://java.sun.com/beans.

Sun Microsystems. 2000. A simple assertion facility. http://www.javasoft.com/aboutJava/communityprocess/review/jsr041.

World Wide Web Consortium. 2001. XML Schema. W3C Recommendation 2 May 2001, http://www.w3.org/TR/xmlschema-1.