Kent Academic Repository

Kent Academic Repository
Full text document (pdf)

Link to record in KAR

http://kar.kent.ac.uk/475/

Document Version

UNSPECIFIED

REGULAR PAPER

Implementing associations: UML 2.0 to Java 5

D. Akehurst · G. Howells · K. McDonald-Maier

Abstract A significant current software engineering problem is the conceptual mismatch between the abstract concept of an association as found in modelling languages such as UML and the lower level expressive facilities available in object-oriented languages such as Java. This paper introduces some code generation patterns that aid the production of Java based implementations from UML models. The work is motivated by a project to construct model driven development tools in support of the construction of embedded systems. This involves the specification and implementation of a number of meta-models (or models of languages). Many current UML oriented tools provide code generation facilities, in particular the generation of object-oriented code from class diagrams. However, many of the more complex aspects of class diagrams, such as qualified associations are not supported. In addition, several concepts introduced in UML version 2.0 are also not supported. The aim of the work presented in this paper is to develop a number of code generation patterns that allow us to support the automatic generation of Java code from UML class diagrams that support these new and complex association concepts. These patterns significantly improve the code generation abilities of UML tools, providing a useful automation facility that bridges the gap between the concept of an association and lower level object-oriented programming languages.

1 Introduction

With the continuing rise in the complexity of embedded systems (i.e. computer systems placed within domestic and industrial devices such as washing machines, motor vehicles, watches, etc.) developers and manufacturers often experience a significant increased difficulty in ensuring the correct operation and reliability of their systems. Although system verification techniques can be employed to provide additional confidence in the correct working of the systems, the creation of the formal verification models is time consuming and requires a high degree of expertise that is expensive and in short supply.

The design process for embedded systems moves from abstract high level descriptions (models), such as block diagrams, to low level specific implementations details, such as a microchip circuit diagram. This paper reports on work that forms part of an international project, ModEasy [24], which aims to produce integrated software tools for the development and verification of embedded systems.

Using Model Driven Development techniques (MDD [32]), the project intends to provide automated support for mapping embedded system designs on to both

Communicated by Dr. Perdita Stevens.

D. Akehurst (✉) · G. Howells
University of Kent, Canterbury,
Kent CT2 7NT, UK
e-mail: D.H.Akehurst@kent.ac.uk

G. Howells
e-mail: W.G.J.Howells@kent.ac.uk

K. McDonald-Maier
University of Essex, Colchester,
Essex CO4 3SQ, UK
e-mail: K.D.Maier@kent.ac.uk

low level implementations and system verification toolkits. There is a clear advantage in providing this kind of automated support – namely that there is a single specification written by the designer, which is used to both verify and implement the system, thus avoiding the significant problem of the introduction of differences between the verified and implemented versions of the system. This approach relies extensively on the necessity for the transformations from high level design specification to verification model[1] and high level design specification to implementation to be correct. The key point is that if we can prove that these transformations are both correct, then the model of the system that is verified will match the model of the system that is actually implemented.

One primary task within the process of providing MDD based tools is the production and implementation of models or meta-models (models of models, or the model of a modelling language). For instance, to produce our ModEasy toolkit we require meta-models for a design language, for a verification language, and for any of the implementation frameworks we intend to map the designs to.

Within the current work, the design and verification support tools are provided within the Eclipse IDE [15] which requires us to implement our models using the Java programming language [34]. The Java programming language is not (and was never intended to be) a modelling or meta-modelling language. The Object Management Group's (OMG) [26] Meta Object Facility (MOF) [27] has, however, been designed specifically as a set of concepts with which to define (or model) other languages. Therefore, in order to best support our needs to define meta-models of a system design language, verification and implementation frameworks, we chose to use the MOF as the language for defining those meta-models. Of course this then leads us to the issue of how to implement models in Java that are defined using MOF. There are a couple of choices regarding a general implementation approach:

- To implement the models directly in Java (using an approach such as EMF [16])
- To implement the models as instances in a meta-model repository (using an approach such as MDR [23])

Both of these choices have their advantages and drawbacks. Use of a meta-model repository provides a quick way to provide support for a given model. One needs only to provide a specification of the model (possibly using XMI [31]) and the meta-model repository can be used as a repository for instances of the given model. Another advantage is that the meta-repository could easily provide support for serialising and un-serialising instances of the given model in a standard format (again potentially XMI). The drawbacks of this approach are efficiency issues introduced by the meta-level objects that represent each of the given model objects, and the consequent reliance on the (probably third party) meta-model repository itself. This latter issue is partially resolved by the standardization of the Java Metadata Interface (JMI) [17] that forms part of the libraries issued with Java 5, which provides a standard interface for MOF repositories, i.e. JMI (version 1.0) defines a Java mapping for the MOF (version 1.4).

The direct implementation approach requires more effort up front (i.e. specifying the mappings from modelling concepts to programming concepts, as addressed in this paper); additionally serialising model instances is more complex. The advantages of the direct approach are that there is no reliance on a third party library, and if you have control of the code generation templates you can have complete control of the implementation patterns and hence some control over the efficiency choices.

Even though it is not desirable practice and a good round trip engineering tool should reduce the problem, it is still the case that code implementations and design models can easily become separated. Considering this, it is quite important that any code that is automatically generated from a model be produced in a way that is easy to understand. Thus we have two main goals for code generation from models:

1. Readable code
2. No use of bespoke or third party libraries

Even though the modelling community has been using MOF and UML [30] based languages for several years, and there are a number of tools that generate Java code from UML models, there are still some significant issues regarding how to implement some of the concepts (especially due to the introduction of some new ones with the advance to UML 2.0). In particular, one of the major abstractions used in object-oriented (OO) modelling languages such as MOF and UML is the concept of an association. This concept does not exist in OO programming languages such as Java (or any of the other mainstream OO languages). Consequently, in order to generate code from a UML or MOF model it is necessary

---

[1] A verification model is a representation of the system design in a language that is suitable for input to a verification toolkit. For example, a potential target could be the language of timed automata, facilitating the verification (by model-checking) of timing properties of the system

Table 1 Evaluation of existing tools—part 1

	Rational Rose (8.3)	Octopus (2.0)	Poseidon (3.0)	Fujaba (4.2)	Omondo (2.0)
Field	Yes	Yes	Yes	Yes	Yes
Accessor	No	Yes	Yes	Yes, but no 'get' method for collections	Yes
Mutator	No	Yes	Yes	Yes	yes
Bi-directional	No	Yes	Partial	Yes	No
Multiplicities	Yes – no bounds check	Yes – can execute a bounds check	Yes – no check on bounds	Yes – no check on bounds	Yes – no check on bounds
Collection types	All as arrays	Yes	Orderd/unorderd as TreeSet/ArrayList	Orderd/unorderd as bespoke Fujaba types	Java types
Derived	No	No	No	No	No
Composition	No	No	No	No	No
Qualifiers	No	No	No	Yes, using bespoke fujaba types	Yes, using Map
N-ary associations	No	No	No	No	No
Subsetting	No	No	No	No	No
Derived Union	No	No	No	No	No
Redefinition	No	No	No	No	No

to devise a mapping from associations on to the chosen OO programming language.

The concept of an association is a complex abstraction facilitating a variety of specification variations all of which affect the semantics of the association. There are many modelling tools that claim to support code generation from UML models; however, none of the tools reviewed by the authors address the generation of code for the complete set of possible variations of an association specification. In particular, some tools do not correctly implement the bi-directional semantics of an association and most do not address generation from the more complex qualified and *n*-ary associations. Additionally, the new UML 2.0 set of standards has been recently released including some new concepts such as redefinition and subsetting which of course the tools do not yet support. Supporting code generation from UML to Java is made easier for some of these new features by the changes to the Java programming language introduced with Java 5.

This paper focuses on providing solutions to the issues of mapping qualified associations and the new (UML 2.0) semantic variations of an association into the Java 5 programming language. Section 2 provides a review of the association related code generation facilities of a number of UML based tools and a review of existing research that addresses the generation of code from associations and the semantics of associations. Section 3 gives an overview of the basic problem regarding the implementation of associations. Section 4 looks at how to implement UML 2.0 (unlinked) properties and in particular the more complex aspects of redefined, subsetted and qualified properties. Sections 5 and 6 subsequently look at how the implementation of uni-directional and bi-directional associations differs from the basic property implementation. We demonstrate the proposed implementation approaches using tools from the Kent Modelling Framework (KMF) as discussed in Sect. 7.

2 Background

There are many UML based tools that offer code generation features (for examples see the Tables 1 and 2). The code that is generated by the majority of these tools treats the UML model as something very close to a '*picture of the code*', i.e. there is not much conceptual difference between the low level generated code and the source model; in these tools the modelling concepts are effectively not much more complex than the concepts available in the target programming language. Although this can be useful, it is not quite as useful as providing support for converting high level, complex, modelling concepts into lower level code that represents a true implementation of them.

In Tables 1 and 2 we show the results of evaluating the association related code generation facilities of a number of major UML tools. A value of *Yes* indicates that the modelling concept is supported by the tool and use of it in a model correctly affects the generated code; and value of No implies that either the concept is not supported, or if it is, using it has no effect on the code generated.

Some of these tools allow you to specify your own code generation templates; however, they are evaluated

Table 2 Evaluation of existing tools—part 2

	EMF (2.02)	Artisan (5.0.22)	MagicDraw (9.0)	Visual UML (4.14)	Enterprise Architect (4.51)
Field	Yes	Yes	Yes	Yes	Yes
Accessor	Yes	No	No	No	No
Mutator	Yes	No	No	No	No
Bi-directional	Yes	No	No	No	No
Multiplicities	Yes, no bounds check	No	No	No	Yes, no bounds check
Collection types	Ordered/Unordered as bespoke types	All as arrays	No	No	User specifiable java types
Derived	No	No	No	No	No
Composition	Yes	No	No	No	No
Qualifiers	No	No	No	No	No
N-ary associations	No	No	Yes – as a class	No	No
Subsetting	No	No	No	No	No
Derived union	No	No	No	No	No
Redefinition	No	No	No	No	No

with what comes "*out of the box*"; additionally some of the tools perform differently when generating code for languages other than Java, but for the purpose of this paper they are evaluated against their Java code generation facilities.

The UML standard does not include a complete semantics of *n*-ary associations so it is not surprising that the tools do not support them. Also the last few concepts (subsetting, derived union, and redefinition) are concepts only introduced with version 2.0 of the UML, so it is not surprising that at the time of writing this paper they are not yet supported. However, it is disappointing that only three of the reviewed tools (EMF, Octopus and Fujaba) generate code that correctly supports the bi-directional semantics of an association; and only one of these (Octopus) supports the full complement of different collection types.

The Fujaba and Omondo tools are the only ones reviewed that provide some support for qualified associations; The Fujaba tool makes use of bespoke class libraries to provide the implementation and the Omondo tool uses the java.util.Map class, thus only providing facility to implement single qualifier values (see later section for details).

2.1 Implementing associations

In addition to tools, there is a significant amount of literature that addresses the meaning, or implementation, of UML associations. The most obvious reference for the semantics of associations and properties is the set of UML standards. This currently consists of four documents: UML Infrastructure, UML Superstructure, MOF Core and OCL [27–30], amounting to over 1,000 pages.

They give pretty complete information on the semantics of the concepts of Property and Association. These documents are an evolution of earlier standards which were less complete and thus inspired a fair amount of research, such as [33] which addresses some of the semantic issues of Associations and in [12] the authors discuss a number of transformations that aid understanding of the semantics of associations. In [5] the authors give a valuable discussion regarding the UML 2.0 semantics of overriding and redefinition with respect to operations. Unfortunately, even though these issues have been addressed to some extent, there are still problems regarding the semantic interpretation of some of the latest UML 2.0 concepts.

With respect to implementing associations in a programming language, there are some books that touch on the subject, however, none cover the issues completely. Larman in [21] suggests refining all associations into uni-directional associations and thus avoids the issue of implementing the more complex association types. In [7] the authors give a 'Composite' design pattern that is suitable for implementing composite associations. Fowler mentions the issue in [6] and gives correct code for implementing a one-to-many association; however, the implementation is limited to being 'read-only' and makes use of special "friend" operations that are used only for implementing the semantics.

From the research literature, [4] mentions the issue but gives no solution, [35] talks about implementing associations and suggests using a 'junction class' but does not really show how. Reference [25] suggests the use of explicit 'relationship objects' and gives a pattern for the implementation of bidirectional associations, recommending a master/slave approach; however, there is no reference to UML, *n*-ary associations or qualified asso-

ciations. Reference [18] addresses the issue of mapping associations to the Ada programming language, but do not address the more complex aspects such as *n*-ary and qualified associations.

The most extensive work has been carried out by Génova initially in his PhD thesis [9] with some of the work published with colleagues elsewhere [8,10, 11]. Their work gives solutions for implementing binary associations and looks at the semantics of all variations of pre-UML version 2.0 associations. In [14] the authors do not explicitly mention the bi-directional issue; however, they do discuss association classes and subtyping associations although not much detail is given on the actual code pattern employed to implement these concepts. Reference [22] specifically highlights the bi-directional issue and suggests an implementation approach that uses Java reflection and provides a variation to the Fujaba approach in an attempt to provide "less cluttered" code. Reference [1] addresses the new UML 2.0 concepts of subset, redefinition and union by a technique involving explicit Association classes and interfaces from the JMI [17] standard.

Additionally, some authors have addressed the reverse issue of reverse engineering Java code into UML models [13,20]; the work of [20] is quite extensive covering patterns of Java code that reverse engineer into a variety of association concepts including *n*-ary, bi-directional, qualified and association classes.

3 The basic problem

One of the most obvious differences between a UML model and code is formed using the concept of an association. Programming languages, such as Java, do not include the concept of an association and so a '*programming pattern'* must be employed in order to map one onto the programming language. The simplest of the solutions offered by many code generation tools is to implement each association end as an attribute in a class, with slightly more complex solutions providing *getter* (accessor) and *setter* (mutator) methods. This solution, although capturing the static structure of the model, does not correctly implement the semantics of an association, hence allowing invalid instances of the model to be created at runtime.

Much of the usefulness of the association concept comes from its bi-directional semantics and the wide variety of characteristics that can be specified for each association end. However, it is these two aspects that cause the complexity in mapping the concept onto a programming language such as Java.

The modelling concept of an association is composed of two or more association ends. Prior to UML/MOF version 2.0 the concept of association end was represented by an 'AssociationEnd', however, as of version 2.0 the concept has been merged with the concept of 'Attribute' to give the notion of a 'Property'. Classifiers (classes) contain a number of 'Properties' (previously 'Attributes') and 'Properties' (previously as 'AssociationEnds') can be linked using 'Associations'. This merge does make things simpler in that from a navigation point of view we need only be concerned with classes and properties, but from an implementation perspective we must remember that properties linked to form an Association have additional semantics.

A simple example to illustrate the concept involves a 1-to-1 association between two properties as shown in Fig. 1a. This figure defines two classes A and B and an Association; A has a property b of type B and B has a property a of type A; the two properties are linked by the Association. Figure 1b shows a similar situation with two classes A and B each having similar properties but without the properties being linked by an Association.

The difference between these two situations becomes clear when we look at potential instances of the classes and how they are linked. Figure 2 shows two sets of instances of four objects each, the dashed arrows indicate property values (consider them as Java object references). Figure 2a is a valid set of instances for either of the class specifications of Fig. 1, Fig. 2b, however, is only a valid set of instances for the class specifications of Fig. 1b. The semantics of the association in Fig. 1a states that the two properties A::b and B::a are linked, i.e. navigating from object a1 via the property b to a B object and the from that B object back to an A object via property a must result in the same object a1 that you started from. More formally this can expressed as a pair of OCL invariants:

context A   **context** B
inv :self.b.a=self   **inv** :self.a.b=self

One solution to the implementation issue is to implement this invariant (perhaps as a Java assertion, or a separate method) and allow it to be checked. However, this does facilitate invalid model instances to be
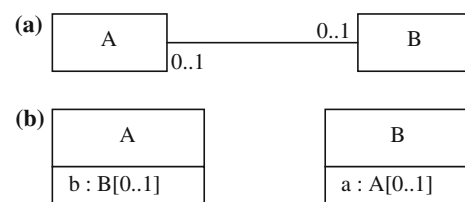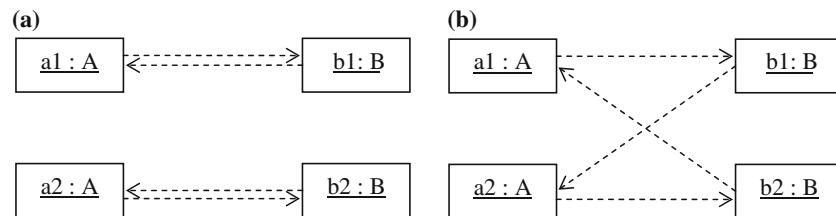

Fig. 1 Simple example

Fig. 2 Simple example instances

(a)

a1 : A ⇠ ⇢ b1 : B

a2 : A ⇠ ⇢ b2 : B

(b)

a1 : A ⇠ ⇢ b1 : B

a2 : A ⇠ ⇢ b2 : B

Table 3 Ensuring bi-directionality

```
class A {
  B b = null;
  public B getB() { return b; }
  public void setB(B b) {
    B old = this.b;
    this.b = b;
    if (old!=null) { old.setA(null); }
    if (b!=null && b.getA()!= this) { b.setA(this); }
  }
}
```

constructed and relies on the invariant actually being checked. Another, perhaps safer option, is to implement the model so that the semantics can not be broken; in fact, to implement these semantics it is fairly straightforward. The properties a and b can be implemented with accessors and mutators. To ensure that the bi-directional semantics of the association is kept true, we implement the mutator behaviour in such a way as to force the bi-directionality to be ensured (see Table 3).

However, this is only illustrating how to implement one variation of an association between classes A and B, there are many other variations; in order to correctly implement the semantics of all these variation, we first look at what the possible variations are (as of UML 2.0), then look at how to implement each of those variations separately, and finally consider how the mix of multiple variation options affect the combined implementation.

3.1 A property's properties

The implementation (discussed above) is for the simple case of the most basic 1-to-1 association. There are many adornments that can be added to the ends of an association (a property) which add complexity to their semantics and consequently to the implementation of the association. Each of the additional adornments that can be added to a property adds some complexity; additionally the combination of multiple adornments causes a certain amount of semantic " *feature interaction*", causing even greater complexity in the required implementation code.

Figure 3 below shows an extract of the UML 2.0 definition of a property with most of the direct and inherited properties of a property[2] shown. In order to correctly provide an implementation for a property we must take into consideration each of the characteristics of a property, and define how the various values of those characteristics affect the basic implementation of a simple property.

The basic properties of a property are its name, type and default value. A property can be owned by either a class or an association; if it is owned by a class then it is 'navigable', 'non-navigable' properties (association ends) are owned by an association. It is necessary to represent a non-navigable property, as characteristics such as a multiplicity may be expressed on it.

If a property has an opposite property, then it is part of a binary association and should have a bi-directional semantics with the opposite property. The lower and upper bounds indicate if the property represents a collection of objects of its designated type. The combination of isUnique and isOrdered indicate the type of the collection (Set, Sequence, Bag, or OrderedSet).

The isReadOnly property is self explanatory (the property can only be read from and not set to a new value) and a derived property is one whose value is calculated from an expression (defined as an attached constraint). If a property is marked as isDerivedUnion then it is considered to be read only and its value is derived from a strict union of all properties that subset it. A composite property indicates that the object owning the property acts as a container for the property value. An object can only be contained by a single container

---

[2] In future we refer to the properties of a property as property characteristics in order to avoid the confusion of discussing a property's properties

Fig. 3 UML 2.0 definition
of class 'Property'
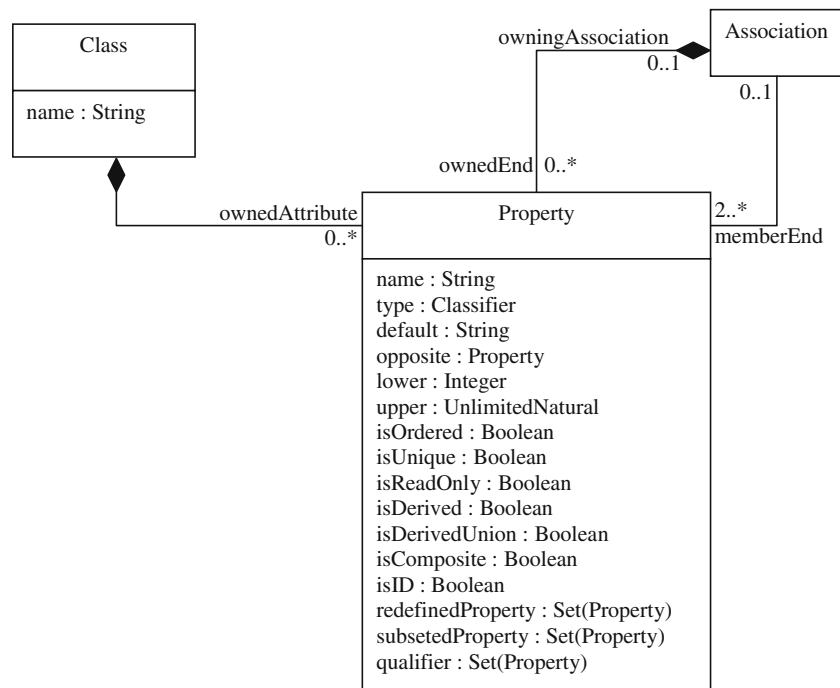
and if the container is deleted then so are its contents (contents can be removed from the container to avoid deletion); a composition defines a transitive asymmetric relationship – i.e. a directed acyclic graph. For example, the model shown in Fig. 3 shows that both a class and an association may act as the container for a property, however, only one may be the container for a single property at any one time.

If a property is marked isID then it forms an identifying property for the owning object, i.e. this property value should be unique for all objects of the owning class. The set of redefined properties of a Property indicate those properties that this property redefines; likewise, the set of subsetted properties indicates those property values for which this property value forms a subset. The set of qualifying properties are values used as qualifiers for the property value.

The following sections give further explanation on the semantics of these properties of a Property and we discus how to implement the semantics effectively in the Java OO programming language.

4 Unlinked properties

A basic unlinked property is semantically equivalent to a uni-directional association with source end multiplicity fixed as '0..*'. Implementing such properties is much simpler than implementing ones that are part of an association or are bi-directional. Many tools that offer code generation from UML models provide an accurate

implementation with respect to some of the characteristics (see Sect. 2). However, there are still many characteristics that complicate the implementation task, and which are not addressed by current tools. In this subsection we look at the variations to a baseline implementation that the property characteristics dictate.

4.1 Baseline

A basic implementation of a simple property Fig. 4 can be provided using a private attribute with accessor and mutator methods as shown in Table 4 for property b:B in a class A.

4.2 Target end: multiplicity, isUnique, isOrdered

The simplest change to the target end multiplicity is to make the value compulsory as opposed to optional, i.e. multiplicity '1' rather than '0..1' Fig. 5. Java references are always optional as they can take the value 'null'; if we specify that a property or association end is compulsory then we must ensure that its value is always set. The only way to do this is to ensure that a value for the property is a parameter to the constructor of the implementation class, and make sure that the mutator for the
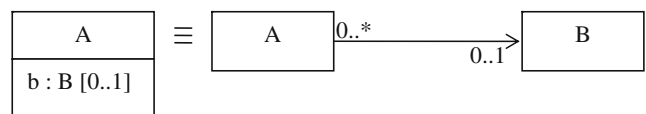
Fig. 4 Baseline

Table 4 Baseline

```
class A {
  private B b = null;
  public B getB() { return b; }
  public void setB(B b) { this.b = b; }
}
```


Fig. 5 Compulsory

property cannot accept the value 'null'. Similarly, if the upper bound is greater than 1 and the lower bound is greater than 0 we must ensure that a collection of values is passed 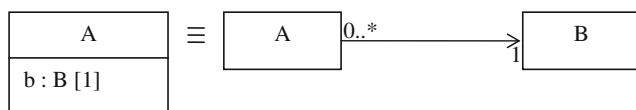to the constructor, and that mutators on the property do not allow the size of the collection to fall outside of these bounds.

In this respect, there is a virtual derived characteristic of a property, given by the expression "lower bound > 0", indicating whether or not the property value is optional. If a property value is not optional then there must be values for it passed to the constructor of the owning implementation class and the mutators must check that there are always value(s) for the property. The issue of implementing properties in accordance with the upper and lower bounds can be divided based on whether or not the property value is optional and whether or not the upper bound is greater than 1. The implementation can be altered to check that the property is never set to a null value by adding an assertion to the property mutator as shown in Table 5; an initial value for the property should be supplied via a constructor. The use of assertions does require that assertions are switched on by the Java compiler and runtime environment. An alternative is to provide an explicit conditional statement (if) with a corresponding exception to be thrown.

4.2.1 Collection type properties

To implement a property with an upper bound that is infinite (or greater than 1), we need to handle the issue of collections, and so we must also consider the isUnique and isOrdered properties. The effect of this on the basic pattern is to change:

- The type of the stored attribute
- The return type of the accessor
- The parameter type of the mutator

We can extend the previous code pattern to support this by making a decision on how to map the UML/OCL collection types onto Java classes. The Set, Sequence and Bag types map in a straightforward manner on to the Java Set, List and Collection classes; there is, however, no standard Java class that is equivalent to the UML/OCL concept of an OrderedSet. The closest option is a java.util.SortedSet, which does provide an ordering for its elements. We could use this interface, and provide the implementation of a java.util.Comparator based on insertion order. Given that Set and Sequence are mapped to java.util.Set and java.util.List, we would rather have OrderedSet mapped to a Java class/interface that extends both of these interfaces; unfortunately this cannot be implemented using an anonymous class, so we must provide an explicit implementation to provide such a type. Another option is to use java.util.Set as the interface, but provide an implementation based on java.util.List; this would provide a Set with a fixed ordering, though not give access via the java.util.List methods.

None of these options is totally satisfactory, we do not really want to provide explicit classes, but neither is the java.util.SortedSet an ideal solution. Ideally a java.util.OrderedSet would be provided by the standard Java libraries, however, until this occurs, we must go with one of the mentioned options.

The code pattern can be adapted to use the collection classes as illustrated in Table 6. With respect to the mutator body, it is important to clear and add all elements within the mutator, rather than setting the implementation field to a new value as this ensures that the selected implementation object for the collection doesn't change. This is particularly important if we implement the collection and override the collection mutators to either check that the bounds are not violated, or to make the collection read-only (see later sections in the paper).

4.2.2 Initial values

For basic collection-based property specifications, the initial value for the implementation should be an empty collection of the appropriate implementation class. The initial value provides the code for instantiation of an

Table 5 Compulsory

```
class A {
  B b = null;
  public B getB() { return b; }
  public void setB(B b) {
    assert b!=null : "Error: Property b is not Optional.";
    this.b = b;
  }
}
```

Fig. 6 Collection

Collection Type Properties
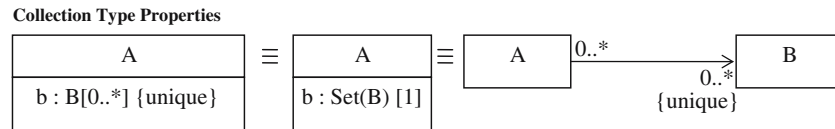
Table 6 Collection

```
  java.util.Set<B> b = new java.util.HashSet<B>();
  public java.util.Set<B> getB() { return b; }
  public void setB(java.util.Set<B> b) {
    this.b.clear();
    this.b.addAll( b );
  }
```

appropriate Java collection class such as "new java.util. Vector<B>()". Later sections of this paper give more complex definitions of initial values that handle bi-directional associations and read-only properties, by creating anonymous classes and overriding the collection mutator operations.

4.2.3 Checking the bounds

Lower and upper bounds that have fixed values greater than one can be supported by adding java assertion statements that check the bounds have not been violated. This does require that assertions are switched on by the Java compiler and runtime environment. The bounds check must be added to the mutators for the collection.

Many of the property characteristics require code changes to collection mutators. To implement these we propose a pattern involving two collections, one to store the collection values, which provides the backing collection for a collection object that is returned by the property accessor but which includes the appropriate mutator code. This is illustrated, including the bounds checks, in Table 7. Note that when using this code pattern, the lower bound checks must be switched off in the 'remove' mutator (via a flag) if we are replacing the entire collection.

Note: the error messages in the assertions should be more descriptive of which object and property bounds have caused the violation; the messages are shown here in a short form for brevity.

4.2.4 Undefined values

What is the value of a collection property if it has not been set? There are two potential answers: undefined or an empty collection.

The value of an unset non-collection property is, in Java, the value *null*, which we can equate with the OCL notion of *undefined*. However, in the context of collection-based properties, if we consider the OCL semantics of asking an undefined value if it is empty, which gives a value of *true*, there is an implication that for a collection object, the empty collection is equivalent to *undefined*. Thus we answer the above question such that the unset collection properties should be implemented as empty collections. It can be argued that equating the empty set with the notion of undefined is surprising; however, that is the semantic as defined by the standard.

4.3 Immutable properties: isReadOnly

If a property is read-only, we must ensure that the property cannot be assigned more than one value during the life time of its owning object Fig 7. One mechanism to do this would be to not provide a mutator, and necessarily require that a value is provided when the owning object is created. An alternative would be to allow a single setting of the value by the mutator (at an arbitrary time); implemented by including an assertion or test to check if the private implementation field has already been set, as illustrated in Table 8. This is, however, an unusual interpretation of the semantics and the first option is

Table 7 Collection with bounds check

```
java.util.Set<B> _b = new java.util.HashSet<B>();
boolean b_replacing=false;
java.util.Set<B> b = new AbstractSet<B>() {
  public int size() { return _b.size(); }
  public java.util.Iterator<B> iterator() {
    return new java.util.Iterator<B>() {
      java.util.Iterator<B> iter = _b.iterator();
        B current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return current; }
        public void remove() {
          if(!b_replacing)
            assert size()-1 >= lowerBound
                                : "Error: Property b lower bound violation.";
          iter.remove();
        }
    };
  }
  public boolean add(final B b) {
    assert size()+1 <= upperBound : "Error: Property b upper bound violation.";
    boolean result = _b.add(b);
    return result;
  }
};
public java.util.Set<B> getB() { return this.b; }
public void setB(java.util.Set<B> b) {
  assert b.size() >= lowerBound : "Error: Property b lower bound violation.";
  assert b.size() <= upperBound : "Error: Property b lower bound violation.";
  b_replacing=true;
  this.b.clear();
  this.b.addAll( b );
  b_replacing=false;
}
```

A	≡	A	0..*		B
b : B [0..1] {readOnly}				0..1 {readOnly}	
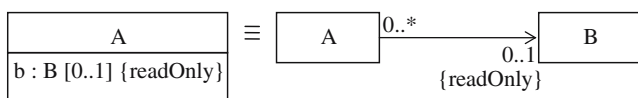
Fig. 7 Read-only

preferable. However, as you will read in a later section of the paper (Sect. 6), the second option could be useful with respect to implementing bi-directional associations.

The accessor for a readOnly collection type property must return a readOnly collection object. This can easily by supplied by wrapping the returned object using the java.util.Collections.unmodifiable(..) methods, e.g:

```
public B getB() { return java.util.
Collections.unmodifiableSet( this.b ); }
```

4.4 Derived properties: isDerived, isDerivedUnion

Implementing a derived property requires use of a pair of derivation expressions, one defining the accessor and

one defining the mutator. For simple derivation expressions, only the accessor would be required, and one could automatically deduce the corresponding mutator expression, however, in the general case both expressions would need to be given.

The UML 2.0 set of standards do not prescriptively define how these expressions should be specified, but indicate that the derivation expression can be given in a constraint. For our purposes, we assume additional characteristics 'accessorExpression : String' and 'mutatorExpression : String' that give the necessary expressions. These expression properties could themselves be derived by searching the appropriate sets of constraints for its definition in accordance with the suggestion in the standard, or perhaps both expressions could be given in associated tagged values. Irrespective of where the expression body specifications come from, the body expressions can be translated into Java (or specified directly in Java) and substituted for the accessor and mutator bodies in the template when the derived property is implemented.

Table 8 Read-only

```
  B b = null;
  public B getB() { return b; }
  public void setB(B b) {
     assert this.b==null : "Error: Property b is readOnly and already set.";
     this.b = b;
  }
```

A derivedUnion property is additionally readOnly, and hence we do not require the mutator. The accessor should return a value that is the union of the elements in all properties that subset the one being implemented. The subsetting properties are generally defined in sub-classes and hence the implementation of the derived-Union property can not be defined unless a property that subsets it is created. Thus, derived union properties that do not have a defined subset should return an empty collection (or undefined/null value). Generation of code for a derived union is addressed in a subsequent section regarding subsetted properties. Note that although the union operation is defined for all kinds of collection, the notion of subset is not clearly defined for kinds other than a Set (see Sect. 4.7).

4.5 Identifying properties: isID

The MOF 2.0 standard defines, as part of EMOF, a characteristic of Property called 'isID'. This Boolean characteristic indicates that the Property can be used to uniquely identify an object. The MOF standard specifies that only one property may be an identifying property. We believe this constraint to be too restrictive. Within the Relational Database community it has long been recognized that entities may require more than one characteristic to identify them, hence the specification of keys as collections of fields.

The UML standards do not impose any other constraints or semantics on the concept of an identifying property, and hence there is no effect on the code generated for a property marked as an identifying property.

It can be argued that an identifying property should be both compulsory and read only, irrespective of whether there are one or many identifying properties. The property should be compulsory because otherwise the owning object does not have an identity, and should be read only because otherwise an object can have its identity changed. If these constraints were to be imposed the impact on the implementation is to require that the identifying property be part of the object's constructor and that we do not provide a mutator.

In addition to these restrictions on the property itself, there is also the notion of uniqueness; the set of identifying values for an object should be unique (i.e. identify a single object). This essentially forms a constraint on the complete set of objects instantiated for a particular class. From the perspective of Java, this can not be directly implemented without utilising an object repository of some form. However, we can implement the Java notion of object equality to define equality between two objects if the values of their identifying properties are equal. This is achieved by basing the implementation of the 'public boolean equals(Object obj)' and 'public int hashCode()' methods on the values of an object's identifying properties.

4.6 Redefining properties: redefinedProperty

A property can only be redefined if its context is properly related to the context of the redefining property. This is formally and precisely defined in the Standard documents, amounting roughly to 'the class of the redefined property must be an ancestor of the class of the redefining property.'

Additionally, there is a constraint on the redefining property that it must be consistent with the redefined property; this too is formally and precisely defined in the standards, informally the definition is as follows, [28] Sect. 11.3.5:

A redefining property is consistent with a redefined property if the type of the redefining property conforms to the type of the redefined property, the multiplicity of the redefining property (if specified) is contained in the multiplicity of the redefined property, and the redefining property is derived if the redefined property is derived.

To implement a redefining property we must actually implement two properties, one of which is the redefining property, implemented as specified, the other is an implementation of the redefined property as a derived

Table 9 Redefinition

```
class C {
  D d = null;
  public D getD() { return d; }            // property d
  public void setD(D d) {
    this.d = d;
  }
  public B getB() { return getD(); }        // redefines property b
  public void setB(B b) { setD((D)b); }
}
```
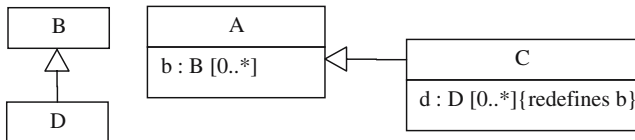

Fig. 8 Redefinition of collections

property whose accessor and mutator expressions reference the redefining property.

There are four different situations that affect the redefinition of a property, depending on the combination of whether the redefining and redefined properties are a collection type or not. The simple case where both are not collection types requires the derived property accessor and mutator to reference the redefining property's accessor and mutator, as is illustrated in Table 9. However, when one or both properties are collection types, the situation is slightly more complex.

4.6.1 Collection type redefining a collection type

If both are collection types, instinctively we would assume that we can do the same as for the situation where both are non-collection types; however, if we consider the redefinition specified in Fig. 8, the Java type java.util. Set<D> does not conform to the Java type java.util. Set<B> and hence the accessor "Set<B> getB()" cannot return a type Set<D> nor can we cast a type Set<D> to a type Set<B>.

The use of the Java generics wildcard based types [3] (e.g. "? extends B") would solve the type conversion problems, but we cannot use the 'add' method on collections defined using the wildcard. Instead to solve the problem we define the collection object that forms the initial value for the 'b' field to use the redefined property as the backing collection, as illustrated in Table 10.

If there are fixed upper or lower bounds, the constraint on the use of property redefinitions regarding the multiplicities ensures that if the redefining property bounds are met, then the redefined property bounds will also be met. There is of course the issue regarding properties

of different collection types that are in a redefinition relationship; however, it is not too complex an issue, the redefining property must be of a collection type that conforms to the redefined property collection type. The only difficulty is with respect to a Sequence (java.util.List) based collection type being redefined by an OrderedSet based collection type, depending on which of the previously discussed options has been chosen to implement OrderedSet based types.

4.6.2 Non-collection redefining a collection type

The constraint on the use of property redefinitions disallows the situation where a collection type property redefines a non-collection type; however, the reverse is allowed. To provide an implementation for this situation, we again use the initial value definition for the field and use the redefining value to construct a collection that is used as the source for the java.util.Abstract collection type; an example is shown in Table 11.

Note that once a property has been redefined, assigning it a value that is of the original type will cause a type consistency error, as the redefined property cannot be assigned the value of the original type.

4.7 Subsetting properties: subsettedProperty

If we look at the UML standards, there are two segments of text (along with a few more formal constraints) that help to explain the semantics of subsetting properties, one which is given in the context of Associations, [28] Sect. 11.3.1:

An end of one association may be marked as a subset of an end of another in circumstances where (a) both have the same number of ends, and (b) each of the set of types connected by the subsetting association conforms to a corresponding type connected by the subsetted association. In this case, given a set of specific instances for the other ends of both associations, the collection

Table 10 Collection redefining a collection

```
java.util.Set<B> b = new java.util.AbstractSet<B>() {
  public int size() { return getD().size(); }
  public java.util.Iterator<B> iterator() {
    return new java.util.Iterator<B>() {
      java.util.Iterator<D> iter = getD().iterator();
        B current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return current; }
        public void remove() { iter.remove(); }
    };
  }
  public boolean add(final B b) {
    boolean result = getD().add((D)b);
    return result;
  }
};
```

Table 11 Non-collection redefining a collection

```
java.util.Set<B> b = new java.util.AbstractSet<B>() {
  public int size() { return Arrays.asList( (getD()==null) ?
                                             new D[]{} :
                                             new D[]{getD()}
                                  ).size(); }
  public java.util.Iterator<B> iterator() {
    return new java.util.Iterator<B>() {
      java.util.Iterator<D> iter = Arrays.asList( (getD()==null) ?
                                             new D[]{} :
                                             new D[]{getD()}
                                  ).iterator();
        B current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return current; }
        public void remove() { setD(null); }
    };
  }
  public boolean add(final B b) {
    boolean result = getD()!=b;
    if (result) setD((D)b);
    return result;
  }
};
```

denoted by the subsetting end is fully included in the collection denoted by the subsetted end.

And the other given in the context of Properties, [28] Sect. 11.3.5:

A property may be marked as a subset of another, as long as every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property. In this case, the collection associated with an instance of the subsetting property must be included in (or the same as) the collection associated with the corresponding instance of the subsetted property.

The conditions for subsetting are very similar to the conditions for redefinition, however, the meaning is of course different. The subsetting property must always contain the same elements as, or a subset of, the subsetted property. Although the UML standard allows us to define subsets between pairs of collections of any type, the notion of 'subset' for any other than a Set type is not obvious. We illustrate in this paper how to implement

subset with respect to collections of type Set, leaving the other collection types for future work.

There are two situations to handle with respect to subsetted properties:

- When the subsetted property is a derivedUnion
- When it is not

If the subsetted property is a derivedUnion, we can implement the derivedUnion Property by overriding the accessor, to return a Union of all the properties that subset the derivedUnion, and we need not alter the implementation of the subsetting properties.

For subsetted properties that are not derivedUnions, it is slightly more complex. The subsetted property is mutable and thus should allow objects to be added to and removed from it. Objects that are removed from the subsetted property must be also removed from any subsetting property. Objects that are added to the subsetting properties should be also added to the subsetted property. The question is what to do about objects that are added to the subsetted property or removed from subsetting properties; as neither of these actions would invalidate the subsetting semantics, doing nothing would not be an incorrect implementation.

4.7.1 Non-set types

The subsetting semantics when both the subsetted and subsetting properties are not set types is pretty straightforward. If the multiplicities of properties 'b' and 'd' in Fig. 9 were [0..1], for d to be a subset of b it must either have the same value as b or the value null (where null or undefined is equivalent to the empty set). We would generate code to implement the mutators for property 'd' as shown in Table 12.

A value of null (or undefined) is considered to have a size of 0 and can thus be considered to be an empty subset of a single object value. In this example, if property 'd' is undefined but 'b' has a value then the value of 'd' is considered to be a subset of the value of 'b'.
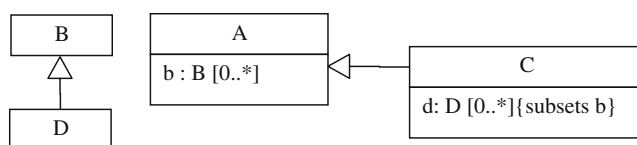
Fig. 9 Subsetting set types

4.7.2 Set types

If both subsetting and subsetted properties are sets, then we provide a specific implementation of the set type for both subsetted and subsetting properties. The 'add' method of the subsetting property must include code to add objects to the subsetted property and the 're-move' method of the subsetting property should include code to remove objects from the subsetting property. We can use a similar mechanism to that used to implement redefined properties, for example the code in Table 13 gives the implementation field for each property from the specification in Fig. 9 (the accessors and mutators are implemented as usual).

4.7.3 Non-set subsets set type

As with the concept of redefining properties, a non-set type property may subset a set type property. In this case we implement a mix of the two implementation situations given above; the remove method of the subsetted property must 'set' the value of the subsetting property rather than 'remove' items from it; and the mutator for the subsetting property must 'add' objects to the subsetted property.

4.8 Composite properties: isComposite

Composition is a notion of containment implying that objects can be 'part of' one (and only one) other object and also implies a relationship between the lifetimes of an object and its parts Fig. 10. The work of [2] gives a good explanation of the composition semantics, and the UML standards [28] state, Sect. 11.3.1:

> An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions define transitive asymmetric relationships – their links form a directed, acyclic graph.

To implement the notion of composition with the correct semantics, we require that:

Table 12 Subsetting non-set types

```
public void setB(B b) {
  this.b = b;
  if (b!=getD()) setD(null);
}

public void setD(D d) {
  this.d = d;
  if (d!=null) setB(d);
}
```

Table 13 Subsetting set types

```
/* subsetted property b:B [0..*] */
Set<B> _b = new HashSet<B>();
Set<B> b = new AbstractSet<B>() {
  public int size() { return _b.size(); }
  public java.util.Iterator<B> iterator() {
    return new java.util.Iterator<B>() {
      java.util.Iterator<B> iter = _b.iterator();
        B current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return current; }
        public void remove() { iter.remove(); getD().remove(current); }
    };
  }
  public boolean add(final B b) {
    boolean result = _b.add(b);
    return result;
  }
};
/* subsetting property d:D [0..*] {subsets b} */
Set<D> _d = new HashSet<D>();
Set<D> d = new AbstractSet<D>() {
  public int size() { return _d.size(); }
  public java.util.Iterator<D> iterator() {
    return new java.util.Iterator<D>() {
      java.util.Iterator<D> iter = _d.iterator();
        D current=null;
        public boolean hasNext() {return iter.hasNext();}
        public D next() {current = iter.next();return current; }
        public void remove() { iter.remove(); }
    };
  }
  public boolean add(final D d) {
    boolean result = _d.add(d);
    if (result) getB().add(d);
    return result;
  }
};
```
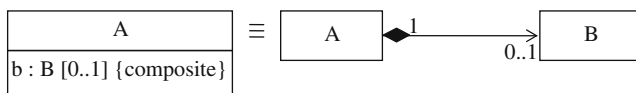

(a) An object is 'contained' by only one other object

(b) That if the container is destroyed, then all contained objects are also destroyed

Fig. 10 Composition

4.8.1 One and only one container

If a property is marked as composite, then this implies that the source end of the association must be defined as having a multiplicity of '1' or '0..1'. In fact the UML infrastructure defines a constraint (11.3.5 [3]) on Property that enforces this situation. Thus, when implementing composite properties we must also address the reverse multiplicity as discussed in Sect. 5 for uni-directional associations. For the composition semantics, it is necessary to check that no other object refers to (in this case) a B object (see Fig. 10) via a composite property. It is feasible to check that no other A object refers to a B object (see source end multiplicity on uni-directional associations in Sect. 5). However, to check that no other object of any type refers to a B object is not feasible (we would have to search all instances of all other types that refer to type B, how do we determine which other types refer to B? and it would potentially take too long to do this anyway).

To implement the composition semantics correctly, it is necessary to add information to an object's state that indicates its container and by which property it is contained. This kind of approach is provided by code generated using EMF [16]. One way to do this is to associate the composition semantics with an association defined on a class that forms the root type for all classifiers in a model; e.g. if the class Any shown in Fig. 11 is used as the root class, then properties that are marked as being composite properties are defined to subset the composite and part properties defined on the root class Any (in EMF there is a root class named 'EObject'). This almost, but not quite gives the correct semantics.

The composite property 'b' from class A in the specification of Fig. 10 is implemented as subsetting the property 'part' inherited from the root class Any; as each 'part' can only have a single 'composite' container object (see sections on bi-directional associations and subsetting properties), any B object that is set to the property 'b' of an A object will have a single 'composite' container.
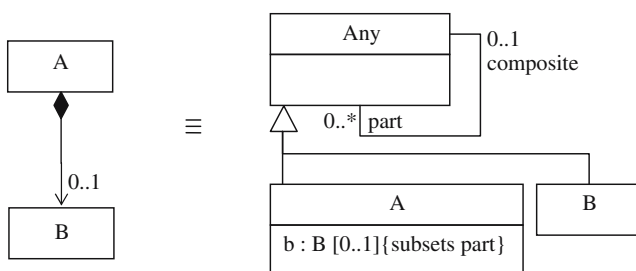
Fig. 11 Implementing composition

Note, this approach to the implementation does mean that we can set the composite or parts of an object directly, this could mean that, for instance, in the above example we could give an A object parts that are not accessible via the 'b' property. This is correct semantics for the notion of property subsetting, but could be considered incorrect for the semantics of composition. If we define 'part' and composite to be read only then we eliminate this issue, however, it then means that the semantics for bi-directional associations cannot be implemented (as this requires write access to each end of the association). Ideally we would be to treat the mutators for 'part' and 'composite' association as a special case, only giving write access to the bi-directional mechanism; however, this is not possible in Java.

An additional issue arises with respect to changing or resetting the values of a property that subsets 'part'. Such actions do not remove the old values from the set, thus leaving them as parts of the composite. Again, this is the correct semantic interpretation for 'subsets' but not for composition. Fortunately, this issue is solvable by using an alternative (almost identical) implementation of 'subsets' that does remove elements from the subsetting property if they are removed from the subsetted property.

4.8.2 Destruction of compositions

Modern OO programming languages such as Java make use of a Garbage collector, thus the notion of 'destroying' an object does not quite match; in order for a Java object to be destroyed, it is necessary to remove all references to it.

We can implement a 'destroy' method for an object that 'destroys' all object that are 'contained' by it, however, it is necessary to determine which other objects potentially reference the one being destroyed and set the references to 'undefined' (null) or remove them from the appropriate collections. This is not possible with respect to uni-directional associations, unless we actually implement a uni-directional association as a bi-directional association in order to keep track of all references, which would of course break the decoupling achieved with uni-directional associations (see discussion on uni-directional associations in Sect. 5).

One possibility would be to implement uni-directional associations using 'weak' Java references[3]. (which allow the garbage collector to collect them even if referenced). But this could still allow access to a 'destroyed' object in the gap between destroy being called and the garbage collector running. It would also require that at

---

[3] See standard java library package java.lang.ref

least one object holds a non-'weak' and thus non-uni-directional reference to every object.

The problem is that to correctly implement the composition semantics for uni-directional associations, it is necessary to determine what other objects hold a reference to the one being destroyed; we cannot do this directly in Java.

The solution we employ is to mark a destroyed object as destroyed, and then when it is accessed (via an accessor of a referencing object, or a collection iterator) we can check the referenced object, and remove it if it is marked as destroyed. The destroy method itself should mark its object as destroyed, call destroy on all objects that are referenced by composite properties and for any bi-directional associations we can directly remove this object from a referencing object's properties.

4.9 Qualified properties: qualifier

From the standards, [30] Sect. 7.11.4:

> A qualifier declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences. In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value.
>
> Note – The multiplicity of a qualifier is given assuming that the qualifier value is supplied. The "raw" multiplicity without the qualifier is assumed to be 0..*. This is not fully general but it is almost always adequate, as a situation in which the raw multiplicity is 1 would best be modelled without a qualifier.
>
> Note – A qualified multiplicity whose lower bound is zero indicates that a given qualifier value may be absent, while a lower bound of 1 indicates that any possible qualifier value must be present.

The latter is reasonable only for qualifiers with a finite number of values (such as enumerated values or integer ranges) that represent full tables indexed by some finite range of values.

4.9.1 Semantics and navigation

Figure 12 shows a qualified property; the interpretation of this is that each A object is connected to a set of tuples with type TupleType(x:X, b:B); there is optionally a single B object for each possible X object.

We can break down this specification into more primitive associations as shown in Fig. 13. Here we can see that the class A is associated with a set of tuples and for each tuple there is a compulsory X value and an optional B value. It is also necessary to ensure that there is only one tuple for each X value involved, this is enforced by the given constraint (there may also be X values that are not involved in the association).

Using this break down of the qualified property/association, we can see how to define the various possible navigation options, and thus see what accessors and mutators to define in an implementation for such a property.

There are two required navigation options corresponding to an association of this nature as shown below:

```
context A   self.b : Bag (B)
context A   self.b[x] : B
```

which map to the Java accessors: public Collection <B> getB() and public B getB(X x).

The second of these accessors returns a B object given an X object as the qualifying value and the first accessor returns a bag containing all associated B objects, irrespective of the qualifier.
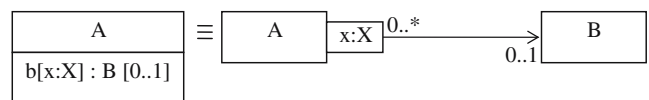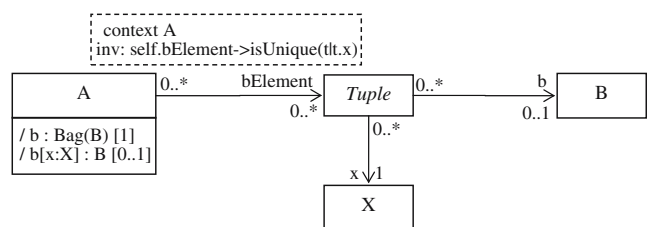

Fig. 12 Qualifier

Fig. 13 Implementing qualifiers

Table 14 Simple approach for qualifiers

```
class A {
  Map<X, B> b = new HashMap<X,B>();
  public Collection<B> getB() { return b.values(); }
  public B getB(X x) { return b.get(x); }
}
```

One easy mechanism to implement the option for qualified associations is to use the java.util.Map class. We could define the implementation as illustrated in Table 14.
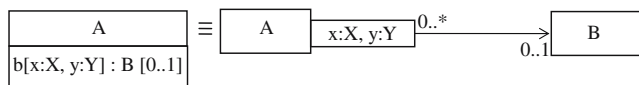
This implementation is only possible for properties with single qualifying values. It does provide the appropriate facility to relate a qualification value with each element in a collection. However, it does not enable us to use more than one qualifying value, such as the association given by the specification in Fig. 14. For this specification, we would want to create a Java type 'Map<TupleType(x:X, y:Y), B>'. Unfortunately there is no direct equivalent of TupleTypes in Java – an issue which we will come up against in a number of places with respect to implementing qualified associations, and is addressed later.

A potential solution to this would be to define a nested class that has properties for each qualifier value and use instances of that class as the Key to the Map based implementation.

4.9.2 Mutators

In addition to accessors for navigating a qualified property, for an implementation we also require mutators for setting the values of a qualified property. Obviously we want a mutator that enables us to set a value for a qualifier, i.e. public void setB(X x, B b). We could also provide a mutator that facilitates setting the whole collection of qualified values, i.e. public void setB(Set<TupleType(x:X, b:B)> b). However, we again here run into the issue of how to map TupleTypes into Java. If we are to provide this mutator it is essential that the elements of the collection are combined with associated qualifier values and hence we require a type for the elements of the set.

As before, a solution to providing a Java equivalent of a TupleType is to define an additional (nested) class

to hold the parts of the tuple; this time we require a class with a property for each qualifier value ('x:X'), and one for the primary property value ('b:B').

4.9.3 Tuples in Java

The previous two subsections have both illustrated a requirement for a representation of TupleTypes in Java. In both cases there is a solution by providing a new class. Separate classes are not an ideal solution, as it would mean that firstly many of these classes may have to be created (essentially "littering" the code with nested classes) and secondly they are not a true representation of a TupleType and assignment issues could arise between tuple objects that have the same type for each part but are implemented as different classes.

What is really needed is a good mechanism for mapping OCL Tuples into Java. This can then implement the qualified association as a Set of these tuples, facilitating multiple qualification values, and correct naming of the parts of the tuple.

Ideally we require the concept of an unnamed value type; which could be provided as something along the lines of a cross between a value type like the C/C++/C# struct and a Java anonymous class that could be used as shown in Table 15. We want to create tuple objects and define their fields and values at the same time (fields are always public). A comparison of tuple objects should compare the names and values of the fields; and access to the field values should be possible.

However, no such concept exists.

A potential equivalent would be to define a class 'Tuple' and use anonymous classes to create tuple objects; the Java reflection mechanism could be used to retrieve and compare each field defined in an object. If we wrap the reflection code into the common super class Tuple, we could then define tuples along the lines of :

```
Tuple t1 = new Tuple(){public final String
x="hello"; public final Integer y=1;};
```

Unfortunately, this does not work due to anonymous classes being package visible rather than public, i.e. the declared fields are not accessible from outside the package in which it is declared, even though the fields

Fig. 14 Multiple qualifiers

Table 15 Tuples in Java

```
Tuple t1 = new Tuple { int x = 1; String n = "fred"; };
Tuple t2 = new Tuple { int x = 1; String n = "fred"; };
Tuple t3 = new Tuple { int y = 1; String m = "fred"; };
Tuple t4 = new Tuple { int z = 2; String n = "jim"; };
boolean b1 = t1==t2;  //comparison should return true
boolean b2 = t1==t3;  //comparison should return false
boolean b3 = t1==t4;  //comparison should return false
boolean b4 = t1.x == t3.y; //comparison should return true
t1.x = 4; //Not possible - compile time error, tuples are immutable
```

themselves are declared public, and we cannot change the visibility of anonymous classes.

A working solution is to use the java.util.Map class as the basis for tuples. This unfortunately, means that we lose all compiler based type checking and we have to cast objects out of a tuple, but that is unavoidable as Java provides no direct mechanism for the implementation of Tuples. A tuple could be defined directly using the Map class and implementations of it, or we can wrap up the behaviour in a bepoke class named Tuple. The following shows examples of each of these approaches, using class initialisers to set values for the fields in the tuple:

```
Map<String, Object> t1 = new
HashMap<String,Object>(){{put("x","hello");put("y",1)}};
Tuple t1 = new Tuple(){{set("x","hello"); set("y",1);}};
```

By using the wrapper class Tuple, we can ensure that the tuple is immutable by making the mutator protected, and thus only useable within the class initialiser; however, using the Map class means that we need not introduce a bespoke library component.

4.9.4 Implementation

Returning to the issue of providing a generic mechanism for implementing qualified properties, we can implement a collection of Tuple (or Map) objects and provide behaviour in the accessors for extracting the required object. In fact, if we look back at Fig. 13 we can use this as a basis for implementing the qualified property specified in Fig. 12. We implement a field that is a collection of Tuples and implement derived properties as the accessors. For example, Table 16 shows an implementation for the specification of Fig. 12.

We do not, at this time, investigate variations to a simple qualified (unlinked) property. To fully support qualified properties, it is necessary to investigate the combination of qualifiers with the other property characteristics. This remains as future work.

5 Uni-directional associations

The previous section has shown how to implement unlinked properties, i.e. those that are defined directly as part of a class specification rather than those defined as part of an association. As illustrated these unlinked properties can be seen as having the same semantics as a particular style of uni-directional association, thus many of the issues regarding the implementation of uni-directional associations have been covered in the previous section.

When specifying uni-directional associations directly, there is however, the facility to specify some of its characteristics to give a different semantics to the unlinked properties discussed above. This section introduces those differences and addresses the related implementation issues.

If a Property is defined via an Association, the property 'opposite' of a Property gives a second property definition that refers to the object at the opposite end of the Association. The Property is owned by a Classifier if it is navigable, or owned by an Association if it is non-navigable.

In most situations, a uni-directional Association can be implemented identically to an unlinked Property (as illustrated in the previous section). The situation where this is different occurs when the specification of the source end of the association has characteristics defined on it that affect the semantics of the association. Many of the characteristics that are possible to define on a Property do not affect the semantics if the Property represents a non-navigable end. However, some do. The specification of source end multiplicity and source end qualifiers are addressed in the following sub-sections along with redefining and subsetting source ends.

The characteristics isOrdered and isUnique do not have any effect when applied to the opposite end of a uni-directional association. The isOrdered characteristic affects navigation, and the order in which a collection of objects is addressed, for a non-navigable property this is thus irrelevant. The isUnique characteristic applied to

Table 16 Improved approach for qualifiers

```
Set<B> b = new AbstractSet<B>() {
  public int size() { return bElement.size(); }
  public java.util.Iterator<B> iterator() {
    return new java.util.Iterator<B>() {
      java.util.Iterator<Map> iter = bElement.iterator();
        Map current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return (B)current.get("b"); }
        public void remove() { iter.remove(); }
    };
  }
};
public Set<B> getB() { // returns the collection of associated B objects
  return b;            // the collection allows removal but not addition of objects.
}
public B getB(X x) {
  for(Map t: this.bElement) {
    if (t.get("x").equals(x))
      return (B)t.get("b");
  }
  return null;
}
void setB( X x, B b) {
  B old = getB(x);
  if (old!=null)
    bElement.remove(new HashMap<String,Object>(){{put("x",x);put("b",old)}});
  bElement.add(new HashMap<String,Object>(){{put("x",x);put("b",b)}});
}
void setB(Set<Map> b) {    // This should also check that the parameter is a Map of
  this.bElement.clear();   // Tuples with the correct parts.
  this.bElement.addAll(b);
}
```

the non-navigable end would specify whether or not a single A object could reference the same B object more than once; it is not possible for this to be anything other than true as an object can only reference another object once by the same property anyway.

Defining the source end of a uni-directional association to be a composite part does not make sense, and neither would defining the source end to be an identifying characteristic. The read only and derived characteristics have no affect as we cannot 'read' or navigate in the reverse direction. Other characteristics do have a semantic affect and are addressed below.

5.1 Source end: multiplicity

The multiplicity of the source end of the baseline association is 0..*. However, it is acceptable UML practice to specify uni-directional associations with alternative multiplicities at the source end of uni-directional associations. In fact the OMG standards have a number of examples of one-to-one uni-directional associations. A '0..1' at the source end of such an association (Fig. 15) implies that only one A object may reference any single
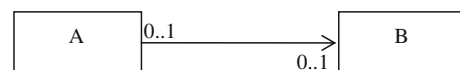
Fig. 15 Source end multiplicity—optional

B object. There is no primitive Java mechanism to enforce such a situation, and it is necessary to map a one-to-one association onto a primitive many-to-one plus an additional constraint. Figure 16 shows the more general case, mapping an association with fixed lower and upper bounds of the source multiplicity onto a primitive Java reference plus a constraint that checks the number of A objects referencing B object via the association.

Unfortunately there is no natural mechanism in Java that equates to the 'allInstances' operation. The only way in which we could implement the semantics correctly, without recording a collection of every object created for a particular class, would be to actually implement a bi-directional association (see later sections), but hide the reverse B->A link and use it only for implementation purposes.

Fig. 16 Source end
multiplicity—general
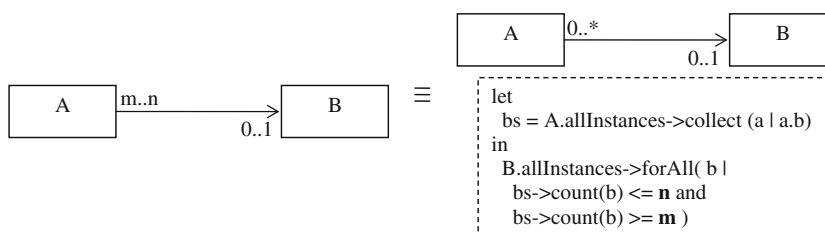
Although this gives a perfectly good mechanism for implementing uni-directional associations with a defined source end multiplicity, it does hamper one of the most useful modelling uses of a uni-directional association. Uni-directional associations are a great way to reuse classes from one package in another package that depends on the first. What is particularly nice about this is that the reused class need not know anything about how many or which other classes in dependent packages reference it. Uni-directional associations in this context provide a great decoupling point, potentially allowing one model or package to reference classes from another model or package that knows nothing about the referring elements, and potentially allows the referred and referring packages to be separately implemented, possibly by different implementers.

This issue raises a question regarding the purpose of making an association uni-directional within a particular model; is it in order to place a restriction on navigation; or is it in order to decouple parts of the model. Restricting navigation does not in general provide much advantage, other than placing limitations on what navigations may be performed by a user of the model; however, decoupling classes and increasing the separation between different parts of the model aides modularisation of the model and is a good practice.

Given this issue, it is arguable that, for the purposes of implementation in Java, all uni-directional associations should have the source multiplicity set to 0..*, or alternatively should be defined as bi-directional. However, if we were to assert this constraint, there would be a number of knock-on effects such as disallowing the concept of composition for uni-directional associations, which requires the source end to have an upper multiplicity bound of '1'.

As a consequence, in order to enforce the multiplicities we have three options:

1. *Disallow uni-directional associations.* This is undesirable as they provide a very useful decoupling mechanism in the context of modularising model specifications.
2. *Implement uni-directional associations as bi-directional.* This is not an entirely satisfactory option as it requires that the implementation of the model cannot be decoupled in the manner of the specification. It could be argued that the specification of the source end multiplicity is already breaking the decoupling.
3. *Provide an implementation of the 'allInstances' operation for all classifiers.* This would enable an implementation of the correct semantics for these associations, or to indicate when the semantics were invalidated; although for large models it would be quite an overhead as the set of all instances would need to be iterated over.

None of these options is a perfect solution, and it depends on the context in which the implementation is to be used as to which option is the best compromise. Option 1 is perhaps the easiest solution, but is arguably not very different to option 2. Option 2 is preferable if both ends of the association are to be implemented at the same time and it is not essential for the implementation to restrict the reverse navigation. If the decoupling is essential, and the performance hit of iterating over all instances is acceptable, then perhaps option 3 is the best choice.

Assuming there is a field 'allInstances' on the implementation of class A, the code in Table 17 shows a possible implementation pattern for the specification of Fig. 15 using option 3.

If we desire the decoupling offered by option 3, it is not possible to construct a B object with anything other than zero A objects referencing it. It is only after creating the B object that we can set A objects to refer to it and thus we cannot restrict the lower bound. If the upper bound is greater than 1, but less than infinite, we could count the number of A objects that already reference the B object and use an assertion to check the upper bound value. In the case represented by the code in Table 17, the mutator code for the b property looks for an A object that already references the b object passed to the mutator, and if one is found its b property is unset, thus at runtime only one A object should ever reference any single B object.

5.2 Source end qualification

To qualify the source end of a uni-directional association is perhaps an odd thing to do. However, it is a valid

Table 17 Source multiplicity

```
B b = null;
public B getB() { return b; }
public void setB(B b) {
  if (b != null)
    for(A oldA: A.allInstances)
      if (oldA.getB() == b) {
        oldA.setB(null);
        break;
      }
  this.b = b;
}
```

specification. The specification of Fig. 17 shows such a specification; the implementation would be equivalent to the implementation of a property whose type is a Set of Tuples containing parts x:X, y:Y and b:B.

5.3 Source end redefinition and subsetting

Redefining the source end of a uni-directional association seems at first glance to be meaningless, however, if the redefined property is navigable and part of a bi-directional association, then the redefinition has meaning. The example specification in Fig. 18 illustrates the situation.

The UML standard contains a constraint stating that "A navigable property (one that is owned by a class) can only be redefined or subsetted by a navigable property". ([28] 11.3.5[5]). However, there are several places within the specifications of the standard where this constraint is violated, in fact Fig. 73 of Sect. 11.3 contains an example.

If only the non-navigable end is redefined (or subsetted) then the semantics do not make sense. However, if the navigable end is redefined as well, then there is
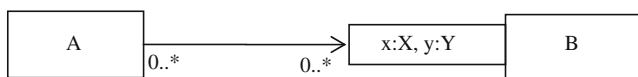
Fig. 17 Source end qualifier
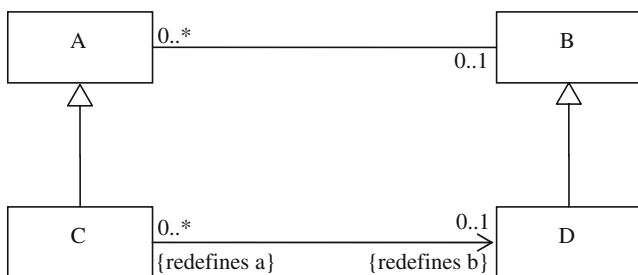
Fig. 18 A source end redefinition

no problem. The implementation of the property C::d should implement as though it were a bi-directional property with its opposite set to the property B::a. In fact this is the same semantics as the situation where only the navigable end is redefined, thus we can assume that the redefinition of the source end is merely informative.

6 Bi-directional associations

If one end of an association is non-navigable, then the navigable property should be implemented as described above for unlinked Properties and uni-directional associations. However, if both ends are navigable, then we must implement code that conforms to the bi-directional semantics of an association. Let us first look at what the bi-directional semantics are.

6.1 A simple bi-directional association

A one-to-one bi-directional association can be considered as two opposing many-to-one associations with two additional constraints that ensure that the bi-directional semantics are enforced. The bi-directional semantics requires that for any instance of the association, navigating from an object along that association and back again will leave you with the object you started from (or with a collection that includes the object you started from Fig. 19).

To implement this semantics, we could map the invariants into a Java assertion, which can be checked. However, it then falls to the user of the implemented code to call the checking code and to ensure that the properties are correctly set. A better solution is to implement the mutator code so that the constraint cannot be invalidated.

Fig. 19 Bi-directional
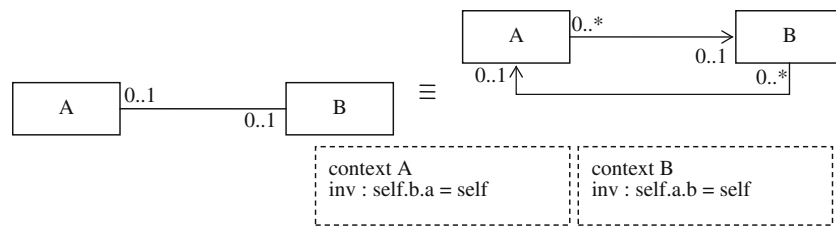
Table 18 Simple bi-directional

```
class A {
  B b = null;
  public B getB() { return b; }
  public void setB(B b) {
    B old = this.b;
    this.b = b;
    if (old!=null) { old.setA(null); }
    if (b!=null && b.getA()!= this) { b.setA(this); }
  }
}
```

Earlier in this paper we saw an example implementation for a bi-directional one-to-one association (such as that in Fig. 19), which is repeated in Table 18.

The difference between this code and that for a uni-directional association is the mutator body. There are three important parts to the implementation of this mutator:

1. Setting the field – this.b = b
2. Resetting the old opposite end's reference to this object – old.setA(null)
3. Setting the new opposite end's reference to this object – b.setA(this)

The other bits of code are simply there to support these three main statements. These main statements will vary depending on the characteristics of the opposite end. For instance, if the opposite end is a collection, then we need to add or remove elements from the collection rather than setting the property value.

The resetting and setting of the opposite end is not carried out in the property mutator in the case of a collection (i.e. a property with upper multiplicity > 1). Rather, the required code is included as part of the specification of the collection class that implements the property, which is defined as the initial value for the field implementing the property. The generated code is subtly different depending on whether or not the opposite end is a collection property or not. Code for setting the other end is similar, setting or adding to the other end the object 'this', but it must also include a check to see if the other end is already set to 'this' or an infinite loop is created at runtime – each end setting the other repeatedly.

6.1.1 Reuse

It should be noted that using this code pattern it is essential that the Java classes at both ends of the association are implemented in this manner. We cannot add bi-directional associations from a class after it has been implemented, i.e. we cannot *reuse* classes as part of bi-directional associations. Hence, if new bi-directional associations are added to a model, then classes at both ends must be re-implemented. This is contrary to one of the options for implementing uni-directional associations (see an earlier section).

6.1.2 ReadOnly

There are issues when we come to read-only properties that form part of a bi-directional association. It is necessary to have the mutator in order for the implementation of the bi-direction semantics to operate; however, if one end of the association is marked as read only, ideally we do not provide a mutator. Clearly, these two requirements are in conflict.

As has been described, with respect to unlinked properties, the value for a read-only property should be provided when an object is created. Thus, this immediately rules out the notion of a bi-directional property with both ends being read-only; we can not create two objects at once.

Providing a private mutator, would allow the read-Only property to be set by an object constructor, and thus we can employ the same mechanism of adding code to the mutator to enforce bi-directional semantics. However, consider what happens if the opposite (writable) end is set. In this case, the bi-directional semantics must

Table 19 Bi-directional collection

```
class A {
  ...
  java.util.Set<B> _b = new java.util.HashSet<B>();
  java.util.Set<B> b = new java.util.AbstractSet<B>() {
    public int size() { return _b.size(); }
    public java.util.Iterator<B> iterator() {
      return new Iterator<B>() {
        java.util.Iterator<B> iter = _b.iterator();
        B current=null;
        public boolean hasNext() {return iter.hasNext();}
        public B next() {current = iter.next();return current; }
        public void remove() {
          iter.remove();
          if (current != null)
            current.setA(null);
        }
      }
    }
    public boolean add(B b) {
      boolean result = _b.add(b);
      if (b != null)
        b.setA( A.this );
      return result;
    }
  }
  ...
}
```

be ensured, and thus the read-only end must also be set. However, it is read-only and hence cannot be set.

Given the conflict described above, we feel that the only semantically sensible option is to deny use of the read-only characteristic with respect to bi-directional associations. Two options were proposed earlier in the paper regarding the implementation of read only properties, either to not provide a mutator, or to add an assertion to the mutator that only allows a single mutation. The second of these options does provide a mechanism for implementing bi-directional read-only associations if they are required.

6.2 Collection type associations

For a bi-directional Association for which one or other end is a collection, we do not add the code for setting and resetting the other end to the property mutator. Rather, we add the code to the collection mutators defined in the initial value for the collection based property.

The initial values for collection properties are intended to provide a collection object that supports the implementation of the association. As such, we implement an anonymous class that is based on one of the java.util.AbstractCollection classes and override the mutator operations so that the correct association

semantics are adhered to. Below we show an example for the case of a Set based property (i.e. one that is unordered and unique), the other three variations of collection type can be implemented in a similar manner.

To instantiate a java.util.AbstractSet, one must provide implementations for the Set.iterator() and Set.size() methods. In this implementation, they are provided by a backing java.util.HashSet (i.e. the '_b' field from Table 19). Additionally, to facilitate mutation of the collection, we must provide implementations for the Set.add() method and its Iterator.remove() method. It is in these methods that we insert the appropriate code to set or reset the opposite ends of a collection based property.

If read only collection type properties are implemented using an *unmodifiable* wrapper in the accessor, we need not alter the implementation of the property, other than to add the appropriate assertion to the set method.

6.2.1 Compulsory bi-directional associations

A general issue is the specification of bi-directional associations with both ends defined as compulsory (i.e. having a lower multiplicity value > 1). For such a specification, a question arises as to how should the objects

at each end be constructed? We cannot construct two objects at once; therefore there must be a point in time where one end of the association is undefined. It can be argued that a constraint should be added to the specification of bi-associations that forbids both ends to be compulsory. In addition, when mutating the association ends, it is easy to cause a situation whereby an object with a compulsory property (at one end) is left with an undefined value. Implementation of the multiplicity constraints will indicate this situation to a user of the model. To avoid the problem, one must first destroy the old object at the non-compulsory end, and then set the new value. If the old object is not destroyed, it will end up with its compulsory property having no value. This requires a couple of changes to the implementation pattern. Firstly, we guard the multiplicity check with a test to see if the object has been destroyed; a destroyed object may well have properties with no value even if those properties are compulsory. Secondly, we alter the mechanism for stopping infinite looping. The former mechanism proves deficient when used in conjunction with compulsory properties. Instead we add a flag which checks if one end of the association is already in the process of being mutated and restricts further mutation accordingly.

6.3 Redefined and subsetting bi-directional associations

The concepts of redefinition and subsetting are heavily used throughout the specifications of the UML 2.0 superstructure and infrastructure documents. For specific examples see the 'Kernel Diagrams' in the UML 2.0 superstructure.

To implement a redefined or subsetting bi-directional association the same technique can be used as for uni-directional associations and unlinked properties. We must simply be careful to implement the setting and resetting other end code with respect to the *correct* other end of the redefined association. For example, if only one end of an association is redefined, the re-implementation of the redefined end must reference the opposite end of the redefining association not the redefined association. The same careful consideration is important with respect to subsetting.

6.4 Qualified bi-directional associations

Bi-directional qualified associations are perhaps one of the most complicated variations to implement. Before looking at the implementation options, let us first look at the semantics of a bi-directional qualified association.
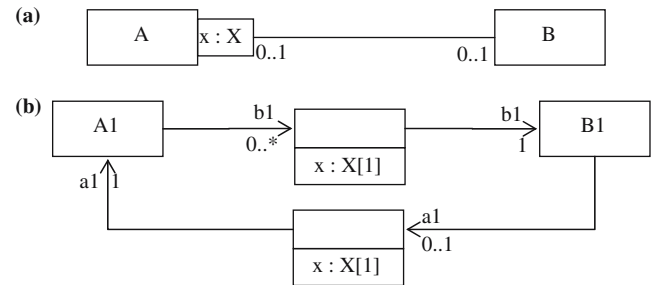
Fig. 20 Bi-directional qualifier

Figure 20a shows a qualified association between classes A and B. Specifically the property b:B, navigable to from objects of type A is qualified with a value of type X. Typically a qualifier is added to a property with a 'many' multiplicity in order to partition the set of elements, i.e. it is a little like providing an index for all the elements in the collection. In Fig. 20a there are a set of B objects associated to an A object, we can select any particular B object using a value of type X.

Without the qualifier, the same kind of structure could be modelled along the lines of that shown in Fig. 20b. The qualification affects the navigation along the association such that a value of type X is required. We can illustrate this using some OCL expressions that show the different navigation options and the type of the value returned by the navigation. We can also show the equivalent navigation that would be required for the model of Fig. 20b (without the qualifier).

```
context A    self.b : Bag(B)
  equivalent to
context A1   self.b1.b : Bag(B1)

context A    self.b[x] : B
  equivalent to
context A1   self.b1->any(p|p.x = x).b1 : B1

context B    self.a : Tuple(a : A, x : X)
  equivalent to
context B1   self.a1 : Tuple(a : A1, x : X)
```

As you can see from the OCL expressions above, the navigation types along a bi-directional qualified association are slightly different to non-qualified associations. In one of the directions, we have two options for navigation: to a set of B or to a single value, as is the case for a uni-directional qualified property discussed previously. In addition to the uni-directional case, we must provide support for navigation in the reverse direction, from the B object back towards an A object. These navigation

options must be reflected in the implementation of the accessors and mutators for each end of the association.

The forward direction code can be implemented in the same manner as for uni-directional associations, but including appropriate code for the resetting or setting the opposite ends. The code for these parts is slightly different in accordance with the type of object representing the other end of the association.

The UML 2.0 standards do not specifically state the intended navigation semantics for the reverse navigation along a qualified association, thus we discus the various options below:

1. To navigate in either direction along a qualified association the qualifier values must be given. If this navigation option is adopted, then we lose some of the directionality of the association.
2. Another option is that reverse navigation simply gives the object at the unqualified end, with no need to supply the qualifier values. This option is suggested in [9]. However, it may useful to be able to deduce the qualifier value for an object that is qualified.
3. A third option is that navigation from the qualified end of an association to the unqualified end returns a tuple object containing the unqualified object at the other end of the association plus the values which qualify the qualified end.

The two viable options are option 2 and option 3. Option 2 is the most useful if we consider the accessors, although it may be useful to determine the qualifier value from the qualified object,.It can be deduced by the expression self.a.b->any(t|t.b=self).x; the most frequent use of the back navigation is likely to be to access the object at that end rather than the qualifier values.

For the mutator, its parameter (i.e. the value being assigned) is normally of the same type as that returned by the accessor; if then for the example above, the mutator takes a object of type A as its parameter, we cannot keep the bidirectional semantics correct as we have no value for the qualifier when adding the B object to the qualified set of Bs. The solution is to insist that to mutate the reverse direction, we must provide an appropriate qualifier value in addition to the A object, i.e. option 3. We have two options for providing a mutator signature:

```
void setA (X x, A a) or void setA(Tuple a)
```

The first treats the qualifier values and A object separately. The second wraps them all up as a tuple. In our opinion, the second option is preferable as it distinguishes the mutator from a forward direction mutator for qualified properties. This choice does mean that the

mutator parameter and accessor return types are different, unlike the implementation of other properties. We must adjust the implementation of the code to set and reset the opposite end of the association to correctly take into account the qualifications. Table 20 illustrates the implementation for the specification of Fig. 20a.

As is the case with unlinked properties, we do not, at this time, investigate variations to a simple qualified bi-directional association. To fully support qualified associations it is necessary to investigate the combination of qualifiers with the other property characteristics. This, again, remains as future work.

6.5 Dual qualifications

What about the situation where there is a qualifier at both ends of an association? The UML standards do not give much information on this situation, rather they simply state that the situation rarely occurs.

If we are consistent with the policy adopted for navigating the reverse of a qualified association (as discussed previously) each set of qualified objects is a tuple consisting of the qualified values and a second tuple. This can be seen in Fig. 21. The set of qualified 'b' elements are tuples of type (Y,B) and in the reverse direction, the set of qualified 'a' elements are tuples of type (X,B). This gives us the following navigation options:

```
context A   self.b : Bag(TupleType(y:Y, b:B))
            = self.bElement.b
context A   self.b[x] : TupleType(y:Y, b:B)
            = self.bElement->any(b|b.x=x).b
context B   self.a : Bag(TupleType(x:X, a:A))
            = self.aElement.a
context B   self.a[y] : TupleType(x:X, a:A)
            = self.aElement->any(a|a.y=y).a
```

The implementation of such a specification requires the same approach as for single ended qualifications, except for the resetting and setting the other end code which must take into account the TupleTypes.

7 Tool support and testing of the proposed patterns

The coding patterns proposed in this paper have been thoroughly tested by providing a mapping from the UML based concepts onto the Java programming language using an Object Constraint Language (OCL) [29] based template language.

Table 20 Bi-directional qualifier

```
class A {
  ...
    java.util.Set<Map> _bElement = new java.util.HashSet<Map>();
    java.util.Set<Map> bElement = new java.util.AbstractSet<Map>() {
      public int size() { return set.size(); }
      public java.util.Iterator<Map> iterator() {
        return new java.util.Iterator<Map>() {
          java.util.Iterator<Map> iter = _bElement.iterator();
          Map current = null;
          public boolean hasNext() { return iter.hasNext(); }
          public Map next() { current = iter.next(); return current; }
          public void remove() {
            iter.remove();
            ((B) current.get("bElement")).setA(null);
          }
        };
      }
      public boolean add(final Map bElement) {
        final Map current = getBElement(((X) bElement.get("x")));
        if (current != null) remove(current);
        if (bElement.get("b") == null) return false;
        boolean result = _bElement.add(bElement);
        if (bElement != null)
          if (bElement.get("b") != null)
            if (A.this != ((B) bElement.get("b")).getA())
              ((B) bElement.get("b"))
                .setA( new Map(){{ put("x", bElement.get("x")); put("a", A.this);}}});
        return result;
      }
    };
    ...
}
```

(a)

(b)
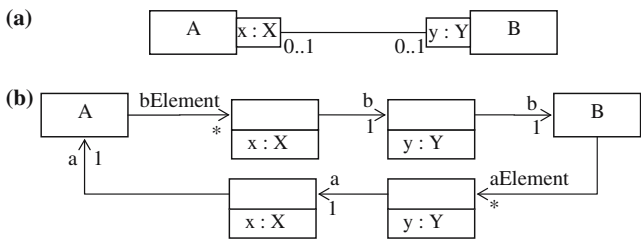
Fig. 21 Duel qualification

The templates are used in conjunction with KMF [19] based tooling to provide a code generator that implements UML 2.0 based models. A number of small models that test the generation of each individual property characteristics and some combinations of characteristics have been produced. In addition we have generated code for a number of larger scale models (or meta-models) related to the ModEasy project, and of course we have used the generation tool to attempt an implementation of the UML 2.0 Infrastructure, Superstructure and MOF 2.0 meta-models.

7.1 The initial goals

At the start of this paper we defined two goals for our code generation approach:

1. To generate readable code
2. Not to require bespoke or third party libraries

Are the code patterns proposed readable? Readability is a subjective issue and not easy to quantifiably measure. Working on the premise that less code is more readable than more code, we have attempted to minimise the amount of code required for each implementation pattern. However, this requirement is at odds with the second of our requirements; if we were permitted to define some bespoke library classes, we could improve the readability of the generated code and further minimise the size of the code patterns.

The patterns we have described in the core part of this paper do not use any library components other than those found in the standard Java libraries; excepting the use of an 'Any' class for supporting the composition

semantics, (which could be specifically generated for each model); and the preference for an 'OrderedSet' class.

There are however, a few issues which we have decided merit the definition and use of a bespoke library classes. Although we would prefer to use only the standard libraries, we have determined a number of reasons why we could forgo the second requirement in favour (primarily) of the first:

1. There is no 'OrderedSet' class defined in the standard Java libraries; therefore we must define our own. The paper has included some discussion on how to make do with the standard libraries; however, a bespoke class is considered by the authors to be the better option.
2. Use of a bespoke root inheritance class (e.g. 'Any'). This provides a useful facility for implementing the composition semantics, and further provides generic support for marking an object as destroyed.
3. Provision of a 'Tuple' class. The paper has shown how to manage without a bespoke class for tuples. However, provision of one, subjectively, greatly improves readability and aids in the support for 'destroying' objects. When a tuple object is used as part of the implementation for qualified associations, destruction of a 'part' of a tuple implies that the tuple should also be marked as destroyed.
4. Provision of alternative collection classes for 'Set', 'Sequence' and 'Bag' in addition to 'OrderedSet'. There are a number of reasons why we opted to do this:

   (a) The names of these classes better match the UML/OCL names than the versions in java.util, which improves comparison of the code with a UML model.
   (b) To support the removal of destroyed objects from collections, the use of a bespoke class means that the implementation can be removed from the generated code.
   (c) We can provide implementations of the operations defined on the OCL versions of the collection classes rather than being limited to the methods on the java.util classes.
   (d) The code patterns frequently involve redefining the iteration or mutation methods of a collection class; by defining our own classes we can provide more succinct and common (across all the collection types) mechanisms to do this.

7.2 Testing property characteristics and their interactions

There are sixteen characteristics defined on the class Property (as discussed in Sect. 3). It is unnecessary to test all combinations, as varying some of the characteristics has no significant semantic effect on others (e.g. changing the name of a property); and some combinations are not allowed by the definitions in the standard (e.g. redefining a compulsory property with a non-compulsory property). The following tables present the combinations of property characteristics that have been explicitly tested. We have duplicated many of the tests using both bespoke library classes and only the java standard libraries; in order to facilitate comparison of the two approaches, in particular with regard to evaluating readability Tables 21 and 22.

A number of characteristics are either denied or semantically nonsensical when used in conjunction with bi-directional association: read-only, derived, identity (which implies read-only). Currently, we have tested the combinations illustrated in Table 23. We realise that there are other characteristic combinations possible that we have not listed in this table.

7.3 Issues

This section discusses some of the issues regarding the generation of an implementation from model specifications, some of which are general issues and some are directly related to the UML Infrastructure model. After this discussion, we also highlight some areas for future work in this area.

7.3.1 Keyword name clashes

It is easy to cause name clashes between names in the model and keywords in the java language. This issue is easily solvable (in a code generation tool) by checking for keywords and using an adapted name where necessary. This typically occurs with respect to property names, the name clashes thus occur within the implementation of the property, the 'get' and 'set' methods don't clash with keywords as no Java keyword begins with 'get' or 'set'. Our tool adds an underscore character ('_') where necessary in order to avoid keyword name clashes.

7.3.2 Object method clashes

Java Method name clashes with methods on java.lang. Object. All Java classes extend java.lang.Object, if any

Table 21 Testing unlinked properties

Baseline	b : B [0..1]
Compulsory	b : B [1]
Collection	b : B [0..*]
Fixed upper bound	b : B [0..5]
Fixed lower bound	b : B [2..*]
Fixed upper and lower bound	b : B [2..5]
Read only	b : B [0..1] {read}
Read only compulsory	b : B [1] {read}
Read only collection	b : B [0..*] {read}
Derived	c : Integer, b : Integer {derived} get{return c; }set{c = c+b; }
Derived union	b : B [0..*] {union}, d : D [0..*] {subsets b}, e : E [0..*] {subsets b}
Identity	b : B [0..1] {id}
Redefines	b : B [0..1], d : D [0..1]{redefines b}
Redefines both compulsory	b : B [1], d : D [1] {redefines b}
Redefines redefining compulsory	b : B [0..1], d : D [1] {redefines b}
Redefines both read only	b : B [0..1] {read}, d : D [0..1] {read, redefines b}
Redefines collection	b : B [0..*], d : D [0..*] {redefines b}
Redefines collection with single	b : B [0..*], d : D [0..1] {redefines b}
Subsets	b : B [0..1], d : D [0..1] {subsets b}
Subsets set with set	b : B [0..*] {unique}, d : D [0..*] {subsets b}, e : E [0..*] {subsets b}
Subsets set with single	b : B [0..*] {unique}, d : D [0..1] {subsets b}
Composite	b : B [0..1] {composite}
Composite collection	b : B [0..*] {composite}
Qualified	b [x:X,y:Y] : B [0..1]

Table 22 Testing uni-directional associations

Fixed upper bound on source	a : A [0..5] $\rightarrow$ [0..1] b : B
Fixed lower bound on source	a : A [1] $\rightarrow$ [0..1] b : B
Fixed both on source	a : A [2..5] $\rightarrow$ [0..1] b : B
Qualified source	a[x:X,y:Y] : A [0..5] $\rightarrow$ [0..1] b : B

of the methods generated for the model have the same name as a method on java.lang.Object we get a method consistency issue; the model method is unlikely to be type consistent with the (erroneously) overridden method in java.lang.Object. The only solution to this is to forbid the names of operations or properties that would cause such a name clash.

Unfortunately just such a clash is present in the UML Infrastructure model; the property 'class' from model classes Property and Operation (in both Constructs and Basic) is mapped to a Java method 'getClass' which clashes with the method of the same name defined on java.lang.Object. Our solution was to alter the name of the property in the UML Infrastructure model, e.g. replacing 'class' with 'clazz'; an alternative would be to add more checks to the code generator in order to look for this and similar problems, and cause it to change the names in the generated code appropriately.

7.3.3 Redefinition, overriding and duplicate property names

Properties or operations with collection based return types, with the same name and which redefine, subset or override each other cause problems.

The model collections are mapped to Java collections using the Java generics feature to retain the collection element types. The difficulty here is a different semantic model regarding the super/sub type relationship between collections. The UML/OCL semantic model appears to assume a subtype relationship between Set(B) and Set(A) if B is a subtype of A; whereas the Java type Set<B> does not subtype Set<A>.

The reason for this in Java has to do with the fact that new items can be added to sets, and if this was allowed, then items not of type B (but of a different subtype of A) could be added to a Set<B> using an alias of type

Table 23 Testing bi-directional associations

Optional one-to-compulsory one	a : A [0..1] --------- [1] b : B
Optional one-to-optional many	a : A [0..1] --------- [0..*] b : B
Optional one-to-fixed upper and lower	a : A [0..1] --------- [2..5] b : B
Compulsory one-to-optional many	a : A [1] ------------ [0..*] b : B
Optional composite one-to-compulsory one	a : A [0..1] ♦-------- [1] b : B
Optional composite one-to-optional many	a : A [0..1] ♦-------- [0..*] b : B
Optional one-to-optional qualified	a : A [0..1] --------- [0..1] b[x:X,y:Y] : B
Optional one-to-optional one redefined	a : A [0..1] --------- [0..1] b : B
	c : C [0..1] ---------{redefines b} [0..1] d : D
Optional one redefined-to-optional one redefined	a : A [0..1] --------- [0..1] b : B
	c : C [0..1] {redefines a} ---------{redefines b} [0..1] d : D
Redefine fixed upper and lower	a : A [0..1] --------- [2..5] b : B
	c : C [0..1] ---------{redefines b} [2..3] d : D
Redefine composite many	a : A [0..1] ♦-------- [0..*] b : B
	c : C [0..1] ♦--------{redefines b} [0..*] d : D
Subset optional one	a : A [0..1] --------- [0..1] b : B
	c : C [0..1] ---------{subsets b} [0..1] d : D
Subset fixed upper and lower	a : A [0..1] --------- [2..5] b : B
	c : C [0..1] ---------{subsets b} [2..3] d : D
Subset composite many	a : A [0..1] ♦-------- [0..*] b : B
	c : C [0..1] ♦--------{subsets b} [0..*] d : D

Set<A> (this is explained in detail in [3]). The OCL collections are assumed to be immutable and thus the same issue is not a problem, until we come to the need to implement collection based properties.

A solution to this problem is to use the more advanced features of the Java generics. We can define the element type of a collection to be something that extends a particular type, e.g. <? extends Type>. The Java collection type Set<? extends B> does subtype Set<? extends A>, thus methods with these types as return types are compatible. It does give us another issue regarding mutators; the type Set<? extends A> is read-only, thus we must explicitly provide mutators 'addB(B b)' and 'addA(A a)' (and similar 'remove' methods) on the class containing the collection based property.

Of course if the two property names are the same, then the signatures of the mutators void setA(Set<A> a) and void setA(Set<B> b) have the same signatures under the type erasure semantics of Java generics, but do not override each other. Unfortunately, defining the mutators as follows does not compile, as one method does not override the other:

```
void setA(Set<? extends A> a)
```

and `void setA(Set<? extends B> b)`

The only solution we have found to work in this case is to loose the type information and fall back on non-generic collections.

7.3.4 Default values

Default (literal) values (such as true, false, 1, 4, "hello") are given in the UML Infrastructure specification and it is assumed that these literal values can be converted to the appropriate type such as PrimitiveTypes::UnlimitedNatural or PrimitiveTypes::Boolean.

To implement this, either we must ignore the specification of the primitive types package and use java.lang types instead, or implement a mechanism to map the strings given as default values for such primitives into objects of the correct type.

7.3.5 Redefinition and upper multiplicity value

Redefining a property with the same name but with upper redefined as 1 rather than > 1 is legal because the multiplicity of the redefining end is contained in the multiplicity of the redefining end. However, in Java the method A getA() is not compatible with the method Set<A> getA().

We can't implement both methods, as the return types are incompatible. The only option is to force the return type of the redefining method to be a collection type, like the return type of the redefined method. This is not very satisfactory as a solution, as it means that a non-collection property is implemented as a collection type. The issue occurs with respect to the classes Extension and ExtensionEnd in the UML Infrastructure package 'Profiles'.

7.3.6 UML infrastructure Constructs::Property

Constructs::Property inherits three versions of the property 'type' (via multiple inheritance paths):

  (a) type : Basic::Type   from Basic::
                                      TypedElement
  (b) type : Constructs::Type   from Constructs::
                                      TypedElement
  (c) type : TypeElements::Type   from Abstractions::
                                      TypesElements::
                                      TypedElement

However, Basic::Type is not compatible with the other two, it would need to extend TypedElements::Type to be so, though this would have knock on effects to other parts of Basic, and require it to redefine and subset some inherited properties. This would mean that to keep to its objective of being able to define itself, Basic would need to define the redefinition and subsetting concepts, which is not an ideal solution. Although the property 'type' is redefined along some of the inheritance paths, it is not redefined along all of them. This is possibly a problem with the specification of the infrastructure model.

7.3.7 UML infrastructure Constructs::Package

There is a problem implementing the properties involved in the association between classes Package and PackageableElement. Property 'ownedMember:PackageableElement' is specified to redefine property 'ownedMember:NamedElement' (from Namespace) and is also specified as uni-directional. Thus, to implement the bi-direction semantics of the redefined property, the opposite end should be assumed to be the opposite end of the redefined property. However, in this case the opposite end of the redefined property is a derivedUnion, subsetted by the non-navigable property that is the opposite of the property 'ownedMember:PackageableElement'. As it is a non-navigable property, no implementation is provided, meaning that the derivedUnion property should take part in the bi-direction semantics imple-

mentation. However, derivedUnion properties are read-Only.

It is unclear as to what the correct semantics should be in this situation, either this is a problem with the specification, or an incorrect interpretation of the semantics regarding this combination of non-navigable, derivedUinon, redefinition and bi-directional characteristics of a property.

7.4 Future work

There is still scope for future work regarding the semantics of qualified properties in conjunction with other property characteristics. In addition we have not addressed *n*-ary association, and the semantics implications of the various characteristics with respect to *n*-ary associations and the implementation of them. Another important issue is with respect to making the generated code "thread safe" and there are also a number of areas in which the efficiency of the generated code could be improved.

Regarding the semantics of UML associations and properties, a number of problems surround the mutation of collection based properties; i.e. destruction of objects contained in the collection and adding or removing objects to/from the collection. Basing the collection semantics on OCL collections is not sufficient as OCL assumes immutable collections, whereas an implementation requires facility to mutate the collections. We feel there is scope for additional work in this area, in particular to investigate what semantics should be implemented.

8 Conclusion

UML Class diagrams are extensively used within analysis and design of software systems and with the recent emergence of Model Driven Development, Domain Specific Languages and other related subjects, Class Diagrams are taking on an additional level of use with respect to the modelling of languages, or meta-modelling. With this in mind it becomes more and more useful to be able to automatically generate implementations of models as part of a fast model-implement-test-model cycle of evolution for the specified models and meta-models.

Existing UML modelling tools provide mixed levels of support for code generation and implementation of Class Diagrams. One area that is particularly lacking in some tools is correct implementation of the semantics of an association. The recent development of the new UML 2.0 set of standards introduces a number of

variations to the concept of an association. The aim of this paper has been to illustrate appropriate code patterns that can be used to correctly implement the semantics of an association, in particular with respect to the new UML concepts of redefinition and subsetting. Additionally, we have provided important code patterns for the implementation of qualified associations, which although implemented adequately in a few tools, are largely ignored with regards to code generation.

Of particular interest has been the *feature interaction* points regarding the multiple different characteristics that can be defined for an association end. We have shown that with careful thought, appropriate implementations can be provided for each meaningful combination although there are still some outstanding issues regarding the complexity of combining qualified properties with other property characteristics. Although, the implementation of the new UML 2.0 redefinition and subsetting concepts does not appear difficult, there are some tricky issues; such as redefining non-navigable ends or subsets involving non-set based collections. We have highlighted these trouble spots and in some cases indicated an appropriate implementation solution.

Regarding the implementation of qualified associations, it proves to be desirable to have a programming concept equivalent to a tuple, which does not exist in Java (or other main stream programming languages); however, an alternative solution is easy to provide although some compile time type information is lost. It has also been necessary to decide upon a reverse navigation policy for qualified associations that differs from that suggested in other literature.

The two initial goals of the work ('readable code' and 'no use of bespoke libraries') have proved to be in conflict with each other regarding the implementation of collection-based properties. We have adequately shown that it is possible to provide implementation patterns that do not require the use of bespoke library components. However, in general bespoke collection types provide a neater code generation option, in part due to differences between the Java and OCL notions of a collection. Also provision of a Java equivalent to the notion of a tuple would clean up some of the generated code.

Overall we have been successful in deriving a mapping from the UML 2.0 concepts of property and association on to the Java 5 programming language. The use of the Java 5 generics feature is especially useful in retaining much of the type information given in the model. The development of code templates for the generation of code in accordance with the patterns described in this paper has enabled us to generate implementations for a number of meta-models that provide support for tools built as part of the ModEasy research project.

References

1. Amelunxen, C., Schürr, A., Bichler, L.: Codegenerierung für Assoziationen in MOF 2.0. In: Proceedings Modellierung 2004, P-45, Marburg, Germany, March 2004, pp. 149–168
2. Bock, C.: UML 2 composition model. J. Object Technol. **3**, 47–73 (2004)
3. Bracha, G.: Generics in the Java programming language. Sun, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, July (2004)
4. Bunse, C., Atkinson, C.: The normal object form: bridging the gap from models to code. In: France, R., Rumpe, B. (eds.) proceedings UML'99 – the Unified Modeling Language. Beyond the Standard, Second International Conference Fort Collins, USA, October 1999.LNCS, vol. 1723, pp. 675–690. Springer, Berlin Heidelberg New York (1999)
5. Büttner, F., Gogolla, M.: On generalization and overriding in UML 2.0. In: Proceedings OCL and Model Driven Engineering Workshop at Seventh International Conference on UML Modeling Languages and Applications <<UML>> 2004, Lisbon, Portugal, October 2004
6. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley Professional, New York, ISBN 0321193687 (2003)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, New York, ISBN 0201633612 (1995)
8. Génova, G.: The meaning of multiplicity of n-ary associations in UML. J. Soft. Syst. Model. **1**, 86–97 (2002)
9. Génova, G.:Interlacement of structural and dynamic aspects in UML associations (Spanish). Ph.D. thesis, Department of Informatica, Carlos III University of Madrid (2003)
10. Génova, G.: UML associations: a structural and contextual view. J. Object Technol. **3**, 83–100 (2004)
11. Génova, G., Castillo, C.R.d., Lloréns, J.: Mapping UML associations into Java Code. J. Object Technol.**2**, 135–162 (2003)
12. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In: Bezivin, J., Muller, P.-A.(eds.) proceedings First International Workshop Unified Modeling Language <<UML>>98. LNCS, 1618, pp. 92–106. Springer, Berlin Heidelberg New York (1998)
13. Guéhéneuc, Y.-G., Albin-Amiot, H.: Recovering binary class relationships: putting icing on the UML cake. In: Proceedings 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 301–314. (2004) ACM Press, Vancouver
14. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML designs to Java. In: Proceedings 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2000, pp. 178 – 187. ACM Press, Minneapolis (2000)
15. IBM: Eclipse universal tool platform. http:/www.eclipse.org (2001)
16. IBM: Eclipse Modeling Framework. http://www.eclipse.org/emf/ (2003)
17. Java Community Process: Java metadata interface (JMI) specification. http://java.sun.com/products/jmi/ (2002)

18. Jean-Pousin, C., Barbey, S.: Implementing associations with Ada. In Proceedings Software Engineering and its Applications, Paris, November 1993, pp. 149–158
19. KMF-team: Kent modelling framework (KMF). http://www.cs.kent.ac.uk/projects/kmf (2002)
20. Kollmann, R., Gogolla, M.: Application of the UML associations and their adornments in design recovery. In: Burd, P. A. a. E. (ed.) Proceedings 8th Working Conference on Reverse Engineering (WCRE'2001), IEEE Computer Society (2001)
21. Larman, C.: Applying UML and patterns: Prentice Hall PTR, New york ISBN 0137488807 (1997)
22. Maier, T., Zündorf, A.: Yet another association implementation. In: proceedings 2nd International Fujaba Days, Darmstadt, Germany, September 2004
23. Matula, M.: NetBeans Metadata Repository. http://mdr.netbeans.org (2003)
24. ModEasy-Team: ModEasy project. http://www.lifl.fr/modeasy/
25. Noble, J.: Basic relationship patterns. In: proceedings 1997 European Conference on Pattern Languages of Programming (EuroPLoP '97), Munich, Germany, Siemens AG, Siemens Technical Report 120/SW1/FB, 1997
26. OMG: The Object Mangement Group. http://www.omg.org
27. OMG: Meta object facility (MOF) 2.0 core specification. Object Mangement Group, ptc/03-10-04, October 2003
28. OMG: UML 2.0 infrastructure specification. Object Management Group, ptc/03-09-15, September 2003
29. OMG: UML 2.0 OCL specification. Object Mangement Group, pct/03-10-14, October 2003
30. OMG: UML 2.0 superstructure specification. Object Management Group, ptc/03-08-02, August 2003
31. OMG: XML metadata interchange (XMI), v2.0. Object Management Group, formal/03-05-02, May 2003
32. Selic, B.: The pragmatics of model-driven development. IEEE Soft. **20**, 19–25, (2003)
33. Stevens, P.: On the interpretation of binary associations in the Unified Modelling Language. J. Soft. Syst. Model. **1**, 68–79, (2002)
34. Sun: Java technology. http://java.sun.com/
35. Suscheck, C., Sanden, B.: A construct for effectively implementing semantic associations. J. Object Technol. **2**, 101–111, (2003)

Author Biography

Dr. David Akehurst is a Research Associate at the University of Kent. He has several years experience working on projects involving techniques and tools for model driven development. Currently he is applying his expertise to the area of embedded systems. Initially he worked on a British Telecom sponsored project involving the use of UML for specifying distributed systems and the transformation of the designs into a discrete event simulation model. David received his PhD in 2000; his thesis was specifically focused around how to specify model transformations and generate implementations from them. Later work made use of UML in relation to distributed system design, drawing on the RM-ODP work from the telecommunications industry; part of this work investigated the transformation from designs into formal models based on Timed Automata and the provision of intelligent design environments supporting such transformations. David has also worked on the development of the Kent Modelling Framework tool kit, which generates implementations from model specifications, and the Kent OCL library, which provides an implementation of the standard Object Constraint Language.

Dr. Gareth Howells is a Lecturer in Electronic Engineering at the University of Kent, U.K. He received his PhD degree in Computer Science in 1991 specialising in the investigation of the practical difficulties present in the efficient exploitation of various aspects of Theoretical Computer Science in the large. He has since been involved in substantial research into the formal verification of digital systems and the practical development and application of artificial neural systems. This work resulted in the development of wholly declarative theorem proving tool for digital system verification and the development of a generic formal mathematical model describing the behaviour of artificial neural systems. His research interests have also involved the practical application of formal mathematical logic to the design both of artificial neural networks and more general pattern classification techniques.

Dr. Klaus McDonald-Maier received Dipl.-Ing. and M.S. degrees in Electrical Engineering from the University of Ulm, Germany and the École Supérieur CPE-Lyon, France in 1995, respectively. In 1999 he received a doctorate in Computer Science from the Friedrich-Schiller-University Jena, Germany. Dr. McDonald-Maier worked as Systems Architect on reusable microcontroller cores and modules at Infineon Technologies AGs Cores and Modules Division in Munich, Germany and as a lecturer in Electronic Engineering at the University of Kent, UK. In 2005 he became a Reader in Computer Science at the University of Essex, Colchester, UK. His current research interests include Embedded Systems and System-on-Chip (SoC) design, development support and technology, parallel and distributed architectures, the application of soft computing techniques for real world problems and robot control.