Generation of Type Safe Association Implementations

Dietrich Travkin, Matthias Meyer
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[travkin|mm]@uni-paderborn.de

ABSTRACT

Model driven development facilitates the specification of software models from which code can be generated automatically. In practice, a software system can often not be modelled completely. Developers still have to implement parts of it manually and thus have to work with the generated code. Therefore, the usability of the code is important.

The Fujaba Tool Suite is a UML case tool which allows to model the structure and behaviour of a system with UML diagrams and to generate Java code for the specifications. However, the code currently generated for associations is not type safe. Furthermore, a lot of code is added to the implementation of the model classes which decreases their usability. In this paper, we present an approach to generate type safe association implementations in Java which also improves the readability and usability of the generated code.

1. INTRODUCTION

Model driven development helps to cope with the continuously increasing complexity of the software systems being developed today. Instead of implementing a whole system by hand in some programming language, models are designed in modelling languages, e.g. UML. Then, code can be generated automatically from these models. Today, typically not the whole system can be modelled and generated, so that developers still have to work with the generated code manually.

When modelling a system with UML, its structure is defined with class diagrams. Class diagrams allow specifying various kinds of associations between classes. However, today's programming languages have no support for associations. Thus, when generating code for class diagrams, associations have to be expressed using the language constructs available in the particular programming language.

The Fujaba Tool Suite [3] is a UML case tool that offers UML class diagrams to model the structure and extended UML activity diagrams, called story diagrams, to model the behaviour of a software system. Fujaba's class diagrams support most binary association types defined by the UML (qualified, sorted or ordered, 1-to-1, 1-to-n, n-to-m, bi- and unidirectional).

Furthermore, Fujaba is able to generate Java code for its models. Classes modelled in class diagrams are translated to Java classes. To implement the associations, the classes participating in an association get additional private attributes to store associated objects and public access methods to link, unlink, or iterate through associated objects. The access methods also ensure consistency for bidirectional associations by mutually calling each other.

Depending on the type of association, up to 18 access methods per association are generated into a model class. Thus, a class participating in several associations becomes badly readable. Furthermore, the generated code is not type safe. Thomas Maier and Albert Zündorf [5, 4] recognised these problems and propose a different solution. They greatly improve the readability and usability of the generated code by extracting the association methods into separate classes representing association ends, so called role classes. However, their approach can not guarantee type safety as well and the support for associations which are qualified on both sides remains unclear.

In this paper, we present an approach that adopts the idea presented in [5]. We also implement associations using general role classes which contain all necessary access methods. However, our hierarchy of role classes is organised differently to explicitly support associations which are qualified on both sides. Furthermore, by generating specialisations of role classes for each concrete association end, we are able to offer a type safe implementation. The role class hierarchy is available as a separate class library which can be used independently of Fujaba. In addition, a Fujaba plug-in has been developed which adapts the code generation of Fujaba for non-qualified associations to use the new approach.

In the following section we present requirements for an association implementation and point out the limitations of the existing approaches in more detail. Section 3 describes our approach for implementing associations and Section 4 describes a Fujaba plug-in which generates code according to it. Section 5 concludes the paper and indicates future work.

2. REQUIREMENTS

There are many ways to implement associations. In all cases the implementations have to manage the references between the objects connected by an association depending on its kind (e.g. qualified 1-to-n) and its constraints (e.g. ordered). In [6] we identified several requirements which should be fulfilled by an association implementation. The four most important of them which we address in this paper are:

1. **Association types**
   Associations may be uni- or bidirectional with multiplicities 1-to-1, 1-to-n or n-to-m. In case of to-n as-

sociations, the associated objects may be ordered or sorted. Furthermore, associations may be qualified on one or both sides. All combinations have to be supported.

2. **Consistency**
In case of bidirectional associations, consistency has to be ensured: if there is a reference from object a to object b then there also has to be a reference from object b to object a. Thus, when object a is linked/unlinked to b, the reverse link from b to a has to be established/removed automatically.

3. **Type safety**
The generated code has to be type safe, i.e. all type errors have to be detected at compile time.

4. **Readability**
The generated code has to be human readable and should add as little code as possible to model classes.

The current Fujaba version fulfills the first two requirements. For each association, private attributes with several public access methods are generated into the code for the model classes. The access methods allow to link or unlink instances of model classes according to the association. In case of bidirectional associations, these access methods automatically ensure consistency by calling each other. When an access method is called on object a to connect it to object b, the method calls the corresponding access method on b to link it to a. The same holds for the access methods to unlink objects. Thus, to establish or remove a link, a call of the appropriate access method on one of the objects is sufficient.

For the implementation of to-n associations, special container classes are used to store an arbitrary amount of associated objects. These containers allow storing objects of the most general type (Object in Java) and the generated code contains type casts. Thus, the code is not type safe and requirement 3 is not fulfilled. A developer is able to (accidently) insert code into a model class that adds objects of the wrong type to a container managing the connected objects of an association. Since this can not be checked by the compiler, type errors occur at runtime.

Depending on the type of association, up to 18 public access methods per association are generated into the implementation of a model class. This heavily decreases the readability and usability of the generated code and the public interface of the model classes becomes very bloated. Requirement 4 is not fulfilled.

The approach presented in [5] greatly improves the readability of the generated code by providing the functionality to manage the associated model elements in separate role classes and thus keeping the code added to the model classes at a minimum. The role classes are implemented using Java Generics. Generics [1, 2] are new with Java 1.5 and enable generic type definitions in Java. However, in spite of using Generics, the resulting association implementations are not type safe. In order to ensure consistency for bidirectional associations, inside the role classes Java's reflection mechanism is used to call methods on the opposite end of the association. The reflection mechanism requires type casts and thus type safety is lost (cf. [6] for details). In addition, it is unclear how associations which are qualified on both sides are supported. Thus, requirement 4 is additionally fulfilled but requirement 3 is not.

3. TYPE SAFE ASSOCIATIONS

In the following, we present an approach which fulfills all the requirements stated above.

3.1 Implementations based on Role Classes

We adopt the idea proposed in [5] to separate the association implementation and the model implementation from each other. Instead of generating all the code to manage references of associated objects into the model classes, the functionality is provided by separate role classes. The role classes contain all the functionality to link or unlink two model elements and to iterate through the connected elements.

For each kind of association end, a special role class exists implementing the required association methods. In case of a 1-to-1 association, each model class must be able to reference one instance of the other model class at runtime. Thus, a to-1 role class is required which is able to manage one reference. In case of a 1-to-n association, one class must be able to manage an arbitrary number of references to instances of the other class, requiring a to-n role class. N-to-m associations can be realised with the help of two instances of a to-n role class. Association constraints and qualified associations require additional role classes.

Figure 1: A qualified bidirectional 1-to-n association

```
class Person
{
   private Tenant_Home_Role home = null;
   public final Tenant_Home_Role home()
   {
      if (this.home == null)
      {
         this.home = new Tenant_Home_Role (this);
      }
      return this.home;
   }
}
```

Figure 2: An implementation of a model class

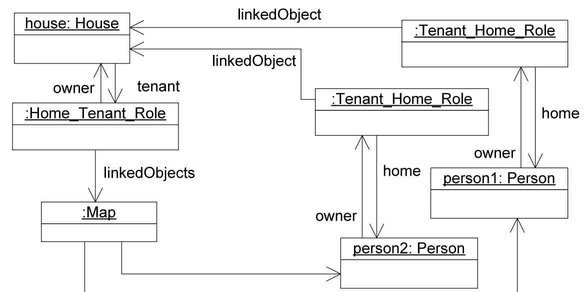

Figure 3: Realisation of the association in Figure 1

An association is realised by two instances of role classes, one for each association end. Instead of the association

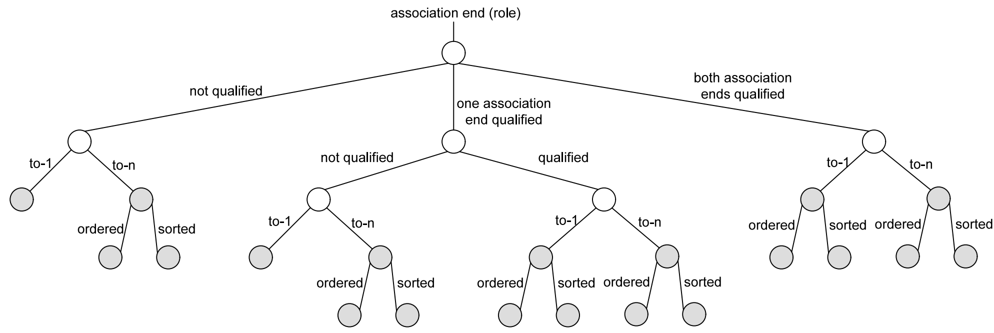

Figure 4: A decision tree describing the role class hierarchy

methods the model classes only get one attribute and one method for each association they are involved in. The attribute saves the role object and the method creates and returns it. Thus, the association methods are implemented only once in a role class and do not have to be generated for each association anymore, which avoids code redundancy. As an example, Figure 2 illustrates the implementation of a model class involved in the association shown in figure 1. Figure 3 shows an object structure in which a House object is connected with two Person objects via the same association using role objects. Note that the figures already use specialised role classes which will be explained in Section 3.3.

3.2 Role Class Hierarchy

Each of the role classes implements a link and an unlink method. In bidirectional associations, these methods have to call each other on either side of an association to maintain consistency. Qualified associations require a key to link two model elements. Since the methods call each other on both sides of an association, they require this key, regardless whether they are called on an object representing a role that is qualified or not (cf. Figure 5). If both sides are qualified even two keys are needed. Therefore the number of parameters in the link and unlink methods are different depending on whether the role is used in an association that is not qualified, qualified on one side or qualified on both sides. Thus, a common abstraction for all role classes would only be possible, if all link and unlink methods had three parameters (one for the object to be linked or unlinked and two for possible keys). In many cases, however, the additional parameters would be useless. Therefore, we propose a hierarchy of role classes which is divided into three rather independent sub hierarchies for associations which are not qualified, qualified on one side, or qualified on both sides, respectively. Inside these hierarchies, the number of required parameters is equal and a common abstraction exists.

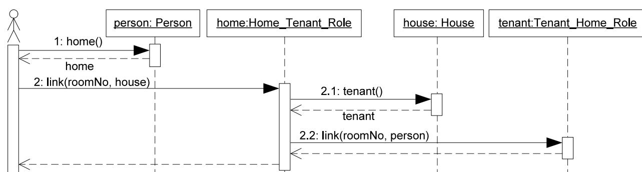

Figure 5: A call of the link method in the qualified association shown in Figure 1

Figure 4 shows a decision tree which describes the whole role class hierarchy and helps to determine the role classes needed to realise a particular association. Each node in the tree corresponds to a role class. An edge is labelled according to the purpose of the child role class and indicates that the child role class refines its parent. At the first level, the tree is divided into the three sub hierarchies. The left hierarchy shows the role classes for non-qualified associations. It is subdivided into a class for the to-1 side and one for the to-n side which again has two children, one for an ordered and one for a sorted to-n side. The hierarchy in the middle contains the role classes for associations qualified on one side. It is subdivided in roles for the non-qualified side and those for the qualified side which in turn are organised according to the type (to-1 or to-n side) as well as to possible constraints (ordered and sorted). The hierarchy on the right shows the roles for associations qualified on both sides and is organised according to the same criteria as the other hierarchies.

Although the number of different role classes is rather high, it is easy to choose the right role classes for a particular association. The decision tree in Figure 4 describes how to do that (dark nodes indicate possible decisions). The association between House and Person (cf. Figure 1), for example, is qualified on one side. Thus, role classes from the hierarchy in the middle have to be used. A House object must be able to manage an arbitrary number of Person objects for each key. Therefore it requires a qualified to-n role. A Person object must be able to store one House object. Since the opposite side is qualified and this side does not use a key, a non-qualified to-1 role class from the hierarchy in the middle has to be chosen.

The role classes are generic. They use type parameters for the type of the elements to be referenced by the roles, the type of the role owner and – in qualified associations – the key types. The generic type definitions are necessary for a type safe implementation but not sufficient. This is described more precisely in the following section.

3.3 Type Safety

To keep bidirectional associations consistent, methods for linking and unlinking two objects are called on both sides of an association (link on one side of an association calls link on the other side of the association, cf. Figure 5). The link and unlink methods are implemented within the general role classes, where the types of the elements referenced by the role classes are only represented by type parameters. The concrete types used in a concrete association are not known

and thus the access method for the role object of a model element (e.g. method home in Figure 2) is not known. This makes a call of the link or unlink method on the other side of an association impossible.

However, the access methods in the model classes can be revealed by subtyping the generic role classes and binding its type parameters to the concrete types of the model elements involved in a concrete association. In each general role class in the hierarchy, an abstract method getOppositeRole is declared which takes a model element to be linked or unlinked as argument. The method is meant to return the role object from the given model element which represents the opposite end of the same association. It is used inside the general role class implementations to get the (opposite) role object from a model element to be linked or unlinked. Thus, all role classes in the hierarchy are abstract. They contain the complete implementation except of the getOppositeRole method.

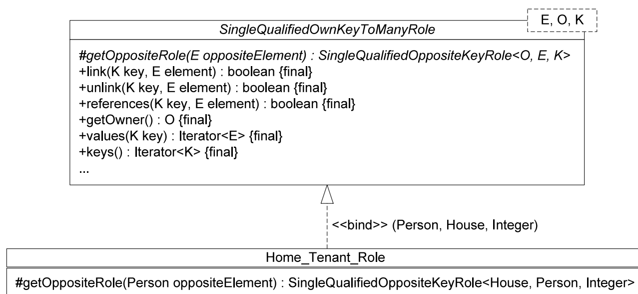

Figure 6: A specialisation of an abstract role class

```
public class Home_Tenant_Role
  extends SingleQualifiedOwnKeyToManyRole<Person,House,Integer>
{
  public Home_Tenant_Role(House owner)
  {
    super("home", owner);
  }
  protected SingleQualifiedOppositeKeyRole<House,Person,Integer>
    getOppositeRole(Person oppositeElement)
  {
    return oppositeElement.home();
  }
}
```

Figure 7: A specialisation of an abstract role class

For each association end, a concrete subclass of a generic role class has to be implemented in which the type parameters are bound to the concrete types of the model elements. Inside these classes, the getOppositeRole method has to be implemented (cf. Figures 6, 7). The concrete subclasses only contain a constructor as well as the getOppositeRole method and thus are very small. Without them, however, a type safe access to the role representing the opposite end of an association would not be possible.

4. CODE GENERATION

The role class hierarchy described in Section 3.2 has been implemented in a class library which can be used to implement associations independently of Fujaba.

In addition, a Fujaba plug-in has been implemented which uses the role class library to generate code for non-qualified associations according to the approach presented in this paper. The plug-in generates the specialised role classes required for each association. To further increase the usability, each concrete role class is placed in a subpackage of the corresponding model class' package named roles. The code generated for model classes uses the specialised role classes to realise the associations. Furthermore, the code generation for story diagrams has been adapted. The code generated for story diagrams calls access methods of associations to create or destroy links between objects.

5. CONCLUSIONS AND FUTURE WORK

This paper describes an adaption and extension of the ideas presented in [5] leading to type safe association implementations with improved readability and usability. The functionality for managing associations is provided by separate role classes and no longer generated into the model classes. By specialising the role classes for each concrete association end the code is made type safe. A class library with all necessary role implementations is available and can be used to implement associations independently of Fujaba. A Fujaba plug-in adapting the code generation of Fujaba for non-qualified associations is available as well.

Fujaba itself is partially implemented using its own code generation mechanism. Re-generating the Fujaba code using the new association implementations would increase its usability. Furthermore, type errors in association implementations would be revealed already at compile time.

The next step could be to support the modelling of generic types with UML templates in Fujaba. This would enable the generation of completely[1] type safe code.

6. REFERENCES

[1] G. Bracha. *Generics in the Java Programming Language*. Online at: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, July 2004, Tutorial.

[2] G. Bracha, N. Cohen, C. Kemper, S. Marx, M. Odersky, S.-E. Panitz, D. Stoutamire, K. Thorup, and P. Wadler. *Adding Generics to the Java Programming Language: Participant Draft Specification*. Online at: http://www.jcp.org/aboutJava/communityprocess/review/jsr014/, April 2001.

[3] Fujaba Development Group. *Fujaba ToolSuite*. Online at: http://www.fujaba.de, 2004.

[4] T. Maier. *Associations*. Online at: http://sourceforge.net/projects/associations/, November 2004. Version 0.4.

[5] T. Maier and A. Zündorf. *Yet Another Association Implementation*. In *Proceedings of the 2nd International Fujaba Days, Darmstadt, Germany*, pages 67–72, Septemeber 2004.

[6] D. Travkin. *Generierung typsicherer Implementierungen für Assoziationen in UML-Modellen (in german)*. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, February 2005.

[1]This paper only focuses on the code generated for associations. The code generated by the described Fujaba plug-in is type safe for associations only.