第 33 卷 第 3 期 计 算 机 工 程 2007 年 2 月
Vol.33 No.3 Computer Engineering February 2007
·软件技术与数据库· 文章编号：1000—3428(2007)03—0067—03 文献标识码：A 中图分类号：TP311

基于 MDA 的关联关系转换方法

夏 雷，欧阳松
(中南大学信息科学与工程学院，长沙 410083)

摘 要：根据 MDA 中模型自动转换到代码的特点，提出了一种 UML 类图中关联关系到代码的转换方法。讨论了 UML 中关联关系及其两种实现模式，对每种模式分别定义了一套从 UML 模型（平台独立模型）到 Java 模型（平台相关模型）的变换规则，给出了两种实现模式按规则转换的实例。

关键词：模型驱动构架；UML；关联；实现模式；变换

Method of Association Transformation Based on MDA

XIA Lei, OUYANG Song
(College of Information Science and Engineering, Central South University, Changsha 410083)

【Abstract】 According to the specialty of automatic transformation from model to code in MDA, this paper proposes a method to transform association in UML class diagrams to code based on MDA. It discusses the association in UML and its two implementation patterns. For every pattern, it defines a suit of rules, which are used in transformation from UML model(PIM) to Java model(PSM), respectively. It presents two transformation examples of implementation pattern by using rules.

【Key words】 MDA; UML; Association; Implementation pattern; Transformation

UML是一种可视化的通用的面向对象建模语言，它可以用来为软件系统建模，描述系统架构和业务过程。在软件开发过程中，设计人员利用UML为系统创造模型，开发人员进一步将该模型手工转换为程序代码。UML中类、属性、操作等概念和面向对象语言中的相应概念直接对应，便于转换；但关联关系、聚合、依赖等概念在面向对象语言中却得不到支持，如何把这些关系转换到程序代码成为软件开发过程的瓶颈问题[1]。为了提高生产效率，国际对象管理组织(OMG)提出了新的软件开发方法——模型驱动构架（MDA），其核心思想为模型到程序代码的自动转换[2]。MDA框架中主要元素有模型、语言、变换、变换定义以及变换工具，其中模型是系统的描述，分为平台独立模型(Platform Independent Model，PIM)和平台相关模型(Platform Specific Model，PSM)[3]。PIM描述了系统的功能和结构，独立于任何实现技术；而PSM在特定的目标平台上对系统进行描述，它与系统实现技术相关，包含了最终实现平台的全部知识。

MDA 开发过程为：首先使用平台无关的建模语言（例如 UML）搭建 PIM，然后根据基于特定平台的变换规则，将 PIM 变换为 PSM，最终生成应用程序代码和测试框架。MDA 方法的特点是模型变换由变换工具自动完成。现有许多工具可以把 PSM 变换成代码，而 MDA 还能实现 PIM 自动变换到 PSM，这样软件开发人员只需用 UML 定义系统的 PIM，即可通过变换工具，产生相应平台上的代码。

由于 MDA 能实现模型到代码的自动转换，可以利用它解决 UML 类图中关联关系转换到代码的难题，该方法的步骤为：(1)定义一套基于特定平台的关联变换规则，并且构造按照规则执行变换的变换工具；(2)将关联关系的 UML 模型输入工具中，产生相应平台的关联模型；(3)通过 PSM 到代码的变换工具生成实现关联的代码。

1 UML 中关联关系及其实现模式

1.1 关联关系简介

关联（Association）用于描述类与类之间的连接。其连接方式多种多样，连接的含义各不相同，但外部表示形式相似，故统称为关联[4]。关联一般用一个线段来表示，线段上附带名称，以表示这个关联的含义。关联两端与类之间的接口表示该类在关联中的作用，称为角色(Role)。角色可以具有多重性，多重角色由类的多个对象扮演，用*表示。按关联导航方向的不同，可将关联分为：单向导航关联和双向导航关联；按两个关联端上重数的不同，可以将关联分为：一对一关联，一对多关联和多对多关联，其中多对多关联较复杂，在实现时，一般将其转化为多个一对多关联。图 1 表示了一个一对多的双向导航关联模型。

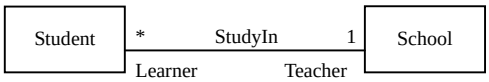

图 1 学生和学校的关联模型

1.2 关联实现模式

所谓关联实现模式，是指处理类与类之间关联关系和关联产生的属性的方法。目前，有两种实现模式：隐式实现模式和显式实现模式[5]。

1.2.1 隐式实现模式

该模式将关联关系及其属性分配到相关联的类中。类与类之间的关系为关联关系，由关联产生的属性作为相应类中的属性，而且每个类还包含指引(reference)属性，通过该属性

作者简介：夏 雷(1982－)，男，硕士生，主研方向：模型驱动构架，UML 建模；欧阳松，教授、博士

收稿日期：2006-02-21 **E-mail：**xialei525@etang.com

可以访问相关联的类。这种实现模式包含有关联关系，但没有以显式的形式给出。

图 2 给出了一个隐式实现模式的实例。在该关联中，由于学生在学校里读书这种关系，因此把“班级”和“年级”这两个关联关系产生的属性作为“学生”类的属性。“学校”类和“学生”类都拥有 reference 属性，“学校”类的对象通过 Student 可以导航到相关联的“学生”类的对象上，而“学生”类的对象通过 School 可以导航到“学校”类的对象上。

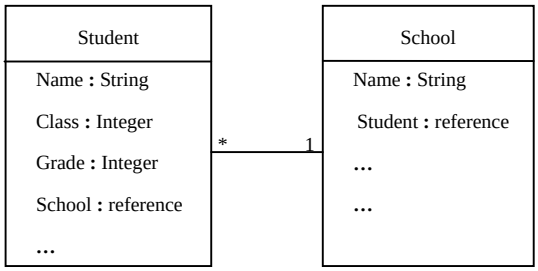

图 2 隐式实现模式

隐式实现模式具有主题突出、结构简练的特点。但它的不足之处也很多：用于实现关联的代码隐含在类中，理解关联关系难度较大；信息冗余大，依赖于关联的信息需要保留多份，给系统带来较大的开销；缺乏灵活性，解除关联关系时，需修改代码，比如在上例中，若有学生离校，就要删除该学生的“班级”、“年级”和 reference 属性。

1.2.2 显式实现模式

由于隐式实现模式存在诸多问题，因此为了增强关联的简单性和灵活性，有人就提出了显式实现模式。此模式引入一个关联类的概念。关联类是一个与关联关系相连的类，表示该关联的信息。它有多个对象，每一个对象称为一个链接(Link)。那么，关联类反映相关联的类之间的关系，而链接则反映的是相关联类的对象之间的关系。由关联产生的属性被移到关联类中作为关联属性，而相关联的类只包含自己的固有属性，且必须通过关联类中的 reference 属性才能访问对方。图 3 给出上例的显式实现模式。

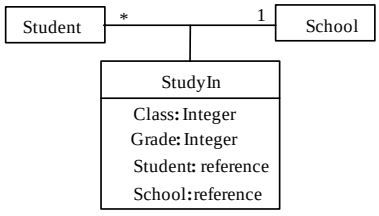

图 3 显式实现模式

在此模式中，StudyIn 类为关联类，其中属性包含由关联产生的属性：“班级”、“年级”和 reference 属性：Student、School。该类的每个对象 Link 把一个 Student 类的对象 St 和一个 School 类的对象 Sc 链接起来。St 和 Sc 是相互独立的，当对象 Link 被撤销，它们之间的关联关系也被解除，而 St 和 Sc 依然可以和其他类的对象相互作用，无需作任何改动。这种实现模式使类彼此之间不再具有依赖性，并且增强了类的可维护性和可重用性。

2 基于 Java 平台的关联变换规则

为了实现两种关联模式，本节给出 MDA 中 PIM 到 PSM 的变换规则。该规则基于 Java 平台，将 UML 描述的关联模型转换到 Java 模型。它用文献[6]中定义的变换规则语言表述，且其中一部分利用了 OCL，可以被变换工具解释和执行。由于两种关联模式实现方法不同，它们的变换规则也有较大差别，以下分别列出两种规则，并对其注释说明。

2.1 隐式关联变换规则

变换的对象为一对多的双向导航隐式关联模型。规则内容是将 UML 描述的相互关联的类变换为 Java 模型中的类。由于关联关系是隐含的，它在 Java 模型中也是隐式实现的，Java 类的属性包含关联产生的属性、reference 属性和类的固有属性，类与类之间用 reference 属性的 get 和 set 方法互相访问。

规则分为两部分：

(1)将重数为 1 关联端的类变换为 Java 类。

```
Transformation
AssociationToJavaClassForOneStep( UML,Java)
{
 source
      assocEndSingle: UML:: AssociationEnd;
      assocEndMultiple: UML:: AssociationEnd;
      assocAttributes: UML:: AssociationAttribute;
          /*UML 模型中两个类的关联端及关联产生的属性作为源对象*/
  target
      keyClass: Java:: JavaClass;
      classAttributes: Java:: JavaClassAttribute;
      refAttribute: Java:: referenceAttribute;
      classGetMethod: Java:: JavaClassMethod;
      classSetMethod: Java:: JavaClassMethod;
      /*Java 模型中的类及其属性、方法作为目标对象*/
  source condition
      assocEndSingle.upper = 1 and
      assocEndMultiple.upper >1;    /*变换的初始条件为一个关联端重数为 1 另一个大于 1*/
  target condition
      classAttributes.class = keyClass   and
      refAttribute.class = keyClass   and
      classGetMethod.class = keyClass   and
      classSetMethod.class = keyClass;   /*结果条件为产生的属性、方法是产生的 Java 类的属性方法*/
  unidirectional;   /*变换是单向的*/
  mapping              /*主体部分*/
  assocEndSingle.name <~> keyClass.name;
    /*重数为 1 的关联端的类名作为 Java 类名*/
  assocEndMultiple.name <~> refAttribute.name;
  refAttribute.type = Set;   /*另一重数大于 1 的关联端的类名作为 Java 类的 reference 属性名，类型为 Set*/
  assocAttributes.name <~> classAttributes.name;
  assocAttributes.type<~>classAttributes.type;
   /*关联产生的属性作为 Java 类的属性*/
    'get' + assocEndMultiple.name <~>
  classGetMethod.name;
    classGetMethod.returnValue.type = Set;
    classGetMethod.parameters->isEmpty();   /*产生 Java 类的 Get 方法，返回值类型为 Set，参数为空，该方法用于获取关联另一端的类*/
     'set' +assocEndMultiple.name <~>
  classSetMethod.name;
    classSetMethod.returnValue.type = void;
    classSetMethod.parameters-> exists (P|P.name = assocEnd Multiple.name and P.type = Set);   /*产生 Java 类的 Set 方法，返回值
```

类型为 void，参数名为重数大于 1 的关联端的类名，参数类型为 Set，该方法用于设置关联另一端的类*/ }

(2)将重数大于 1 的关联端的类变换为 Java 类。该部分与第(1)部分内容大体相同，都是变换 UML 描述的类到 Java 类，类中的属性、方法具有一样的模式，只是类型和名称不一致。规则形式上与第(1)部分略有差异，比如：第(2)部分中 key Class.name 由 assocEndSingle.name 产生，而在此部分中该语句变化为 assocEndMultiple.name <~> keyClass.name；Java 类的 reference 属性名 refAttribute.name 变为 assocEndSingle.name，类型也不再是 Set；在第(2)部分中 Get 和 Set 方法的名称、参数名及返回值类型也有所不同，它们与重数为 1 关联端的类相关。由于该部分规则要占大量篇幅，且与第(1)部分大致相同，在此就不赘述了。

2.2 显式关联变换规则

变换的对象为一对多的双向导航显式关联模型。由于所有的关联信息放在关联类中，因此该规则只包含关联类的变换。具体内容为把 UML 表示的关联类变换为 Java 类，关联类中由关联产生的属性和 reference 属性作为 Java 类的属性，针对每个属性产生 Get 和 Set 方法作为类的方法。相互关联的两个类通过关联产生属性的 Get 和 Set 方法获取和设置属性值，通过 reference 属性的 Get 和 Set 方法相互访问。

以下为关联类的变换规则：

```
Transformation AssociationClassToJavaClass(UML,Java)
{ source
    assocClass: UML:: AssociationClass;  /*源对象为 UML 模型中的关联类*/
  target
    javaClass: Java:: JavaClass;  /*目标对象为 Java 模型中的 Java 类*/
unidirectional;  /*变换是单向的*/
mapping          /*主体部分*/
    assocClass.name <~> javaClass.name;  /*关联类名作为产生的 Java 类名*/
    assocClass.attributes.name <~>
javaClass.attributes.name;
        assocClass.attributes.type  <~>
javaClass.attributes.type;  /*对于关联类的每一个属性，生成相应的 Java 类属性，名称、类型保持一致*/
    'get'+assocClass.attributes.name <~>
javaClass.getMethods.name;
    javaClass.getMethods.returnValue.type= assocClass.attributes.type;
javaClass.getMethods.parameters->isEmpty();  /*对每个 Java 类属性产生 Get 方法，返回值类型为属性的类型，参数为空，该方法用于获得属性值*/
    'set'+ assocClass.attributes.name <~>
javaClass.setMethods.name;
javaClass.setMethods.returnValue.type = void;
    javaClass.setMethods.parameters->exists(P|
P.name = assocClass.attributes.name and P.type = assocClass.attributes.type);  /*对每个 Java 类属性产生 Set 方法，返回值类型为 void，参数名为属性名，参数类型为属性的类型，该方法用于设置属性值*/ }
```

2.3 对比两种规则

显式实现模式比隐式实现模式简单直观，同样，从以上两小节可以看出，显式关联变换规则简洁易懂，而隐式变换规则过于冗长。显示变换规则只需包含关联类的变换，即可实现关联关系；而隐式变换规则需要分别对两个关联的类变换，才能将实现关联关系的部分嵌入相关联的类中。因此，笔者建议，若用 MDA 方法开发软件，在用 UML 为系统建模时，尽量选用关联的显式实现模式。

上述两种关联变换规则是从 PIM 到 PSM 的规则，它们把 UML 模型变换到 Java 模型。这种 Java 模型仍为一种框架，与最终可编译执行的代码还有差距，需要定义从 PSM 到代码模型的变换规则，并通过变换工具执行变换，才能生成最终的代码。

3 变换的实例

为证明上一节提出的变换规则是正确的，本节针对两种关联实现模式，分别给出按规则变换后生成的 Java 模型实例。通过实例分析，可以得出结论。

3.1 隐式关联变换实例

图 2 给出的隐式关联实例按变换规则执行变换后，可得到图 4 中 Java 关联模型。

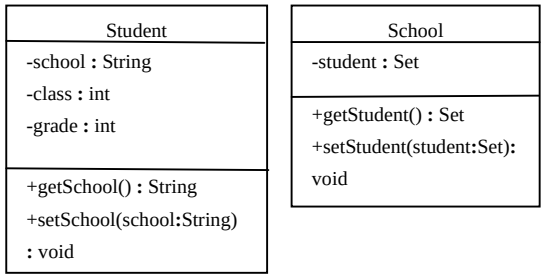

图 4 隐式关联模式 Java 模型

3.2 显式关联变换实例

图 5 为图 3 中的显式关联实例按规则变换得到的 Java 关联模型。

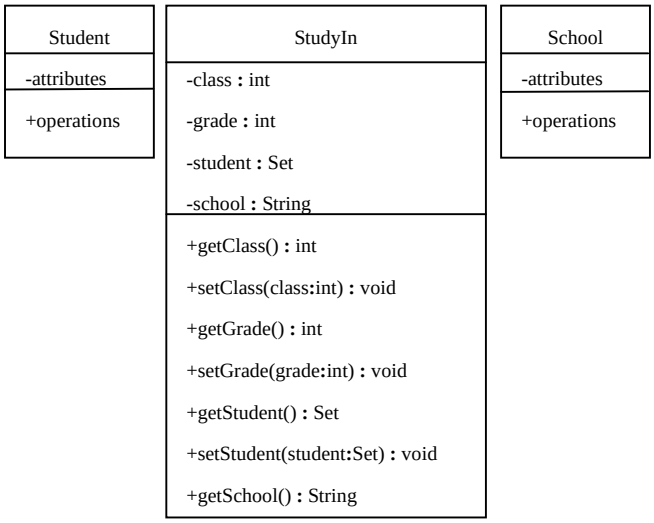

图 5 显示关联模式 Java 模型

以上两个模型与按 UML 源模型生成的 Java 期望模型一致，因此，关联变换规则是可以被用来实现 UML 模型到 Java 模型转换的。

4 总结和今后的工作

本文对 UML 类视图中关联关系及其两种实现模式进行了探讨，提出了利用 MDA 方法实现关联从 UML 模型到程序代码的转换。该转换分为两个阶段：第 1 阶段为 PIM 到 PSM，

（下转第 78 页）