ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2203−2217
DOI: 10.3724/SP.J.1001.2008.02203
© 2008 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于 MDA 的设计模式建模与模型转换*

张 天+, 张 岩, 于笑丰, 王林章, 李宣东

(南京大学 计算机科学与技术系,江苏 南京 210093)
(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

MDA Based Design Patterns Modeling and Model Transformation

ZHANG Tian+, ZHANG Yan, YU Xiao-Feng, WANG Lin-Zhang, LI Xuan-Dong

(Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)
+ Corresponding author: E-mail: ztluck@seg.nju.edu.cn

Zhang T, Zhang Y, Yu XF, Wang LZ, Li XD. MDA based design patterns modeling and model transformation. *Journal of Software*, 2008,19(9):2203−2217. http://www.jos.org.cn/1000-9825/19/2203.htm

Abstract: A motivation of MDA (model driven architecture) is to use models as programs so as to increase the abstract level of software development. Design patterns provide reusable constructs that can be introduced into MDA to increase the modeling granularity as well as the reusability of model transformation rules. This can be achieved by applying design patterns as the integrated development units in MDA, which raises two problems of modeling and model transforming. First, patterns should be used as integrated modeling units. Second, pattern models should be transformed by applying pattern transformation rules built on the pattern units. To solve these two problems, a solution is to define pattern unit metamodel for each pattern and to provide the EJB model transformation rules basing on each pattern unit metamodel respectively. In this way, patterns can be used as integrated modeling units by instantiating the pattern metamodel units, and then the pattern models can be transformed into the EJB models by applying the transformation rules. This paper also demonstrates how to extend MOF (meta object facility) meta-metamodels to define the pattern specifications and how to define the transformation rules to map the pattern models as well as related business models onto the EJB platform.
Key words: MDA (model driven architecture); design pattern; modeling; model transformation

摘 要: MDA(model driven architecture)的一个重要意图是将模型作为软件开发的基本单元,以进一步提高软件开发的抽象层次.为此,MDA 划分了 3 种抽象级的模型,并通过建立高抽象级的模型和向低抽象级模型及代码的转换来构造可运行的应用程序.在 MDA 的框架下,将设计模式作为一种独立的建模和转换单元能够在较高的抽象层次上充分支持复用并提高建模粒度,从而进一步发挥设计模式的优点,提高软件开发效率、降低生产成本.然而,要在 MDA 的框架下将设计模式作为完整的开发单元来使用,必须解决以模式为单元的建模及转换两个具体问题.针对

* Supported by the National Natural Science Foundation of China under Grant No.60425204 (国家自然科学基金); the National Basic Research Program of China under Grant No.2002CB312001 (国家重大基础研究发展计划(973)); the National High-Tech Research and Development Plan of China under Grant No.2007AA010302 (国家高技术研究发展计划(863)); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2007714 (江苏省自然科学基金)

Received 2006-11-16; Accepted 2007-12-24

© 中国科学院软件研究所 http://www.jos.org.cn

单元化模式建模的问题,通过扩展 MOF(meta object facility)的方式定义了模式单元元模型,并提供了基于此元模型的单元化建模支撑机制,以分离业务模型与模式模型的方式解决了该问题.针对单元化模式模型转换问题,在模式单元元模型的基础上定义了向 EJB 平台的转换规则.该转换规则使用 QVT 标准描述,支持单元化的模式模型转换,并具有良好的复用性.

关键词: MDA(model driven architecture);设计模式;建模;模型转换

中图法分类号: TP311 **文献标识码**: A

模型驱动体系结构(model driven architecture,简称 MDA)[1]是国际标准化组织 OMG 于 2001 年提出的一种基于模型的软件开发框架性标准.与传统的软件开发方法相比,MDA 致力于将软件开发从以代码为中心提高到以模型为中心,使模型不仅被作为设计文档和规格说明来使用,更成为一种能够自动转换为最终可运行系统的重要软件制品.

设计模式(design pattern)[2]为面向对象的软件开发提供了可重用的设计方案,有利于提高软件的复用性和系统的可维护性,在工业软件开发过程中获得了广泛的应用.将设计模式引入到 MDA 中作为独立的开发单元使用可以结合两者的优点,进一步推动 MDA 的发展和应用.具体地讲,在 MDA 框架下的设计过程中,以设计模式为独立的建模单元可以复用已有的设计经验,针对“不断重复出现的问题[2]”进行建模,从而提高建模粒度;在模型转换过程中,以模式为独立的转换单元提供向具体平台的转换规则可以提高规则的重用性,降低转换规则的开发工作量.因此,以完整开发单元的形式将设计模式引入到 MDA 中,可以更大程度地挖掘这两种技术的应用潜力,更好地结合两者的优点.

然而,要在 MDA 的框架下将设计模式作为完整的开发单元来使用,必须解决以模式为单元的建模和转换问题.其中,在建模方面需要提供一种方式,使得在 MDA 下的建模过程中既可以将模式作为整体操纵(如整个模式的实例化)的对象,还可以访问模式模型单元中的不同部分.另外,在模型转换方面,需要提供针对模式单元的转换规则.这些规则必须具有重用性,使建模过程所得到的模型可以通过复用模式单元转换规则进行转换.

本文针对上述问题,定义了单元化模式规约元模型,并基于此元模型提出了具体的单元化模式建模途径以及单元化模式模型转换途径.我们的策略是:扩充 MOF(meta object facility)[3]的元-元模型,用以构造设计模式的元模型规约.每一个设计模式对应一个元模型规约,这些模式元模型就作为单元化使用模式进行建模和转换的基础.通过实例化设计模式的元模型获得以模式为单位的建模单元.在使用 UML 2.0[4]进行 PIM 建模的过程中,引入 RolePlay 机制对模式单元中的不同角色进行绑定,从而解决将设计模式作为建模单元的问题.针对设计模式的元模型定义向 PSM 映射的转换规则,并使用 QVT(query view transformation)[5]标准描述该转换规则.每一个模式的元模型对应一组转换规则.以模式为单元复用转换规则将 PIM 中对应的模式模型转换为 PSM,从而解决将设计模式作为转换单元的问题.这样,本文就以一种统一的方式解决了以模式为单元的建模和转换问题,从而将设计模式作为完整的开发单元引入到了 MDA 的框架中.此外,模式单元还为模式组合提供了良好的支持.在本文中,我们主要选择了 J2EE 下的 EJB 平台作为向 PSM 转换的目标平台.

本文首先简要介绍在 MDA 框架下引入设计模式的途径;然后详细介绍扩充的 MOF 元-元模型以及相应的建模支撑机制——RolePlay 机制,其中包括了扩充部分元-元模型的语义以及 RolePlay 关系的元模型.同时,还示例性地给出了 Observer 模式的元模型规约;之后,我们选定 EJB 平台以 Observer 模式为例定义了可复用的模式转换规则,其中包括了转换规则的设计思路以及使用 QVT 描述的映射关系和转换过程;最后,本文使用股票系统中一个场景的开发实例,说明了在 MDA 框架下以模式为开发单元的具体开发过程.

1 在MDA框架下引入设计模式的途径

将设计模式作为完整的开发单元引入到 MDA 的框架下需要解决两个核心问题,即单元化模式建模和单元化模式模型转换.针对前一个问题,本文通过扩展 MOF 定义了模式单元元模型,并引入 RolePlay 机制用以支撑基于此元模型的单元化模式建模.对于第 2 个问题,本文同样是基于模式单元元模型定义了向 EJB 平台的转换

© 中国科学院软件研究所 http://www.jos.org.cn

规则.由于建模和转换都是建立在共同的元模型基础之上的,因此这两部分的工作形成了一个有机的整体.

MOF 将建模语言分为 4 个层次,分别为 M3(元-元模型层)、M2(元模型层)、M1(用户模型层)和 M0(模型实例层).其中每一层模型都可以视为是上一层模型的实例,而 M3 层模型则是这个体系的根.本文在 MOF 的 M3 层增加了 EPattern 和 ERole 两个元-元模型,分别用来定义模式单元的边界和模式结构中的参与者.通过这两个元-元模型,就可以在 UML 这样的建模语言中定义模式单元的元模型.模式单元的元模型位于 MOF 四层体系中的 M2 层位置,它们是本文实现单元化建模和转换所依赖的模式规约(pattern specification).

以模式为单元的建模过程就是将 M2 层的模式元模型进行实例化的过程.所谓设计模式的实例化指的是运用设计模式的思想进行具体建模的过程.本文引入文献[6]的思想定义了 RolePlay 机制来分离模式逻辑与业务逻辑,使得 PIM 建模过程中的模式模型易于理解并支持模式组合.当实例化过程中存在多个模式被组合起来建模的时候,模型会变得复杂和难于理解.我们使单元化的模式规约可以被用来独立地构造模式模型,然后再绑定到通过 UML 这样的建模语言构造的业务模型上.

在解决了关于 PIM 建模方面的问题之后,我们又基于 QVT(query,view,transformation)标准定义了针对模式单元规约的 EJB(enterprise Java bean)转换规则.这一部分工作仍然是以前面定义的模式单元的元模型为基础的,因此可以在 PIM 建模的基础上直接复用转换规则得到相应的 PSM.我们首先根据 EJB 标准[7]给出了 M2 层的 EJB 元模型,然后针对模式单元的元模型和 EJB 元模型定义 PIM-PSM 转换规则.应用该规则就可以以模式为单元进行转换,将 PIM 映射到 EJB 平台上,从而得到 EJB 上的 PSM,如图 1 所示.

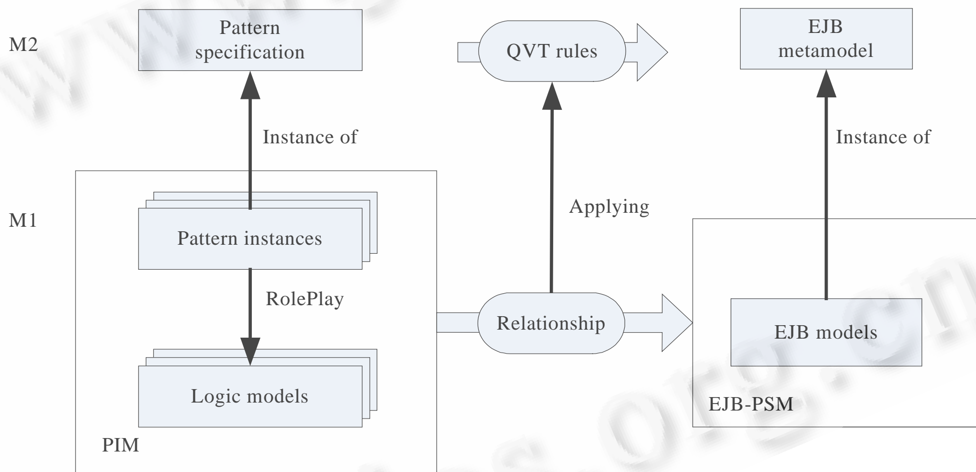


Fig.1 Definition and application of transformation rule

图 1 转换规则的定义和应用

2 设计模式的元模型及相关建模支撑机制

本节围绕着单元化模式建模的问题首先介绍了 Complete MOF 元-元模型,然后给出了扩充的 EPattern 和 ERole 两个元-元模型的语义,并基于扩充后的 MOF 定义了 Observer 模式单元元模型,最后介绍了用于支撑单元化模式建模的 RolePlay 机制.

2.1 Complete MOF元-元模型

MOF 的核心规范主要是通过复用 UML2.0 Infrastructure[8]的 Core 包并引入一组元语言构造机制,如 Reflection,Identifiers 和 Extension,而形成的.该规范由 Essential MOF(EMOF)和 Complete MOF(CMOF)两部分组成,其中,EMOF 只包含了最基本的概念,而 CMOF 则提供了完整的 MOF 元-元模型的概念和扩展机制.图 2 给出简化的 Complete-MOF(CMOF)元-元模型.

CMOF 利用 Package 来划分和复用元模型,以包为单位组织概念.Package 可以包含从 Type 继承下来的元模

型.Classifier 继承自 Type,是 CMOF 中用来描述有类型元素的重要元模型.Classifier 中可以包含继承自 Feature 元模型的元素,在图中主要是 StructuralFeature 和 BehavioralFeature.由斜线“/”标注的关联端是通过导出而得到的,如通过继承关系产生的关联(association)或形成的属性(property)等.Class 继承自 Classifier,用来描述具有同样类型的一组对象.这些对象拥有相同的结构特点,即 Property 和 Operation.Class 之间可以通过 Association 进行关联,关联端/endType 都必须是以 Type 为根的子类型.但是当 Class 之间通过聚合,以属性方式形成关联时,被包含方的关联端以 memberEnd 表示.Operation 描述的是行为信息,可以根据已有类型元素作为参数.

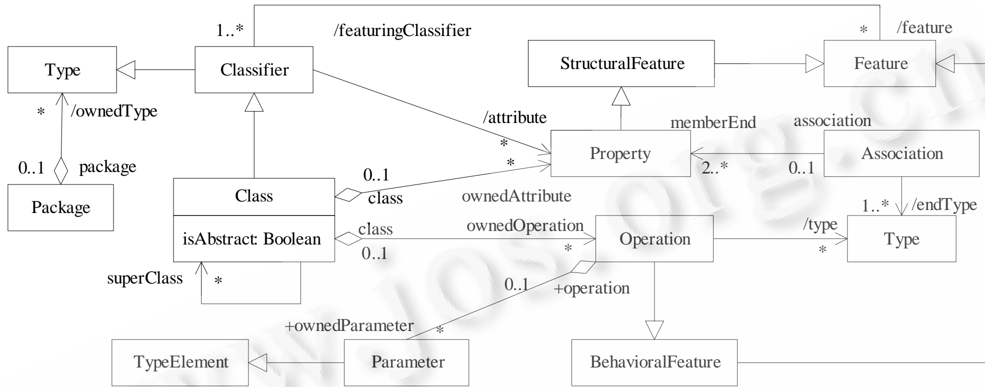


Fig.2 Simplified CMOF meta-metamodel

图 2 简化的 CMOF 元-元模型

UML 2.0Infrastructure 规范的 Core::Basic 章节指出用于构造 MOF 2.0 以及 UML 2.0 Superstructure 的 Core::Basic 包中的元素同时也是自身的实例.这就说明 Basic 包既是 M3 层模型,也是 M2 层模型.其优势在于是消除了元模型层次伸展的复杂性,但同时也给 UML 2.0 以及 MOF 2.0 规范的理解带来了一定的困难.UML 2.0 Superstructure 的 Kernel 包就主要是通过在 M2 层复用 Core::Constructs,从而间接复用 Core::Basic 得到的.UML 2.0 Superstructure 中的其他包则是通过扩展 Kernel 包产生的.图 3 给出了简化的 UML 2.0 元模型.

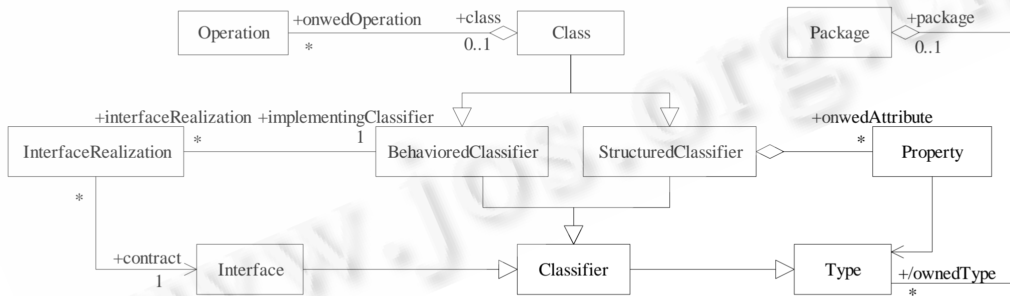


Fig.3 A simplified UML 2.0 metamodel

图 3 简化的 UML 2.0 结构元模型

2.2 模式元模型的语义

本文在MOF的基础上扩充了EPattern(Extended Pattern)和ERole(Extended Role)两个元-元模型(文中也称为模式元模型),专门用于定义模式单元规约(文中也称为模式单元元模型).图 4 主要针对本文所扩充的部分,给出了简化的模式元-元模型,下面给出EPattern和ERole这两个元-元模型的语义.

EPattern:是 Complete MOF 中 Classifier 的子类,定义了一个模式规约的边界,或者说表示了一个模式单元元模型的整体.一个模式单元元模型也就是一种设计模式的模式规约,它表示为一个 EPattern 的实例,其中包含了

© 中国科学院软件研究所 http://www.jos.org.cn

多个用角色表示的参与者(participant).角色使用 ERole 来定义,它们用来描述一个模式的结构以及各个参与者的责任.针对某一种设计模式构造其模式规约的过程就是实例化一个 EPattern 的过程,这个过程同时也包含了其中 ERole 的实例化过程.当 EPattern 的实例化过程完成后,该实例就表示了一种设计模式的规约,其中的 ERole 实例就表示了同一个设计模式的参与者.

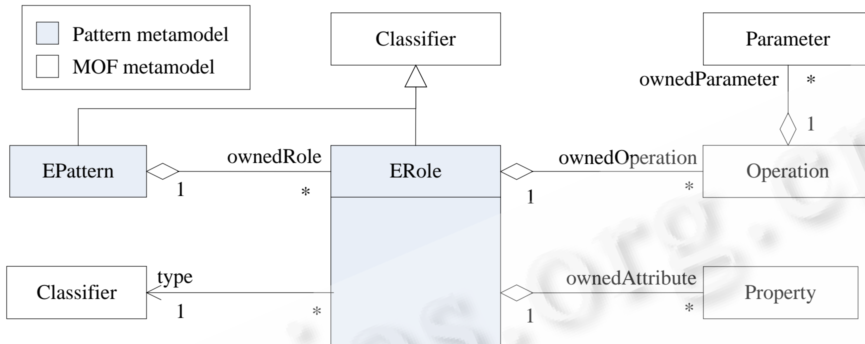


Fig.4 Pattern meta-metamodels

图 4 模式元-元模型

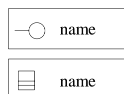
ERole:是 Complete MOF 中 Classifier 的子类,它用来在模式规约中定义模式参与者.通过复用和扩展 Classifier 的定义,ERole 在抽象语法结构上具有多个操作(operation)和属性(property).在语义方面,ERole 表示模式规约中用来定义参与者的元模型,每个参与者只能对应一个由 Classifier 定义的类型.以 UML 2.0 为建模的目标元模型时,每个参与者的类型是类(class,from kernel)或接口(interface,from kernel).如果该参与者是一个类元模型的话,那么该参与者在 M1 层上的实例就是一个 UML 中的类,绑定该参与者实例的业务模型也就只能是类;如果该参与者是一个接口的话,那么该参与者在 M1 层上的实例就是一个 UML 中的接口,而绑定该参与者实例的业务模型也就同样只能是接口.

2.3 模式元模型的图形符号

EPattern 和 ERole 的图形符号采用与 UML 2.0 相似的风格,这样有利于在以 UML 2.0 为目标元模型的建模环境中进行模式规约的构建.该组图形符号同时也作为模式实例的图形表示法.由于 ERole 在 M1 层的实例为 Class 或 Interface,因此,其 M1 层符号与 UML 2.0 Superstructure 中相同.

EPattern:一个由名称部和结构部组成的矩形虚线框.上半部分是名称部,显示该模式的名字;下半部分是结构部,包含该模式的组成结构.模式的组成结构通过 ERole 定义.

ERole:一个由名称和类型标示组成的矩形实线框.当类型标示为 −○时,表示该参与者的类型是一个接口(interface,from kernel);▤表示该参与者的类型是一个类(class,from kernel).



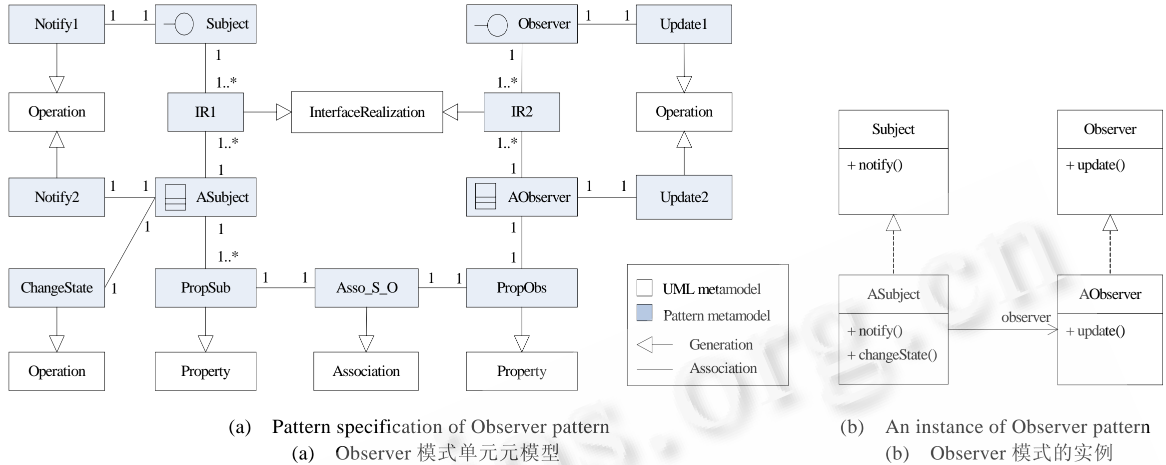
2.4 Observer模式单元元模型

Observer 模式是 GoF 的 23 种模式中的一种常见的行为模式(behavioral pattern),它通过定义多个对象对某一对象的依赖关系,来保证当该对象的状态发生变化时,其他对象的状态也会随之发生变化.

Observer 模式单元元模型是在 M2 层定义 Observer 模式的元模型结构,其中使用了扩展的 MOF 模式元模

© 中国科学院软件研究所 http://www.jos.org.cn

型以及 UML 2.0 Superstructure 中定义的元模型.下面给出简化的 Observer 模式规约模型及实例,如图 5 所示.



(a) Pattern specification of Observer pattern
(a) Observer 模式单元元模型

(b) An instance of Observer pattern
(b) Observer 模式的实例

Fig.5 Pattern specification and instance of Observer pattern
图 5 Observer 模式元模型及实例

图 5a 中深色部分是位于 M2 层的 Observer 模式规约模型,它们都是通过扩展 M2 层 UML 元模型(如图 3 所示)以及实例化 M3 层 ERole 元模型而得到的.为了简化起见,图中省略了 EPattern 实例的定义.默认情况下,所有位于同一个模式规约中的 Pattern 元模型都处于同一个 EPattern 实例中.值得注意的是,图中通过定义 ASubject 和 PropSub 之间的一对多关系(表示为 1:*),规定了每个 ASubject 可以对应一个或多个 AObserver.第 4 节还会对此作进一步说明.该模型图定义了 Observer 模式在 M2 层的规约,它在 M1 层的实例化就是具体的 Observer 模式实现形式.图 5(b)是该规约对应的一个模式实例.其中,ASubject 和 AObserver 被实例化为一对一的形式.

2.5 RolePlay机制

当使用本文所扩展的模式元模型定义模式规约时,相当于是在 M2 层进行元模型建模,也就是定义模式单元的元模型.每一种设计模式都对应一个模式规约,该规约可以实例化为 M1 上的具体模式模型.这样建模所得到的结果就是可以独立于业务逻辑的模式逻辑,而业务逻辑则可以通过使用像 UML 这样的建模语言来单独创建.以独立的方式对模式逻辑和业务逻辑分别建模可以避免出现模式的追溯性以及组合等问题,但需要提供一种方式来连接两者,使之成为一个整体.对此,我们引入了一种基于角色依赖关系的 RolePlay 机制,在 M1 层连接以 UML 2.0 为目标语言的业务模型与模式模型,实现对模式逻辑和业务逻辑的绑定.图 6 给出了 RolePlay 关系的元模型.给出 RolePlay 的元模型并非单纯为了定义 RolePlay 的语义,另一个重要目的是为了在后面的转换中能够通过 RolePlay 关系将捆绑在一起的模式模型与业务模型转换到同一个 PSM 上(第 3.3 节将对此作进一步讨论).

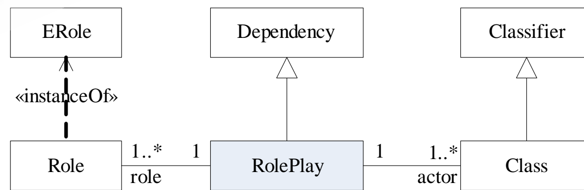
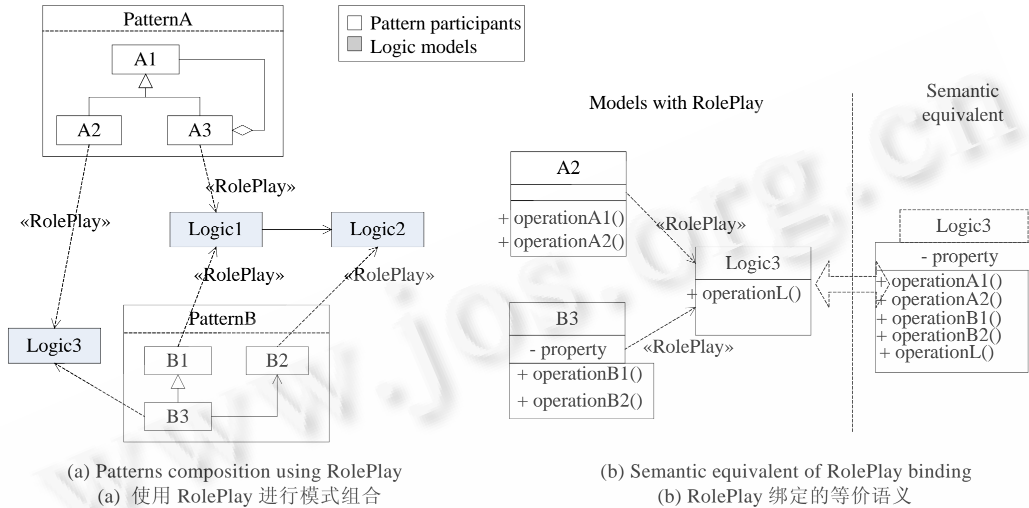


Fig.6 RolePlay metamodel
图 6 RolePlay 关系的元模型

RolePlay 被定义为一条从 Role 到 Class 的关系.其中 Role 为 ERole 的实例,表示模式规约中的参与者元模

型,Class 表示使用 UML 等建模语言定义的业务模型.RolePlay 的图形符号为一条单向虚线空心箭头,并且在虚线上用«RolePlay»标识.RolePlay 关系是一个多对一的关系,它表示一个业务模型同时可以担任一个或多个设计模式中的参与者角色(role[1..*]端).RolePlay 关系的绑定方是作为参与者的 ERole 实例,被绑定方是业务模型.ERole 实例只能绑定一个业务模型(actor[1]端),而一个业务模型却可以被处于不同模式规约实例中的 ERole 所绑定.通过多次绑定,可以实现对设计模式的组合使用,如图 7(a)所示.



(a) Patterns composition using RolePlay
(a) 使用 RolePlay 进行模式组合

(b) Semantic equivalent of RolePlay binding
(b) RolePlay 绑定的等价语义

Fig.7 RolePlay mechanism and its binding semantics
图 7 RolePlay 机制及其绑定语义

RolePlay 关系从语义上将模式逻辑中参与者所拥有的方法和属性添加到了被关联的业务模型上.这种添加方式同时也保证了模式参与者之间的结构关系被带入到了业务模型中.因此,被同一个模式规约实例的参与者所绑定的业务模型也处于同一个模式逻辑中.如果同一个业务模型被多个模式的 ERole 实例绑定,则所有 ERole 实例中的方法和属性都会被添加到该业务模型中,如图 7(b)所示.在定义 RolePlay 时,限定被绑定的业务模型为 Class 的实例是为了避免语义添加过程中接口和类之间的冲突.另外,避免命名冲突的问题,我们规定业务模型中的操作和属性不能与模式模型中的相同.

3 面向设计模式的模型转换

本节围绕着单元化模式模型转换的问题,首先介绍了模型转换中的源和目标,其中包含了转换规则的定义和实施两个层次,然后给出了目标平台的元模型.最后,本节还给出了基于 QVT 标准定义的转换规则.

3.1 模型转换中的源与目标

J2EE 是企业级的中间件平台,它针对分布式多层的软件系统提供了一整套技术标准,在业界得到了广泛的支持.本文选择了其中的核心标准 EJB 作为目标的映射平台,以模式为单元定义了 PIM-EJBPSM 转换规则.

Table 1 Source and target in model transformation
表 1 模型转换中的源与目标

	Source	Target
Metamodel	Pattern specification, M2	EJB metamodels, M2
Model	Pattern models, M1	EJB models, M1

在 MDA 框架标准中,QVT 是专门支持模型的查询、视图和转换的标准,它被集成到 MOF 标准族中,可以应用到任何 MOF 模型体系下的模型转换中.在 QVT 标准中,模型转换规则是建立在元模型层,而应用则是针对模型层.因此,要定义设计模式规约向 EJB 映射的模型转换规则就首先要明确源模型(source model)和目标模型

© 中国科学院软件研究所 http://www.jos.org.cn

(target model)以及它们的元模型,见表 1.

表 1 中列出的模式规约模型是 EPattern 元模型在 M2 层的实例,它定义了特定的设计模式的结构.模式实例模型是以模式规约为元模型产生的模式实例.

3.2 EJB元模型

我们基于 SUN 的 EJB2.1 规范[7]以及 OMG 的规范[9]给出了使用 UML 形式定义的 EJB 元模型,如图 8 所示.EJB2.1 中的 Enterprise Bean 共有 3 种,本文的转换部分主要针对其中的 Session Bean 和 Entity Bean 进行映射,因此,图中主要围绕这两个 Bean 给出了相关的元模型.

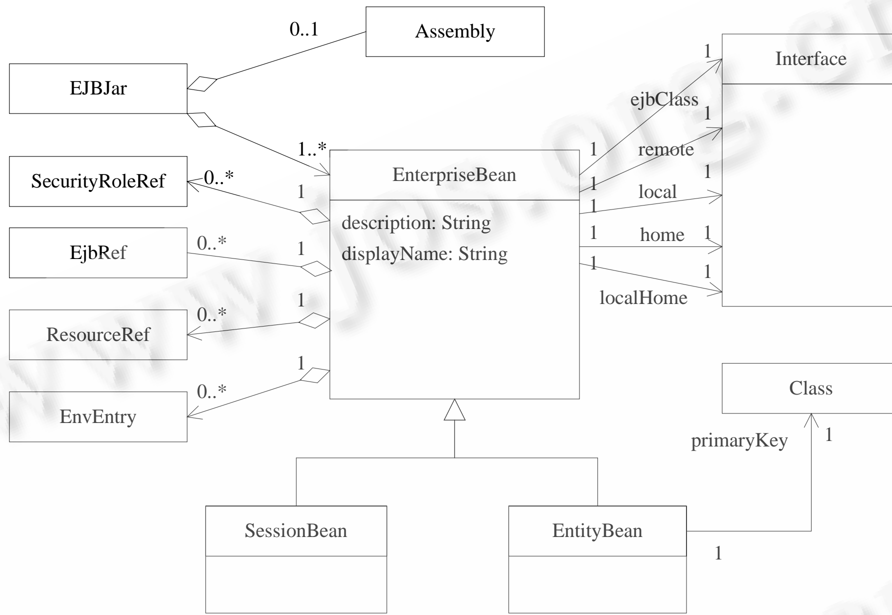


Fig.8 EJB metamodels

图 8 EJB 元模型

EJB 2.1 规范中定义了 6 种角色在 EJB 的整个应用开发和部署担任不同的职责.它们分别为:Bean 供应商、应用程序组装商、EJB 系统部署员、系统管理员、容器和服务器供应商以及工具供应商.本文框架在进行 EJB 转换过程中主要担任了 Bean 供应商的角色,负责开发和提供商务组件或 Enterprise Beans.因此,转换部分所关注的 EJB 模型主要是图 8 中定义的 SessionBean 和 EntityBean 及相应的接口.

3.3 QVT转换规则

QVT 标准中提供了两种声明式(declarative)语言,一种命令式(imperative)语言及黑盒机制来定义模型转换规则.声明式语言便于描述源模型到目标模型的对应关系,而命令式语言更易于描述转化过程.本文使用声明式的关系语言(relation language)给出了 Observer 模式模型到 EJB 模型的映射关系,同时还使用命令式的操作映射语言(operational mapping language)给出了 RolePlay 关系绑定后的模式及业务模型到 EJB 模型的转化过程.在基于本文的支撑工具的开发上,前者可以构建直观的映射视图,后者可以用来实现转换引擎.

在关系语言描述的转换规则中,使用域(domain)定义源模型组和目标模型组,使用关系(relation)和对象模板表达式(object template expression)定义其中具体模型之间的对应关系.源模型和目标模型之间的关系映射就定义为域到域的对应关系,而其中具体模型之间的映射则定义为对象表达式匹配的对象之间的关系.下面给出了 Observer 模式的 PIM 到 EJB-PSM 的部分映射规则.

© 中国科学院软件研究所 http://www.jos.org.cn

规则 1. 关系映射规则.

```
1. transformation ObserverPatternToEJB(pattern: ObserverPattern, ejb: EJB)
2. {
3. top relation SubjectToInterface{
4.     checkonly domain pattern s: Subejct {name='Subject'};
5.     enforce domain ejb i: Interface {name='Subject'};
6. }
7. relation Notify1ToOperation{
8.     checkonly domain pattern n: Notify1 {name='notify'};
9.     enforce domain ejb o: Operation {name='notify', /type=OclVoid, class=i};
10. when{SubjectToInterface(s,i);}
11. }
12. ...
13. }
```

其中,第 1 行的关键字 transformation 定义了一个从 ObserverPattern 域到 EJB 域的映射,第 3 行的 relation 关键字定义了 ObserverPattern 域中的 Subject 到 EJB 域中的 Interface 的映射,前面的 top 表示该关系是一个必须首先被满足的关系.第 4、第 5 两行中的{name='Subject'}就是对象模板表达式,它们表示匹配的条件是 name 属性的值为字符串'Subject'.第 7~第 11 行定义了一个从 ObserverPattern 域中的 Notify1 到 EJB 域中的 Operation 的映射,其中第 10 行的关键字 when 表示当 SubjectToInterface 关系满足时,该关系也必须满足.

需要特别指出的是,由于使用模式单元建立的模型必须与业务模型捆绑之后才能表示具体的业务信息,因此用来绑定两者的 RolePlay 就成为将两者转换为同一个 PSM 的关键衔接点.我们已经定义了 RolePlay 的元模型为一个扩展的 Dependency 关系,它的两端分别为 ERole 类型的 role 和 Class 类型的 actor.其中,role 端连接模式模型,actor 端连接业务模型.通过在 PIM 中读取 RolePlay 关系的 role 端和 actor 端,就可以获得被同一个 RolePlay 绑定的模式模型和业务模型,进而完成两者到同一个 PSM 的转换.

下面首先给出将 Observer 模式模型及被绑定的业务模型转换到 EJB 模型的基本算法,然后再给出使用操作映射语言定义的 Observer 模式模型向 EJB 模型转换的操作映射规则的关键部分.

将 Observer 模式模型及被绑定的业务模型转换到 EJB 模型的基本算法如下:

步骤 1: 根据 RolePlay 关系合并模式模型及业务模型,生成临时类 CombClass.

a) 读取 RolePlay 关系(其元模型如图 6 所示)的 actor(被绑定方,即业务模型)和 role(绑定方,即 Observer 模式模型);

b) 将 actor 中的方法和属性都添加到 role 中.

步骤 2: ERole 的实例 Subject 和 Observer 的类型都是接口类型,因此直接转换为 EJB 中的普通接口,并保持原有的操作不变.

步骤 3: 将合并后的 CombClass 转换为 SessionBean.

a) 生成相应的 Remote 接口与 Local 接口,并设置其操作为 CombClass 的操作;

b) 生成相应的 RemoteHome 接口与 LocalHome 接口,并添加 create()操作;

c) 生成相应的 SessionBean 类,并设置其属性为 CombClass 的属性;

d) 在已生成的 SessionBean 类中添加 ejbCreate(),ejbRemove(),ejbActivate(),ejbPassivate()以及 setSessionContext(SessionContext sc)操作.

Observer 模式模型向 EJB 模型转换的操作映射规则如下:

规则 2. 操作映射规则.

```
1.  transformation ObserverPatternToEJB(in srcModel: ObserverPattern, out: EJB);
2.  query ObserverPattern::Role.isRolePlayed() Boolean{
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
3.     rp:=self.ownedMember->selsect(p| #RolePlay);
4.     return rp->exists(rp.role=self);
5.  }
6.  intermediate class ObserverPattern::CombClass{type: Classifier}
7.  main() {
8.     srcModel.objectsOfType(RolePlay)->map combinedByRolePlay();
9.     srcModel.objectsOfType(Interface)->map interface2EJB();
10.    srcModel.objectsOfType(CombClass)->map combClass2Remote();
11.    srcModel.objectsOfType(CombClass)->map combClass2Local();
12.    srcModel.objectsOfType(CombClass)->map combClass2RemoteHome();
13.    srcModel.objectsOfType(CombClass)->map combClass2LocalHome();
14.    srcModel.objectsOfType(CombClass)->map combClass2EJBClass();
15. }
16. mapping RolePlay::combinedByRolePlay(): CombClass{
17.    var _actor:=self.actor;
18.    var _role:=self.role;
19.    name:='comb_'+self.name;
20.    result.ownedAttribute:=_actor.ownedAttribute;
21.    role.ownedAttribute->forEach(i){
22.       result.ownedAttribute->add(i);
23.    }
24.    result.ownedOperation:=_actor.ownedOperation;
25.    role.ownedOperation->forEach(i){
26.       result.ownedOperation->add(i);
27.    }
28.    result.type:=_role.type;
29.    }
30.    ...
```

其中,第 2~第 5 行定义了一个查询函数,用来判断当前的 Role 模型是否连接有 RolePlay 关系.如果连接了 RolePlay 关系,则返回 true,否则返回 false.第 6 行 intermediate 关键字表示定义了一个临时的中间变量类型,中间变量主要用来保存根据 RolePlay 关系合并的模型,在转换结束前,该变量就会被释放掉.第 16~第 29 行通过关键字 mapping 定义了合并 RolePlay 关系所连接模型的操作映射,该操作映射所返回变量的类型就是 intermediate 定义的中间变量类型.

第 7~第 15 行通过关键字 main 定义了该转换规则执行时的入口点(类似于 Java 中的主程序).该部分包含了 7 个操作映射,它们的执行顺序与它们在规则中出现的顺序是一致的.其中,第 8 行的操作映射反映了基本思路中的步骤 1,第 9 行对应于步骤 2,余下的第 10~第 14 行对应于步骤 3.

4 实例研究

本节通过对股票交易(stock transaction)系统中一个场景的开发实例来具体说明如何将模式作为完整的开发单元进行 MDA 框架下的软件开发.该实例主要分为两个部分,首先是在 PIM 层建模,得到平台无关模型;然后利用映射规则将平台无关模型转换为 J2EE 上的平台相关模型.

© 中国科学院软件研究所 http://www.jos.org.cn

4.1 场景描述

在网上的股票交易系统中,需要即时地反映股票价格的变化.无论是统计性的股票价格列表,还是反映单支股票的走势图,都必须随股票价格的变动作即时的更新.图 9 是股票系统的这种数据-视图形式的示意图.图中的股票数据于前台视图之间保持一致的方式是以股票数据层为主导的.当股票价格发生变化时,数据层主动向视图层发出更新要求,视图层收到消息后,读取数据层的股票价格,然后根据各自不同的视图要求将其展示出来.

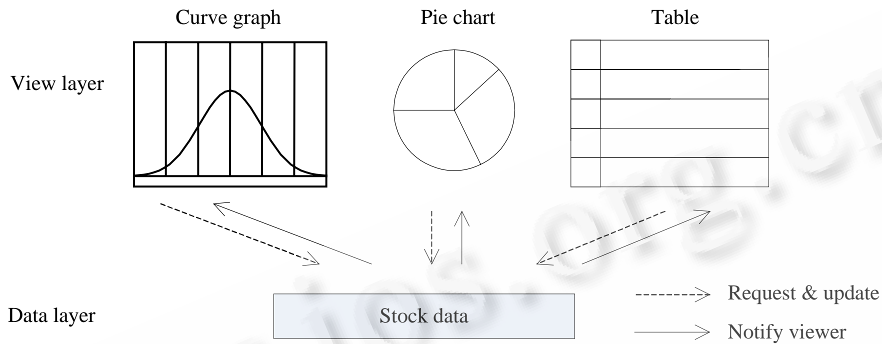


Fig.9 Overview of data-view in stock system

图 9 股票系统数据-视图示意图

4.2 PIM建模

在 MDA 的开发思想下,PIM 层建模是设计过程中最为重要的环节之一.该环节交付的软件制品是系统的业务逻辑在平台无关抽象层上的模型.

股票信息的视图更新涉及到两个部分的内容,一个是对数据层的定义,另一个是对表示层的定义.为了简化问题,本文只抽象了这两部分的关键信息进行建模.图 10 给出了该场景业务类的定义.其中 Stock 表示股票数据模型,StockView 定义了所有视图形式的统一接口,其中的 display()方法表示对股票价格信息的显示.StockView 有 3 个实现体,分别用来定义曲线图(curve)、饼图(pie)以及表格(table)的显示形式.

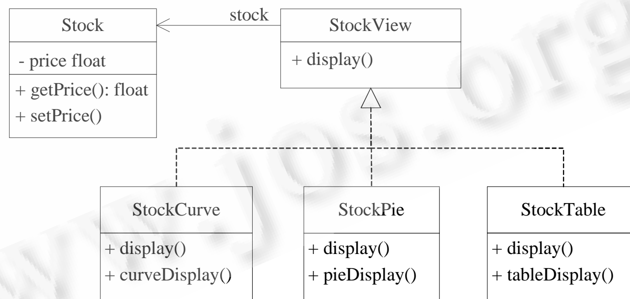


Fig.10 Business logic models

图 10 业务逻辑模型

根据业务逻辑的要求,视图部分要及时地反映股票价格的变动情况.而此类问题正是 Observer 模式的应用范畴,因此考虑到使用 Observer 模式来实现视图层与数据层的耦合.由于本文框架引入 RolePlay 机制实现业务逻辑与模式逻辑的分离,因此可以在保持业务逻辑不变的情况下直接对模式逻辑建模,然后再使用 RolePlay 关系绑定两者,完成对该场景的 PIM 建模.

为了引入 Observer 模式来解决数据-视图的耦合问题,必须首先根据业务逻辑的情况实例化 Observer 模式的模式规约,从而得到 Observer 模式模型.然后再将 Observer 模式模型与业务模型进行 RolePlay 绑定,进而得到

© 中国科学院软件研究所 http://www.jos.org.cn

该场景完整的 PIM.在第 2.4 节中已经给出了 Observer 模式的模式规约,下面就直接给出 Observer 模式在 M1 层的实例模型及其与业务模型的绑定,如图 11 所示.

图 11 的虚线框部分是 Observer 模式规约(第 2.4 节,图 5(a))的一个实例,其中一个 ASubject 聚合了 3 个 Observer 的实现类,分别是 AObserver1,AObserver2 和 AObserver3.之所以可以有这种一对多的实例化结果,是因为在 Observer 模式规约中定义的 ASubject 元模型和 AObserver 元模型之间是一对多的关系.通过对 AObserver 产生 3 个实例就可以对业务模型中的 StockCurve,StockPie 和 StockTable 这 3 个视图类分别进行 RolePlay 绑定,以保证它们处于同一个模式中.

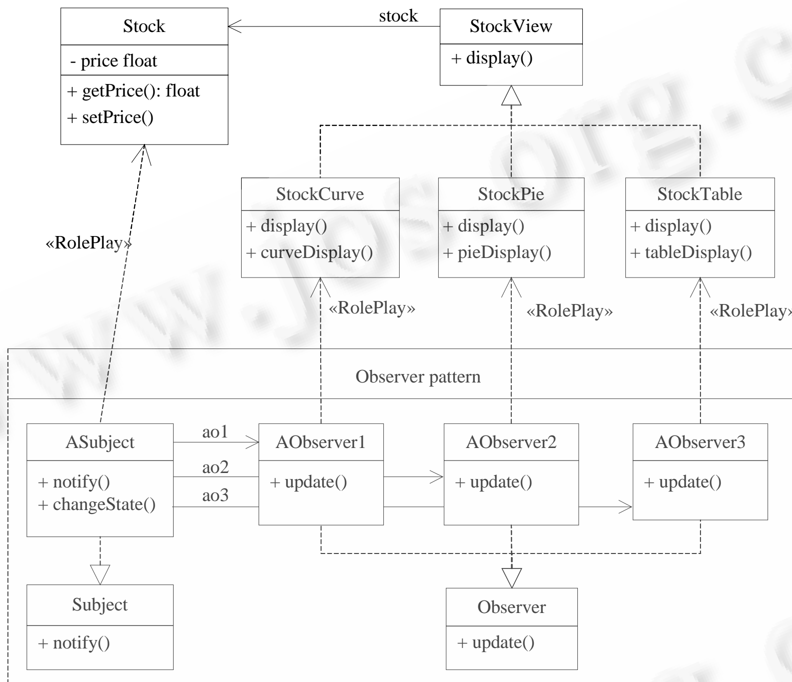


Fig.11 PIM of stock view updating scenario

图 11 股价视图更新场景的 PIM

通过对 Observer 模式规约的整体实例化实现了以 Observer 模式为单元的建模,获得了模式逻辑模型.这个过程虽然在一定程度上是独立于业务逻辑建模的,但其目的是为了解决业务逻辑中的具体问题,因此从本质上讲仍然是依赖于业务模型的.但通过明确地划分业务逻辑模型与模式逻辑模型,可以在很大程度上解决模式的可追溯性以及组合问题,这有助于我们更加灵活地进行单元化设计模式建模的活动.

4.3 PIM-PSM映射

在完成了 PIM 层的建模工作之后,就可以应用转换部分中定义的 PIM-EJBPSM 转换规则将 PIM 映射为相应的 EJB 平台上的 PSM.前面已经介绍了基于 QVT 标准对模式规约定义的 EJB 转换规则,在第 3.3 节中也给出了针对 Observer 模式规约的 QVT 转换规则,这里,我们主要使用规则 2 的操作映射规则将图 11 股票视图更新场景的 PIM 转换到 EJB 平台上.

对于规则 2 而言,源模型为图 11 中的 Observer Pattern(虚线框内的部分)及相应通过 RolePlay 关系连接的部分.因为具体规则的制定是针对模式单元以及相应 RolePlay 关系所绑定的模型,因此,其他部分的模型不受规则的制约.

应用操作映射规则时首先从 main()定义的入口点部分开始执行,顺序调用其中的操作映射并同时产生出目标模型,当调用完所有的操作映射后,规则的执行也即结束.对图 11 的 PIM 应用规则 2 时首先执行第 8 行的

© 中国科学院软件研究所 http://www.jos.org.cn

映射,读取源模型中的所有 RolePlay 关系,然后调用 combinedByRolePlay()合并模式模型与业务模型.在这里,源模型中会增加 4 个临时产生的中间模型,分别用来保存 Stock 和 Asubject,StockCurve 和 AObserver1,StockPie 和 AObserver2 以及 StockTable 和 AObserver3 合并的结果.接下来,规则执行到第 9 行,读取源模型中所有的接口,然后将其转换为 EJB 上的接口.这里,目标模型中会得到 Subject 和 Observer 两个 EJB 上的接口.第 10~第 14 行都是针对合并后的临时模型进行转换,产生相应 EJB 平台上的 SessionBean.在这里,目标模型中会得到 4 个 SessionBean 组件,每个 SessionBean 组件中都包含对应的 Remote 接口、Local 接口、RemoteHome 接口、LocalHome 接口以及 SessionBean 类.图 12 给出了规则 2 执行后得到的所有目标模型.

上面给出的是一个具体 Observer 模式模型的自动转换过程.对于完整的开发而言,可能会用到若干个设计模式以及相应的自动转换,但在每一次应用模式单元转换规则进行转换时,都只会将相应的模式模型及通过 RolePlay 绑定的模型进行转换.因此,完整的系统开发可能会涉及到多次以模式为单元的自动转换,但剩余部分模型的转换仍需要通过其他途径实现.

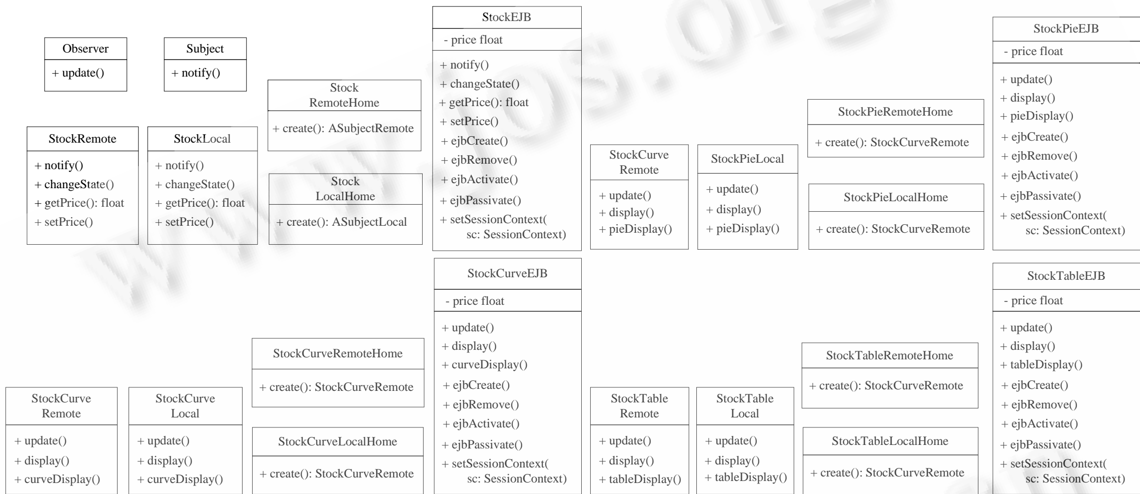


Fig.12 EJB models of Observer pattern after transformation

图 12 转换后的 Observer 模式相关的 EJB 模型

4.4 应用效果分析

实例研究的应用效果主要体现在建模和模型转换两个部分.在建模方面,通过以设计模式为开发单元整体进行建模,降低了传统方式实例化设计模式时的工作量,同时提高了模型的可读性.此外,通过支持模式组合,进一步提高了在应用多个模式进行建模时,系统模型的可理解性.

限于篇幅,文中使用的实例较为简单,因此这方面的效果不是非常明显.当开发复杂的应用系统时,模式单元所带来的模型可读性会极大地提高开发的效率,提高设计、开发以及维护人员对模型理解的准确性,从而降低开发工作量.

在模型转换方面,通过向特定平台的自动映射,降低了传统手工编写代码的工作量.同时,设计人员可以利用自动转换方便地看到设计方案在具体平台上的实现效果,有利于检查设计方案的正确性和合理性,从而提高了设计效率.

5 相关工作比较

本文工作的目的是要在 MDA 的框架下将设计模式作为完整的开发单元来使用,以此提高建模粒度和转换规则的复用程度,这与以往围绕设计模式的工作有着很大的不同.这种不同之处的根源在于,MDA 与传统软件开发方式的区别.在 MDA 的框架下,模型不仅作为软件系统的设计文档和规格说明来使用,更被用来直接导

© 中国科学院软件研究所 http://www.jos.org.cn

出最终可运行的系统,因此,建模和模型转换就具有了更加重要的意义.

本文的工作主要体现在利用设计模式对 MDA 框架下的建模和模型转换提供进一步的支持,而传统的设计模式相关工作在正向工程(forward engineering)方面主要是针对着模式的规约和识别[10−12],在逆向工程(reverse engineering)方面主要是利用模式理解程序[13,14]和精化程序[15].虽然目前一些扩充 UML 等 MOF 相容的建模语言以支持设计模式的工作也可以看作是将设计模式引入到 MDA 中,但与本文的单元化使用设计模式的工作仍有很大区别.

文献[6]提出一种基于角色的设计模式建模和实现方法,通过分离模式逻辑与业务逻辑来解决模式的文档、组合及实现问题.该方法虽然在建模层面上分离了模式模型和业务模型,但未能从元模型层揭示设计模式的本质特性,所以也不支持以模式为单位的整体建模.但文献[6]中基于角色来分离模式逻辑与业务逻辑的思想为本文模式建模支撑机制的提出提供了思路.

文献[10]基于高阶一元逻辑定义了 LePUS,以形式化的方式描述设计模式.该方法的缺点主要是复杂,难以掌握.与之相比,我们的模式规约方法具有更加直观的特点.文献[12]将 LePUS 引入到了 UML 中,结合了两者的优点,提高了 LePUS 的实用性.然而,该方法并未从根本上消除 LePUS 的复杂性,尤其是当定义模式组合时,建模结果会格外复杂.尽管如此,他们的工作仍在一定程度上揭示了设计模式的本质特征,为针对模式的规约和建模方面的工作提供了依据.

文献[11]提出了设计模式在结构上的 4 种本质特征,然后通过 UML 的 Profile 机制扩充 UML 语言,实现了对设计模式结构上的精确描述.然而在实际建模中,我们发现该方法虽然可以精确地描述模式,却不便于明确地区分模式逻辑和业务逻辑.当存在模式组合时,这种情况就更加突出了.我们的工作采用类似文献[6]的方法分离了模式逻辑与业务逻辑,在建模上更易于识别模式和组合模式.另外,文献[12,13]的工作都是借助了 UML 1.4 中的协作图(collaboration diagram)来描述模式元素之间的结构关系.协作图在 UML 1.4[16]中提供了对设计模式的专门支持,但在 UML2.0 中却被弱化了这方面的能力.而本文则是通过扩充 MOF 的元-元模型建立模式规约,避免受到建模语言版本变化的影响.

文献[17]通过扩充 MOF 元-元模型定义了 EPattern 模式规约语言,并基于该语言提供了模式识别生成算法,用以在模型中识别所包含的设计模式.该工作的主要意图仍然在于识别模式、理解模型,这与本文的工作有着很大的差异.该工作对使用设计模式进行建模方面的支持还不够,因此无法直接引入到本文的工作中.但是该工作基于 MOF 元-元模型扩展的思想和方法都具有很好的借鉴意义,为我们设计模式单元规约提供了思路.

6 结 论

MDA 是一种新型的软件开发方法,它与传统的软件开发方法相比更加重视了对模型的充分利用,希望以模型为开发要素,进一步提高软件开发的抽象层次.设计模式是一种成熟的技术,为面向对象的软件开发提供可重用的设计方案.本文工作结合了这两者的优点,在 MDA 的框架下引入设计模式,将其作为完整的开发单元进行建模和转换,在一定程度上提高了 PIM 的建模粒度,降低了 PIM-EJBPSM 转换规则的开发工作量.

我们认为本文工作的主要贡献在于:在将设计模式引入到 MDA 的过程中,把设计模式作为独立的建模单元来使用,以此提高了 PIM 层开发的建模粒度;为设计模式单元提供可复用的转换规则,使 PIM 层的模式及相关业务模型能够通过重用规则自动地转换为 EJB 平台上的 PSM.

此外,通过使用本文提供的设计模式单元进行模型驱动开发,还可以在设计模型(体现为 PIM)中灵活地支持模式组合,并实现模式的可追溯性.其中,对模式组合的支持主要依靠 RolePlay 建模机制以及相应的转换规则来实现.但在转换之后的平台相关模型中,则无法实现模式的追溯性.

在下一步的工作中,我们将考虑增加补充性的转换规则,以支持对系统全面的模型自动转换.目前的转换规则主要针对模式单元模型以及建模中通过 RolePlay 绑定的模型,剩余部分的模型仍无法自动转换,因此,增加的补充性转换规则将主要针对这一部分的剩余模型进行自动转换.另外,我们还将增加对模式行为方面的支持以及模型转换的验证工作.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Miller J, Mukerji J. MDA guide version 1.0.1. OMG, 2003. http://www.omg.org/docs/omg/03-06-01.pdf

[2] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[3] OMG. Meta object facility core specification v2.0, 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf

[4] OMG. Unified modeling language: Superstructure v2.0, 2005. http://www.omg.org/docs/formal/05-07-04.pdf

[5] OMG. MOF Query/Views/Transformations final adopted specification, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf

[6] He CW, He KQ. A role-based approach to design pattern modeling and implementation. Journal of Software, 2006,17(4):658−669 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/658.htm

[7] Sun Microsystems. Enterprise JavaBeans Specification v2.1. http://java.sun.com/products/ejb/docs.html

[8] OMG. Unified modeling language: Infrastructure v2.0, 2006. http://www.omg.org/docs/formal/05-07-05.pdf

[9] OMG. Metamodel and UML profile for Java and EJB specification v1.0, 2004. http://www.omg.org/docs/formal/04-02-02.pdf

[10] Eden AH. Precise specification of design patterns and tool support in their application [Ph.D. Thesis]. Tel Aviv University, 2000.

[11] Mak JKH, Choy CST, Lun DPK. Precise modeling of design patterns in UML. In: Giacobazzi R, ed. Proc. of the 26th Int'l Conf. on Software Engineering (ICSE'04). Washington: IEEE Computer Society, 2004. 252−261.

[12] Guennec AL, Sunyé G, Jézéquel JM. Precise modeling of design patterns. In: Evans A, Kent S, Selic B, eds. Proc. of the 3rd Int'l Conf. of Modeling Languages, Concepts, and Tools (UML 2000). LNCS 1939, Springer-Verlag, 2000. 482−496.

[13] Agustín F, Espinoza C. Automatic design patterns identification of C++ programs. In: Shafazand MH, Tjoa AM, eds. Proc. of the 1st Eur-Asian Conf. on Information and Communication Technology. LNCS 2510, Springer-Verlag, 2002. 816−823.

[14] Arcelli F. Elemental design patterns recognition in Java. In: Kontogiannis K, Zou Y, Penta MD, eds. Proc. of the 13th IEEE Int'l Workshop on Software Technology and Engineering Practice (STEP'05). Washington: IEEE Computer Society, 2005. 196−205.

[15] Kerievsky J. Refactoring to Patterns. Addison-Wesley, 2004.

[16] OMG. Unified modeling language specification v1.4, 2001. http://www.omg.org/docs/formal/01-09-67.pdf

[17] Elaasar M, Briand LC, Labiche Y. A metamodeling approach to pattern specification. In: Proc. of the 9th ACM/IEEE Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS 2006). LNCS 4199, Berlin, Heidelberg: Springer-Verlag, 2006. 484−498.

[18] France R, Kim DK, Ghosh S, Song E. A UML-based pattern specification technique. IEEE Trans. on Software Engineering, 2004, 30(3):193−206.

附中文参考文献:

[6] 何成万,何克清.基于角色的设计模式建模和实现方法.软件学报,2006,17(4):658−669. http://www.jos.org.cn/1000-9825/17/658.htm

张天(1978-),男,江苏南京人,博士生,主要研究领域为模型驱动软件工程.

王林章(1973-),男,博士,副教授,CCF 高级会员,主要研究领域为模型驱动软件工程,模型驱动软件测试与验证.

张岩(1974-),男,博士,讲师,主要研究领域为软件工程,形式化方法,模型驱动开发方法,软件度量.

李宣东(1963-),博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件建模与分析,软件测试与验证.

于笑丰(1976-),男,博士,讲师,主要研究领域为软件工程,模型驱动工程,服务工程,隐私访问控制.

© 中国科学院软件研究所 http://www.jos.org.cn