分类号	TP311	密级	公开
UDC	004.41	学位论文编号	D-10617-308-(2018)-12006

重庆邮电大学硕士学位论文

中文题目	基于模型驱动架构的 Web 代码生成方法研究与应用
英文题目	**Research and Application of Web Code Generation Method Based on Model Driven Architecture**
学　　号	**S151201006**
姓　　名	罗异
学位类别	工学硕士
学科专业	软件工程
指导教师	张力生　教授
完成日期	**2018 年 6 月 10 日**

摘要

对象管理组织(Object Management Group, OMG)提出的模型驱动架构(Model Driven Architecture, MDA)思想强调了模型在软件开发过程中的重要性，它的核心是通过模型转换思想构建软件系统。目前 Web 应用程序愈加庞大和复杂，开发人员往往花费大量时间编写 Web 表示层代码，导致开发效率低下。本文基于 MDA 中的模型表示和模型转换思想，研究了领域模型、对象模型和状态模型的表示以及领域模型到对象模型、对象模型到状态模型、状态模型到 Web 表示层框架代码的转换规则，提出一种基于模型驱动架构的 Web 代码生成方法，从而解决了上述问题。

可能也是不包含什么逻辑

首先，本文研究了模型驱动软件开发的基本思想以及包含的元对象机制(Meta Object Facility, MOF)和统一建模语言(Unified Modeling Language, UML)标准，研究了基于 MDA 标准的“模型实例转换”和“应用设计模式的转换”两种转换思路，研究了策略、组合和观察者设计模式与 MVC 的关系，研究了查询/视图/转换(Query/View/Transformation, QVT)模型转换描述语言的框架结构和使用方法。

其次，本文研究了在设计建模工具和使用建模工具建模的过程中，使用继承和实例化两种方式分别实现对象重用的内容，解决了在模型表示过程中建模工具设计人员和建模人员如何根据具体场景选择恰当的表示方式的问题，研究了采用继承和实例化方式将设计模式信息扩展到类图元模型中的方法，解决了如何在模型转换中加入设计模式信息的问题。

然后，本文在建模和代码生成阶段分别给出了类图到状态图、状态图到 JSF 表示层框架代码的转换规则，并使用 QVT 对转换规则进行描述，解决了对象模型到状态模型、状态模型到表示层框架代码的转换过程中，源模型与目标模型如何建立映射关系的问题。

最后，将本文提出的 Web 代码生成方法应用到程序设计自主学习系统场景中，在 PowerDesigner 平台实现领域模型到对象模型、对象模型到状态模型的转化以及状态模型到 JavaServer Faces 表示层框架代码的生成，验证了生成方法的可行性。

关键词：模型驱动架构，设计模式，领域模型，状态模型，表示层代码生成

Abstract

OMG's Model Driven Architecture emphasizes the importance of the model in the process of software development, It's core is to build a software system through model transformation. At present, Web applications are increasingly large and complex. The developers often spend a lot of time coding the interface of the Web presentation layer, which results in inefficient development. Based on the model representation and model transformation in MDA, the representation of domain models, object models, state models, and the transformation rules of domain model to object model, object model to state model, state model to Web presentation layer framework code are studied. Finally a Web code generation method based on model-driven architecture is proposed, which solves the above problems.

Firstly, the basic ideas of model-driven software development and the Meta Object Facility and the Unified Modeling Language specifications are studied. And the model instance transformation and join the design pattern transformation based on the MDA specification are studied. And then the relationship between strategy, composition and observer design patterns with MVC and the framework and usage of Query/View/Transformation model transformation description language are studied.

Secondly, the methods of inheritance and instantiation to implement object reuse in the process of designing modeling tools and modeling using modeling tools are studied, which solves the problem of how modeling tool designers and modelers select appropriate representation methods according to specific scenarios during model representation. And the methods for adding design pattern informations to class metamodel using inheritance and instantiation are studied, which solves the problem of how to add design mode information to the model transformation.

Thirdly, the transformation rules from class diagram to state diagram and state diagram to JSF presentation layer framework code in the modeling and code generation phase are respectively presented and their transformation rules are described by QVT, which solves the problem of how to establish the mapping relationship between the source model and the target model in the transformation process from the object model to the state model, the state model to the presentation layer framework code.

Finally, the method of Web code generation proposed in this thesis is applied to the scene of program design autonomous learning system. The PowerDesigner implementation platform is used to implement the code generation of domain model to object model, object model to state model, state model to JavaServer Faces presentation layer framework, which verifies the feasibility of the method.

Keywords: model driven architecture, design pattern, domain model, state model, presentation layer code generation

目录

第 1 章 引言

1.1 研究背景及意义

随着计算机技术，特别是 Intemet 技术的发展，基于 Web 的软件技术得到了空前的发展，各行业信息化进程日益加快，国内大小企业都在构建满足本行业发展的 Web 信息系统[1]。

Web 应用在企业信息化建设和人们的日常生活中占据着重要的地位，Web 应用开发也吸引着越来越多的软件开发人员，但在开发中总会出现的一些问题导致应用程序在维护性、扩展性、健壮性等方面达不到要求[2]。

MDA 的出现使模型在软件开发过程中起到了重要作用。基于模型驱动思想开发出的 Web 应用程序更加稳定并且往往能够满足用户真正的需求。所以，以模型为主导、快速生成原型、提高代码的重用性和可维护性将是今后 Web 应用开发的必然方向[3]。

2003 年，随着软件开发行业标准的确立和完善，OMG 正式提出 MDA 标准[4]，该标准不仅强调了模型在软件开发过程中的重要性，而且通过模型驱动的方式指导软件开发过程。MDA 的核心是模型的转换，首先建立计算无关模型(Computation Independent Model, CIM)，然后建立平台无关模型(Platform Independent Model, PIM)和平台相关模型(Platform Specific Model, PSM)，再将 PSM 生成代码，在特定环境中部署系统[5]。

UML 在 1997 年被 OMG 标准化[6]。UML 语言的出现建立了统一的面向对象开发方法。MOF 提供符号以完成 MDA 中模型和模型转换的表示[7]。MOF 是一个四层元模型架构[8]，M0 信息层是对客观世界抽象的结果，M1 模型层描述客观世界的多种可能性，它是对客观世界建模后的结果，M2 元模型层描述 M1 层的模型中使用的概念及其关系，它是一种描述模型的语言，M3 元元模型层，元元模型用来定义语言，可定义多种语言。

MDA 思想中模型通过元模型进行描述，源模型到目标模型的转换规则基于元模型进行定义。因此理解模型、元模型的概念以及模型转换思想和转换规则对模型驱动的代码生成有重要意义。

1.2 研究现状

应用软件开发过程中，编码人员经常需要重复编写一些简洁的、常用的代码。为了减轻程序员的编码负担、减少程序员的开发强度、缩短项目的开发周期，国内外研究者对代码自动生成技术进行了研究。MDA 模型驱动思想指导软件开发过程采用 CIM 到 PIM、PIM 到 PSM、PSM 到可运行代码的方法实现代码自动化生成。代码自动生成技术的应用十分广泛，不仅可以应用到自动生成简单的 Web 页面，还可以应用到开发企业级程序。

杨凌云等[9]在 2015 年为解决安全苛求系统通过采用冗余配置来增强系统的可靠性，但是冗余结构增加了安全苛求系统的复杂度。为了解决这个问题，他们采用构造型来扩展 UML 对系统冗余结构的描述并提供一个语义对照表，进而确定模型中的元素、关联和构造型语义以及相对应的故障树实现方法，提出一种与故障树生成相关的系统 UML 模型扩展方法。

A. Al-Alshuhai 等[10]在 2015 年为解决 UML 的活动图缺乏用于捕获上下文感知系统的上下文感知需求符号等问题，提出了具有新的符号的活动图的扩展，使其能够用于对系统功能进行分离，通过使用一些真实世界的案例证明了提出的扩展方法的可行性。

胡洁等[11]在 2016 年为了解决特征变更在建模过程导致的“涟漪”效应以及因此产生新的共性和可变性演化等问题，提出了一种特征模型扩展和演化分析方法，该方法通过扩展特征关联关系和模型演化元操作，实现对特征变更“涟漪”效应的分析，最后通过案例验证该方法的可行性。

模型转换指一组转换规则的集合，这些规则共同描述了如何转换源模型到目标模型的方法。模型转换是模型驱动架构的核心技术，用来解决模型到模型以及模型到代码间的映射问题[12]，MDA 的模型转换实际上是基于元模型的模型转换机制[13]。

张天等[14]在 2008 年为解决如何在 MDA 框架下以设计模式为单元进行建模和转换两个问题，首先扩展 CMOF 元元模型，得到自己定义的模式单元元模型。然后采用 RolePlay 绑定机制，解决了业务模型与模式模型的组合问题和模式单元建模问题。最后他们在该模式元模型的基础上定义了向 EJB 平台的转换规则，解决了以模式为单元的转换问题。

曾一等[15]在 2012 年为解决设计模式建模中存在模式消失和模式组合复杂化等问题，根据 MDA 和“角色”的建模思想，对设计模式中通用元素的元模型进行了扩展，提出了基于 Ecore 的设计模式建模以及模型转换的途径，为在 MDA 框架中围绕设计模式构建模型和实现模型转换提供了一种有效的指导。

马丽等[16]在 2015 年为解决复杂软件系统需求模型难以理解的问题，研究了基于行为描述语言构建的需求模型，通过定义转换规则，将该需求模型与状态图模型相关联，提出了一种 UML 状态图描述需求模型的可视化方法，对软件开发过程中保持模型间的一致性起到了一定的作用。

Y. Rhazali 等[17]在 2018 年为了完成将 CIM 转换到 PIM 的转换，在良好的 CIM 级别上定义了明确的转换规则，确保了半自动化转换的进行，进而提出了一种模型转换方法，该方法的主要价值在于指导了面相业务的模型向面向 Web 的模型的转换。

在编译器的设计和开发中出现了“自动代码生成”这个概念。编译器首先将源程序作为输入并将其翻译成一种中间产物，然后将中间产物作为代码生成器的输入，最后代码生成器以中间产物为基础，生成出语义一致的目标程序。软件工程领域的学者一直致力于代码自动化的研究[18]。

L. Zouhaier 等[19]在 2017 年为了提高用户界面与系统的交互能力，提出了一种基于元模型转的模型驱动方法，该方法生成适用于各种基于模态的用户界面模型。为了生成平台无关和特定的用户界面模型，还定义了一套元模型转换规则。

王建成等[20]在 2007 年为了促进基于模型的 UI 设计工程的进一步发展，研究模型的功能和组成并采用分层方法对用户界面进行抽象和呈现。提出了一种称为 EIP 的全新的 UI 模型，并在 Visual Basic，J2EE 和 ASP.Net 平台下实现了代码的自动生成。

蔡奎等[21]在 2009 年为解决基于模型的 Web 用户界面研究缺乏对复杂行为的模型设计和开发方法，极大的限制了此类方法的工程化应用等问题，对基于 Web 的用户界面中的复杂行为进行建模，提出了一种形式化的 Web 界面行为描述语言并实现了行为模型到代码的自动生成。

杨鹤标等[22]在 2010 年为了解决由于缺乏界面布局的准确描述，难以满足用户界面复杂性和个性化的需求等问题，研究了界面模型的特点和界面建模的技术，给出

了任务分析树到界面的抽象视图的转换规则并提出了支持界面自动生成的 ADS 模型，最终由框架解析生成界面。

王金恒等[23]在 2012 年为了解决业务逻辑层代码自动生成问题，分析了业务逻辑代码难以生成的原因，在研究平台相关模型到代码的转换过程中结合产生式规则思想，提出一种基于产生式规则的建模方法，该方法成功生成了业务逻辑与数据分离的代码，在一定程度上解决了业务逻辑代码自动生成问题。

B. Selimi 等[24]在 2015 年实现了基于文本管理的 Web 界面代码生成规范，其目标是提供不引人注目的自动代码生成。它基于设计模式，适合 Web 应用程序开发中的常见工作流程。

S. Roubi 等[25]在 2016 年为了解决实现具有符合用户意愿的图形用户界面困难等问题，提出一个模型驱动的开发过程来生成富网络应用程序(Rich Internet Applications, RIA)，实现图形用户界面的生成。

G. Kövesdán 等[26]在 2018 年通过模型表示和操作来实现灵活和健壮的代码生成器，提出了一个新的 DSL 代码生成器框架，它确定了使用 DSL 生成代码的常见工作流程，并为这些处理阶段提出了一组组件。

1.3 本文主要工作

本文研究了 MDA 的基本思想和开发流程以及其包含的 MOF、UML 等相关标准，重点对 MDA 中模型的表示和模型转换方法以及模型转换过程进行了研究。面向对象思想的核心是实现重用，本文着重研究了基于面向对象思想的 MDA 软件开发过程中 Web 表示层代码的生成方法。针对建模过程中如何选择实现重用的方法问题，研究继承和实例化两种实现重用的方法和它们分别重用的内容。针对如何在模型转换中加入设计模式的问题，研究在模型转换过程中加入设计模式的方法。最后通过 PowerDesigner 模型转换工具实现基于模型驱动架构的 Web 表示层代码的自动生成。

本文的主要工作如下：

1. 研究模型驱动架构、元建模机制、统一建模语言等相关标准，重点研究了基于模型驱动架构思想的软件开发过程中模型的表示和模型的转换原理和方法。

2. 根据面向对象重用的思想，研究继承和实例化两种将模型转换信息扩展到元模型中的方式。研究在软件建模和建模工具设计过程中两种方式实现重用的内容，结合软件建模人员和软件建模工具设计人员工作场景，讨论在实现重用内容上两种方式的区别。

3. 为解决模型转换中如何加入设计模式的问题，研究在模型转换中使用继承和实例化两种方式加入设计模式的模型转换方法。借助 PowerDesigner 工具，实现加入设计模式的模型转换。

4. 将基于模型驱动的 Web 代码生成方法应用于程序设计自主学习场景中，验证代码自动生成方法的有效性。

1.4 本文组织结构

本文共分为 5 章，各章内容安排如下：

第 1 章引言：介绍了本文的研究背景和意义、国内外研究现状、主要研究工作和组织结构。

第 2 章模型驱动架构软件开发相关知识：介绍了 MDA 的背景、MDA 中两个重要标准以及 MDA 模型转换类型，还介绍了表示层 MVC 框架应用的设计模式、描述模型转换的语言 QVT 和 PowerDesigner 提供的相关技术。

第 3 章 Web 代码生成方法研究：基于 MDA 中模型的表示和模型转换的思想，首先给出了 Web 代码生成整体思路，然后基于建模过程中的两个关键问题，讨论了通过继承和实例化扩展模型表达能力的方法以及两种方法在重用性上的区别，讨论了在模型转换中加入设计模式的方法，接着给出了模型转换的规则和过程描述，最后为了自动生成表示层代码，介绍基于模型驱动架构的 Web 表示层代码生成过程。

第 4 章 Web 代码生成方法的应用：介绍了 PowerDesigner 中提供的模型转换和代码生成技术。将程序自主学习系统作为实例，介绍了如何在 PowerDesigner 中实现领域模型到状态模型的转换以及状态模型到表示层代码的生成。

第 5 章总结与展望：对本文研究工作进行了总结，阐述了下一步的工作和计划。

第 2 章 模型驱动架构软件开发相关知识

本章介绍了 MDA 的背景以及 MDA 包含的 MOF 和 UML 两个重要标准。同时，介绍了 MDA 标准中给出的两种模型转换类型。最后介绍了表示层 MVC 框架应用的设计模式和描述模型转换的 QVT 语言相关知识以及 PowerDesigner 实现平台提供的相关技术。

2.1 MDA 背景

OMG 在 2001 年提出 MDA，旨在定义基于建模和自动将模型映射到实现的软件开发方法[27]。

MDA 标准 1.0 给出系统架构的定义，系统架构是系统部件和连接部件的连接器的标准，也是使用连接器进行交互的部件间的交互规则[28]。MDA 思想是系统开发的一种方法，它增加了模型在开发工作中的作用。MDA 思想的核心是模型转换，模型转换整体思路是将 CIM 转换到 PIM，再将 PIM 到 PSM，最后将 PSM 到可运行的代码。

MDA 标准 2.0 指出 MDA 主张将建模应用于架构过程，并将产生的构件形式化，使得系统的实现或改进可能更具有可操作性，其成本低，风险小[29]。MDA 是基于统一建模语言和其他行业标准的框架，用于可视化存储和交换软件设计和模型[30]。

2.2 MDA 中的标准

MDA 包含了 MOF、UML 和 CWM 等一系列标准。本节重点介绍 MOF 和 UML 两个重要的标准。

2.2.1 MOF

元模型是建模语言的一个模型，元建模是开发元模型的过程[31]。元建模拥有四层经典框架[32]，MOF 框架套用经典的元建模四层框架，它包含 M0 信息层、M1 模型层、M2 元模型层和 M3 元元模型层，如图 2.1 所示。

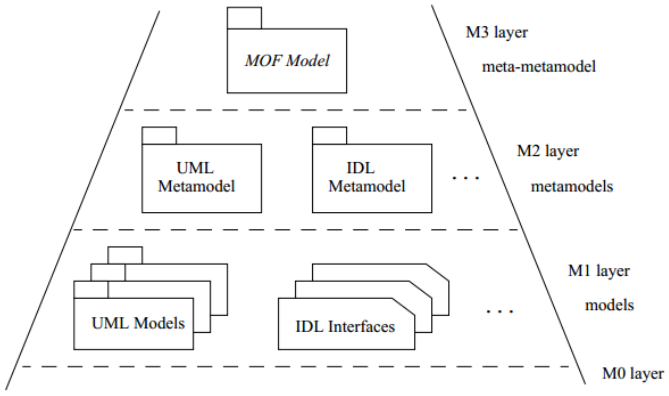


图 2.1 MOF 四层架构

M0 信息层是对客观世界的抽象。M1 模型层是对 M0 层的抽象，该层对应的模型由描述客观世界中事物和事物间关系的模型元素聚合而成，模型是对客观世界进行抽象后形成的结果。M2 元模型是对 M1 层的抽象，该层对应的元模型由描述模型中模型元素和元素间关系的元元素聚合而成，元模型是对模型抽象后形成的结果。M3 元元模型层是对 M2 层的抽象，该层对应的元元模型由描述元模型中元元素和元元素间关系的元元元素聚合而成，元元模型是对元模型抽象后形成的结果，该层元元模型采取自描述的形式减少了模型描述的复杂程度。

2.2.2 UML

1995 年“统一方法”以 UML1.0 的正式名提交给对象管理组织进行审批，并最终于 1997 年首次推出。软件建模是软件工程中的一种普遍实践，UML 是此实践过程的标准符号[33]。UML 是一种用于领域建模和应用程序设计的标准建模语言[34]。它的语法是语义的一种可视化表示，用于对系统的静态结构和动态行为建模，因此 UML 语法可以理解为建模工具的使用指导方法。

UML 标准分为 Infrastructure[35]和 Superstructure[36]两个标准。UML Infrastructure 标准包含组成元模型的基本概念，UML Superstructure 标准中包含由基本概念组成的描述系统静态结构的用例图、类图元模型等和描述动态行为时序图、状态图元模型等。UML 语义是使用建模工具对客观世界抽象后得到的模型元素和元素间的关联的语义，即它们所代表的现实世界中的概念以及概念间的关系的含义。

2.3 类图元模型和状态图元模型

MDA 中的模型转换是基于语义的转换，转换前后语义不变。源模型到目标模型的转换首先需要根据元模型层面建立源元模型与目标元模型的映射规则，然后在模型层面应用转换规则实现源模型到目标模型的转换。下文类图和状态图元模型的语义进行介绍。

2.3.1 类图元模型

对 UML2.0 标准[35, 36]中给出的“Classes Diagram”、“Generalization Diagram”、“DataTypes Diagram”、“Types Diagram”等元模型融合和简化后得到如图 2.2 所示的类图元模型，它的主要元元素有表示类的“Class”、表示类与类之间关系的“Association”、表示属性的“Property”、表示数据类型的“DataType”和表示继承关系的“Generelization”等元元素。

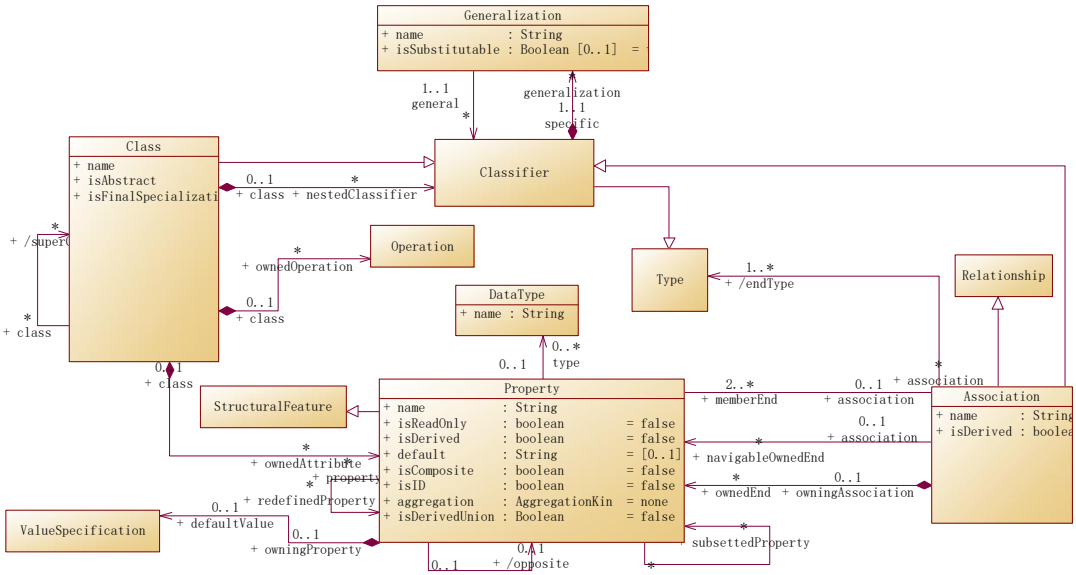


图 2.2 类图元模型

“Class”元元素表示一组拥有相同属性、方法的对象集合，它含有表示名称的“name”、表示是否是抽象的“isAbstract”等元属性。“Property”元元素表示一个类具有的特征，它含有表示名称的“name”、表示是否只读的“isReadOnly”、表示默认值的“default”等元属性。“Association”元元素表示类与类之间的关系或

结构，它含有表示名称的“name”和表示是否派生的“isDerived”元属性。“DataType”元元素表示数据类型或类与属性之间的关联关系，它含有表示名称的“name”元属性。“Generelization”元元素表示类与类间的泛化关系。它含有表示名称的“name”和表示是否使用特定分类器的“isSubstitutable”元属性。

2.3.2 状态图元模型

UML2.0 标准[36]中给出的“State Machines”元模型中主要元元素有表示初始和退出伪状态的“Pseudostate”、表示状态的“State”、表示状态转移的“Transition”等元元素。这些元元素组成了如图 2.3 所示的状态图的行为状态机元模型。

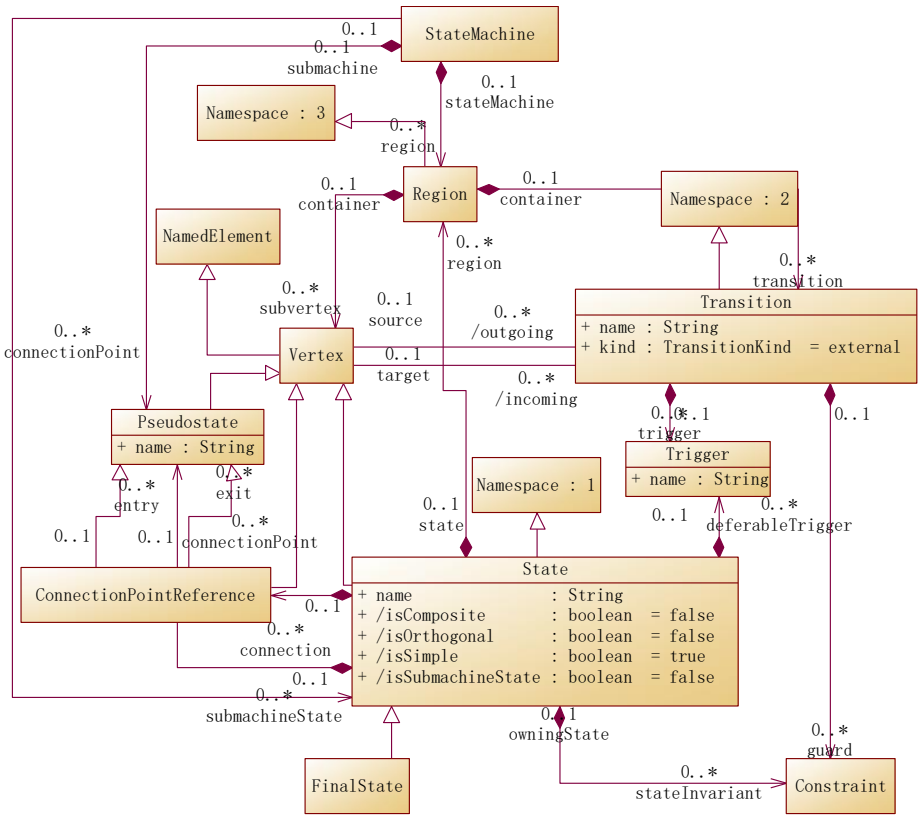


图 2.3 行为状态机元模型

“Pseudostate”元元素有多种类型，它可以表示初始伪状态(Initial Pseudostate)、进入点(Entry Point)、退出点(Exit Point)等，它含有表示名称的“name”元属性。“State”元元素表示在某种条件下对象所处的一种情况，它含有表示名称的“name”、表示是否为复合状态的“isComposite”、表示是否是正交复合状态的“isOrthogonal”、

表示是否是简单状态的“isSimple”等元属性。“Transition”表示状态之间的转移，它包含表示名称的“name”和表示转移的类型的“kind”元属性。“Trigger”元元素表示触发器。它含有表示名称的“name”元属性。

2.4 MDA 中的模型转换类型

模型转换是 MDA 的核心，它将源语言描述的模型中的概念映射到另一种目标语言的概念，同时将前者的语义转换为另一种[37]。MDA 标准 1.0 给出了“模型实例转换”和“应用设计模式的模型转换”两种模型转换思路。

1. 模型实例转换

“模型实例转换”通过标记将包含 PSM 信息的映射规则应用于 PIM，进而指导 PIM 的中的元素根据语义转换为 PSM 中的元素。标记值具有名称和类型，标记值是在 UML 元模型中附加到 UML 元元素的扩展属性[30]。模型实例转换的思路和过程如图 2.4 所示。

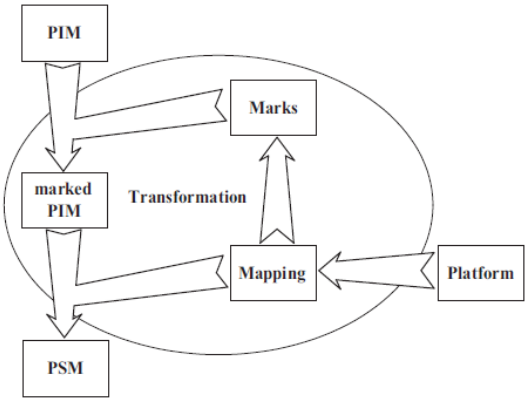


图 2.4 标记模型

首先抽象映射规则，开发人员在平台选定后就已经确定了平台包含的特性。根据平台包含的平台特性信息抽象出映射，然后获取标记并应用于 PIM，每一条映射规则抽象为用于标记 PIM 的标记的语义，如果一个标记拥有多个标记值，那么该标记对应多条转换规则，接着使用这些标记对 PIM 元模型的元元素进行标记，指导这些元元素的实例如何根据标记值包含的语义转换为具有平台特性的 PSM 元模型元元素的实例。最后转换带标记的 PIM 到 PSM，被标记后的 PIM 称为带标记的平台

无关模型(marked PIM)，带标记的平台无关模型根据标记值的语义转换为平台相关模型。

2. 应用设计模式的转换

“应用设计模式的模型转换”是“模型实例转换”思路的扩展。在“应用设计模式的模型转换”中，“模型实例转换”中的映射(Mapping)变为了表示模式信息的“Patterns”，表示标记的“Marks”变为了模式名称(Patterns Names)，即扩展后的转换过程中使用模式名称对 PIM 元模型的元元素进行标记。扩展后的转换思路和过程如图 2.5 所示

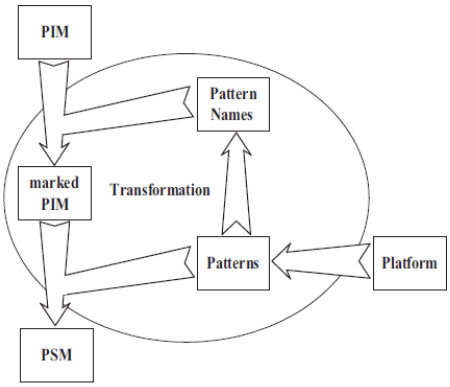


图 2.5 模式的应用

首先获取模式信息，根据实现平台应用的设计模式获取设计模式信息。然后获取设计模式中的名称形成标记，对设计模式具有的特定信息的语义进行抽象并为这些特性进行命名，形成 PIM 中元素的标记。接着使用特性的名称标记 PIM，使用名称标记 PIM 中的元素，每一个用于标记的模式特性的名称用于指导如何根据标记的语义将 PIM 中被标记的元素转换为 PSM 中的元素，从而使转换得到的 PSM 包含设计模式信息。最后转换标记的 PIM 到 PSM，标记的 PIM 根据标记的值拥有的语义转换到 PSM，最终实现应用设计模式的平台无关模型到平台相关模型的转换。

2.5 MVC 应用的设计模式

面向对象分析和设计中应该遵守单一职责原则、开放封闭原则、依赖倒置原则、接口隔离原则和里氏替换原则五个设计原则。设计模式是在解决实际问题中对设计原则的应用。有经验的面向对象开发者建立了既有通用原则又有惯用方案的指令系

统来指导他们完成软件设计。如果以结构化形式对这些问题、解决方案和命名进行描述使其系统化，那么这些原则和习惯用法称为模式[38]。设计模式越来越被人们接受，它不仅作为软件开发的可重用构造，而且也是软件系统体系结构设计的文档和理解结果[39]。

模型/视图/控制(Model/View/Control, MVC)在 1970 年被施乐帕罗奥多研究中心的教授 Trygve Reenskaug 提出[40]，它的目的是建立计算机中用户模型和数据模型之间的桥梁。它将应用程序划分为实体、控制和视图三个逻辑部件，所以可以说它是一个软件架构。Struts、JSF、WEBWORK 等开源项目则分别以自己的方式实现了这一架构，则称这些实现者为框架[41]。图 2.6 展示了 MVC 框架模式中三个组件之间的关系。模型是系统中管理所有事物的组件，它主要用于业务逻辑的处理和数据的存取。视图主要用于管理用户的图形化界面，用于向用户展示数据。控制器用于处理用户在视图中发起的事件。

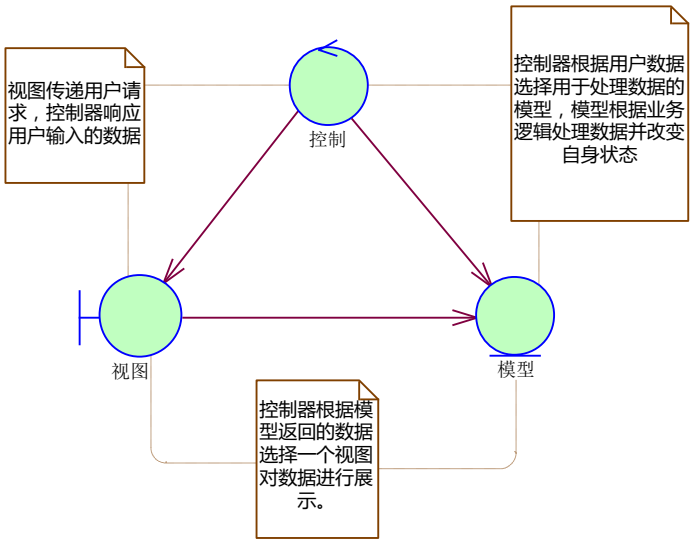


图 2.6 MVC 组件之间的关系

用户在视图输入数据，视图将用户请求和数据传递给控制器，控制器根据用户输入的数据选择对应的模型。模型根据业务逻辑对控制器传递的数据进行处理并更新数据库或从数据库获取新的数据。模型处理完毕后，将数据提交给控制器，控制器根据模型提交的数据选择合适的视图对数据进行展示。

虽然 MVC 是一种框架模式，而非设计模式。但 MVC 应用了观察者设计模式、策略设计模式和组合设计模式三种设计模式。

2.5.1 策略模式在 MVC 中的应用

策略(Strategy)设计模式也称为政策(Policy)设计模式，主要解决如何设计变换且相关的算法，使得算法可以独立于使用它的客户而变化。策略模式的结构如下图 2.7 所示。

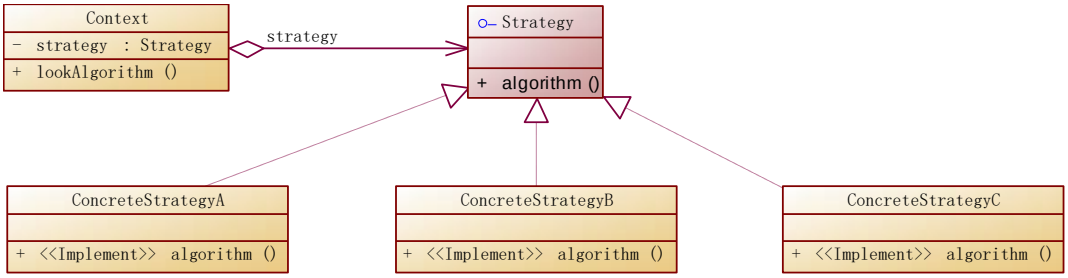


图 2.7 策略设计模式结构

Context 称为“上下文”，它是用于维护一个“Strategy”对象的引用的类。Context 中包含一个名为“strategy”的属性，数据类型是 Strategy 类型，用于存储对象的引用。Strategy 称为“策略”，它是一个用于定义所有算法的公共接口。Strategy 中包含名为“algorithm”的抽象方法。ConcreteStrategyA、ConcreteStrategyB 和 ConcreteStrategyC 称为“具体策略”，它们分别实现了 Strategy 接口的抽象方法。

策略设计模式和 MVC 的关系如图 2.8 所示。

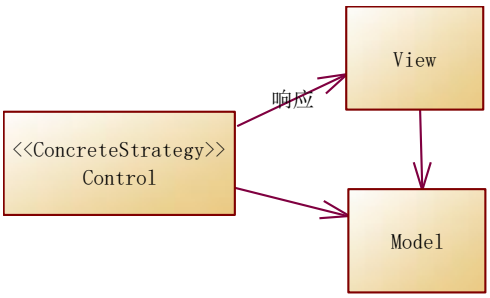


图 2.8 策略设计模式与 MVC 的关系

控制器扮演“具体策略”角色。视图将行为委托给控制器，控制器动态的改变行为。相同的数据可以有不同的处理方式，每种方式对应着一种控制器去调用模型，用以实现业务逻辑处理。

2.5.2 组合模式在 MVC 中的应用

组合(Composite)设计模式主要用于将对象组合成树型结构以表示“整体-部分”的层次关系。组合设计模式的结构如图 2.9 所示。

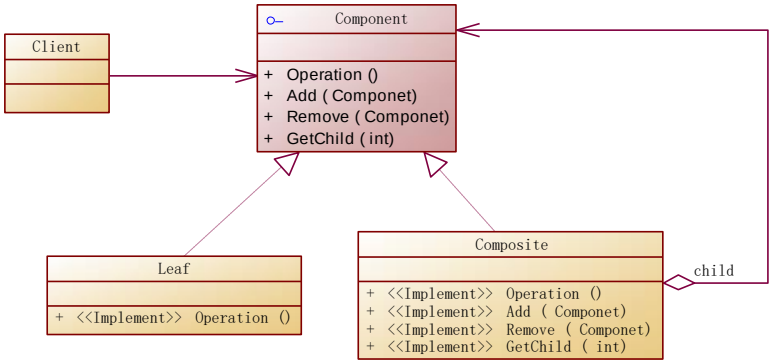


图 2.9 组合设计模式结构

Component 称为“组合部件”，它是一个包含了操作“Operation”、增加子节点“Add”、删除子节点“Remove”和获取子节点“GetChild”四个抽象方法的接口。Leaf 称为“叶子”，是一个实现“Component”接口的类，主要目的是在组合中表示子结点对象且规定叶子结点不能有子结点。Leaf 中包含了接口方法“Operation”的实现。Composite 称为“合成部件”，实现了 Component 接口中的“Operation”、“Add”、“Remove”和“GetChild”接口方法，主要用于对有子结点的行为进行定义并对部件信息进行存储。

组合设计模式与 MVC 的对应关系如图 2.10 所示。

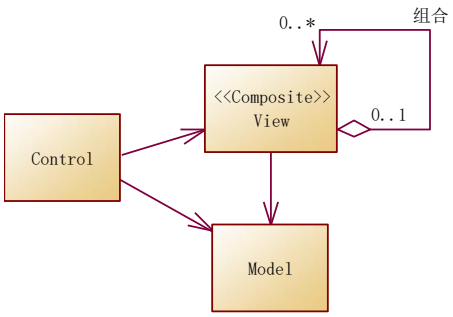


图 2.10 策略设计模式与 MVC 的关系

视图扮演“合成部件”的角色，允许视图嵌套。也就是说，一个视图可以由多个其他的视图组合而成，这涉及到“部分视图”的使用。视图中的窗口、面板、按钮、标签等，有的是组合结点，有的是叶子结点，利用组合模式可以让这些节点采取统一的处理方式。

2.5.3 观察者模式在 MVC 中的应用

观察者(Observer)设计模式也叫发布-订阅模式，不同类型的订阅者对象关注于发布者对象的状态变化或事件，并且想要在发布者产生事件时以自己独立的方式作出回应[38]。观察者设计模式的结构如图 2.11 所示。

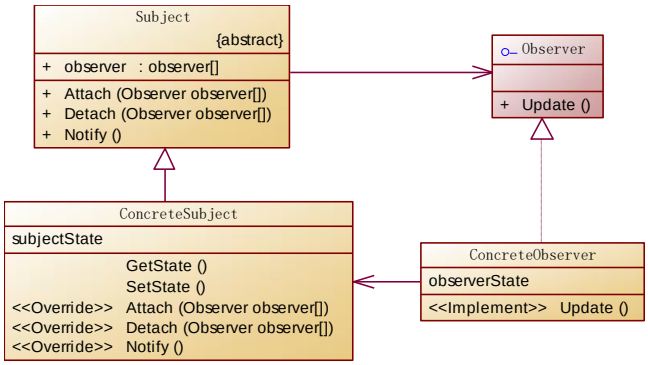


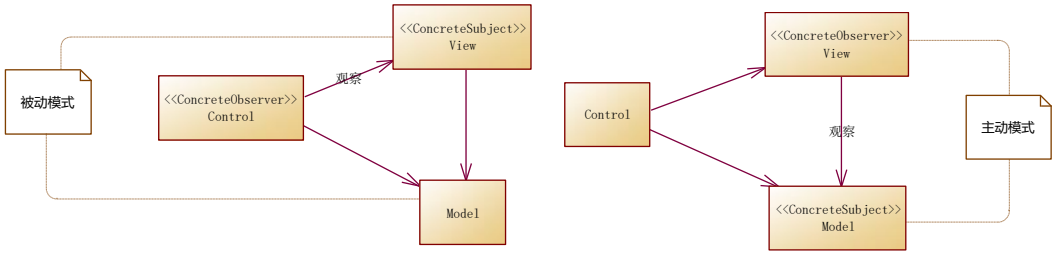
图 2.11 观察者设计模式

Subject 称为“目标”，它是一个包含用于为“具体观察者”绑定“具体目标”的“Attach”、用于从“具体目标”解绑“具体观察者”的“Detach”、用于在“具体目标”状态发生改变时通知观察者改变状态的“Notify”三个抽象方法的抽象类。

ConcreteSubject 称为“具体目标”，它重载了“Subject”抽象类中的三个抽象方法并且包含用于获得自身状态的“GetState”和用于设置自身状态的“SetState”方法。

Observer 称为“观察者”，它是一个包含“Update”抽象方法的接口。ConcreteObserver 称为“具体观察者”，它重写了 Observer 接口中的“Update”抽象方法，该方法用于在接收到“具体目标”的状态改变通知时，更新自己的状态。

MVC 有被动和主动模式两种模式。被动模式中，控制协调模型与视图的交互。主动模式中，模型直接与视图交互。两种情况与 MVC 的对应关系如图 2.12 所示。



(a) 观察者被动模式与 MVC 的关系 (b) 观察者主动模式与 MVC 的关系

图 2.12 观察者模式与 MVC 的关系

被动模式中，视图扮演“具体目标”的角色，控制扮演了“具体观察者”的角色。将控制与视图进行绑定，控制告知视图有哪些控制在观察它的变化，当某个事件触发后改变了视图的状态，视图遍历所有已经注册的控制，并将更改通知这些控制，控制接收到通知后进行更新。

主动模式中，模型扮演了“具体目标”的角色，视图扮演了“具体观察者”的角色。视图与模型进行绑定，视图告知模型有哪些视图在观察它的变化。当某个事件触发改变了模型的状态，模型将遍历所有已经注册的视图，并将更改通知它们，视图接收到通知后进行更新。

2.6 QVT

MDA 模型转换涉及将同一种语言、不同语言或不同抽象层次的模型转换为另一种模型。在 MDA 中，QVT 是指定上下文中模型转换的 OMG 标准语言，QVT 标准[42]由关系、核心和操作映射三种语言组成。

在操作语言中，“transformation”关键字用于定义一个模型转换函数并通过“in”和“out”两个形式参数指定模型转换的方向。转换函数包含“main”关键字表示的程序入口，表示转换算法从 main 开始执行。

“mapping”操作定义一个或多个源模型的元素到一个或多个目标模型元素之间的映射关系。映射操作在语法上由签名、作为前置条件的 when 子句、主体和作为后置条件的 where 子句组成。

通过“mapping”操作定义映射规则后，在 main 函数体中使用“map”关键字完成在算法流程中调用映射规则的过程。

2.7 PD 实现平台提供的相关技术

PowerDesigner 实现平台基于面向对象和模型驱动思想，是 Sybase 的企业建模和设计解决方案。该平台使得建模人员可以通过可视化的方式完成系统模型的创建以及目标代码的生成。因此本文选用 PowerDesigner 实现平台，结合本文提出的 Web 代码生成方法，实现 Web 表示层代码的生成。本节对 PowerDesigner 工具在代码生成过程中提供的资源文件编辑器、VBScript 和 GTL 三个重要技术进行介绍。

2.7.1 资源文件编辑器

PowerDesigner 提供的如图 2.13 所示的资源文件编辑器可以对目标语言资源文件(Resource File)和扩展文件(Extension File)进行浏览和编，以实现对元模型进行扩展。

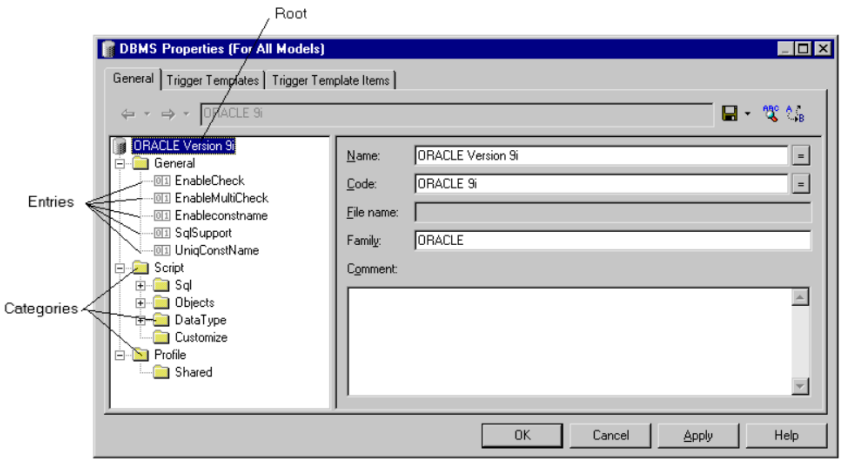


图 2.13 资源文件编辑器

左边部分显示资源文件中包含条目的树状目录，右边部分显示当前选定元素的属性。左侧树状目录中最主要的两个文件夹为“Genaeration”和“Profile”。“Genaeration”文件夹用于定义和激活生成过程。它包含类别和条目(Categories and Entries)。类别中包含“Command”和“Options”两个文件夹。“Command”包含用于在代码生成过程结尾调用的生成命令。“Options”包含在代码生成对话框中提供给代码生成人员选择的选项，选项的值可以通过 GTL 模板进行获取。Profile 文件夹用于实现 UML Profile 扩展机制。它包含代表元模型中元元素的扩展单元，每个扩展单元中可以包

含“Generated Files”、“Templates”、“Stereotypes”、“Extended attributes”、“Methods”等文件夹，其中最重要的是“Template”文件夹，它用于创建针对该元元素生成代码时所使用到的模板。通过这些文件夹，编码人员可以添加新的元元素、通过构造型和标准细分元元素类型、自定义元元素的符号、自定义 PowerDesigner 菜单和窗体以及修改代码生成过程的输出。

2.7.2 VBScript 和 GTL

模型中的元素是元模型中元元素的实例，模型中的信息存储在模型的元素中，PowerDesigner 通过使用 VBScript 或 GTL 访问模型中的元素或属性获取模型中的信息。

VBScript 是微软提供的一种脚本语言。PowerDesigner 提供了对该语言的支持，因此建模人员可以通过在 PowerDesigner 中编写和运行 VBScript 实现模型元素之间的转换，包括完成对模型元素信息的读取、模型元素的创建、模型元素的更新、模型元素的删除。

模板是实现模型到文本转换中最常用的方法[43]。GTL 由变量、操作符和宏(GTL Macro)组成，它是一种基于模板的语言，也是一种面向对象的语言，主要用于在模型到代码转化时通过使用“%元元素名称.元元素的元属性名称%”或“%元元素的元属性名称%”来访问元模型或元模型的元属性，从而获取模型的元素中存储的信息，并且通过使用“%模板名%”实现模板间的调用，从而将多个模板组合后形成用于生成目标代码的模板集。

2.8 本章小节

本章首先介绍了 MDA 的基础知识以及 MOF 和 UML 标准，其次介绍了 MDA 标准中的“模型类型”和“加入设计模式的模型转换”两种模型转换类型，然后介绍了表示层 MVC 框架与策略、组合和观察者设计模式的对应关系，最后介绍了 QVT 语言的组成部件和架构以及 PowerDesigner 实现平台提供的 VBScript 和 GTL 技术。

第 3 章 Web 代码生成方法研究

本章首先介绍了基于 MDA 中模型的表示和模型转换思想给出的 Web 代码生成整体思路，其次讨论了建模过程中继承和实例化在实现对象重用上的区别，然后讨论了在模型转换中加入设计模式的方法，最后介绍了代码生成过程中对象模型到状态模型、状态模型到表示层框架代码的转换规则和转换过程描述。

3.1 Web 代码生成方法整体思路

系统架构的好坏是决定一个系统是否稳定和是否易维护的重要因素。微软推荐的分层方式采用三层架构的思想，将系统的架构分为表示层、业务逻辑层和数据访问层。

表示层主要为用户提供用户交互的界面、显示用户感兴趣的数据和完成用户与界面的交互过程。业务逻辑层主要根据业务规则处理用户在表示层中提交的数据和搭建表示层和数据访问层的沟通桥梁。数据访问层主要完成将业务逻辑层的数据持久化到数据库中和将数据库中的数据获取后返回给业务逻辑层的工作。

因此，一个基于三层架构思想设计的系统包含如图 3.1 所示的表示层代码、业务逻辑层代码和数据访问层代码。

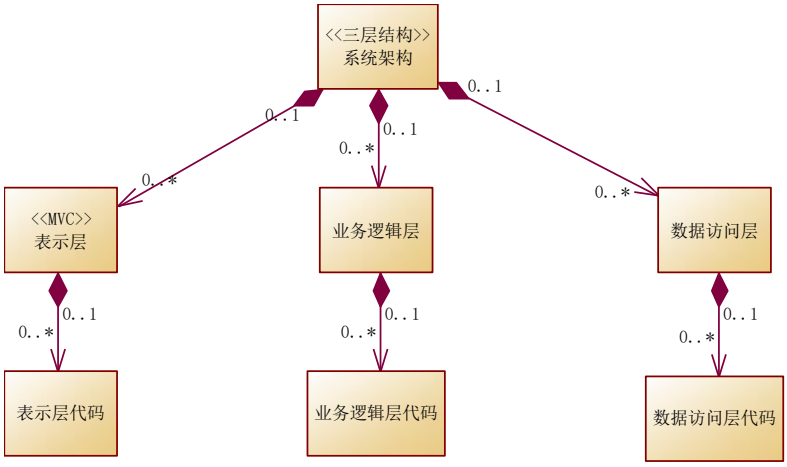


图 3.1 三层架构和代码的对应关系

本文基于模型驱动的 Web 代码生成方法主要研究应用了 MVC 框架模式的表示层代码的自动生成，而业务逻辑层和数据访问层的代码生成不做重点讨论。

基于 MDA 中模型表示和模型转换的思想，本文提出的 Web 代码生成方法整体思路如图 3.2 所示。首先使用类图对客观世界进行抽象得到类图表示的领域模型，然后将 Java 语言平台信息添加到领域模型中，得到类图表示的对象模型。接着在对象模型到状态模型的转换过程中加入 JSF 表示层框架和设计模式信息，并根据类图到状态图的转换规则将对象模型转换为状态模型。最后根据状态图到 JSF 框架代码的转换规则，使用生成模板语言(Generation Template Language, GTL)表示出 Web 表示层框架代码的结构，将状态模型生成 JSF 表示层框架代码。

类图没有操作细节，而这个本来是UI页面，所以才这么做的吧

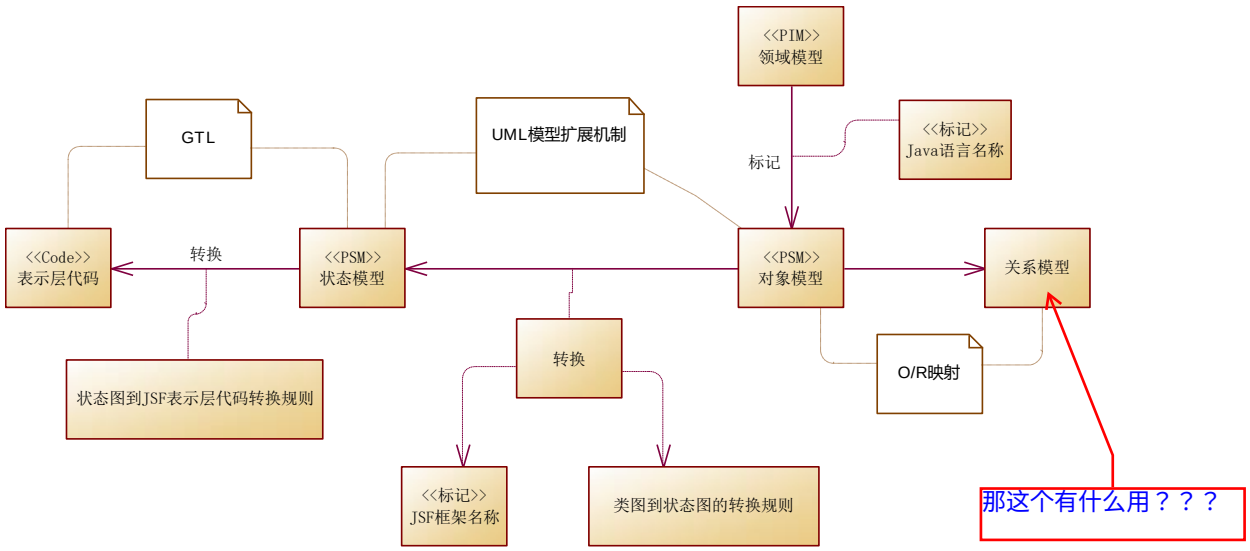


图 3.2 Web 代码生成整体思路

Web 代码生成整体思路包含“标记领域模型得到对象模型”、“转换对象模型到关系模型”、“转换对象模型到状态模型”、“转换状态模型到表示层代码”四个过程。

1. 标记领域模型得到对象模型

领域模型是对客观世界中真实存在的对象以及对象之间的关系的抽象。对象建模的主要概念是标识系统环境中的对象以及对象之间的关系和交互[44]。本文通过类图表示领域模型，类图中的类代表具有属性和操作的域对象[45]。首先在业务分析和系统分析阶段使用类图创建与实现平台无关的领域模型，完成对客观世界的抽象。

然后将 Java 编程语言平台的数据类型等信息添加到类图元模型中。最后类图表示的领域模型自动转换为类图表示的具有 Java 编程语言平台信息的对象模型。

2. 转换对象模型到关系模型

首先建立对象模型与关系模型的 O/R 映射，然后根据映射关系将对象模型转换为关系模型。得到关系模型后，可以在 PowerDesigner 中通过关系模型生成 SQL 语言文件。然后在数据库管理工具中直接应用 SQL 语言文件完成数据库表的创建。因为该过程涉及的知识范围为三层架构中数据访问层代码的生成，所以本文不作重点讨论。

3. 转换对象模型到状态模型

首先将 JSF 表示层框架信息和设计模式信息扩展到状态图元模型中。然后为了使得状态模型中的状态元素能够表示一个页面，通过 UML 扩展机制将页面语义扩展到状态图元模型中。最后在应用类图到状态图的转换规则实现模型转换的过程中使用 JSF 框架名称标记对象模型，使得对象模型转换到具有表示层框架信息和设计模式信息的状态模型。

4. 转换状态模型到表示层代码

根据状态图到 JSF 表示层框架代码的转换规则，将具有 JSF 表示层框架信息和设计模式信息的状态模型生成 Web 表示层框架代码。

3.2 建模方法

领域模型、对象模型和状态模型的建模过程对应了本文 Web 代码生成方法整体思路中创建领域模型、领域模型到对象模型和对象模型到状态模型三个过程。

为了将对客观世界抽象后得到的平台无关的领域模型转换到具有 Java 语言平台信息的对象模型，采用继承的方式将 Java 语言数据类型等信息扩展到类图元模型，使得实例化类图元模型得到的对象模型具有 Java 语言平台的信息。为了将对象模型转换到具有 JSF 表示层框架信息、设计模式信息和页面语义的状态模型，采用继承的方式将 JSF 表示层框架信息、设计模式信息和页面语义扩展到状态图元模型，使得实例化状态图元模型得到的状态模型具有 JSF 表示层框架信息、设计模式信息和页面语义。为了将对象模型生成状态模型，基于类图元模型和状态图元模型定义转换规则，转换过程中应用转换规则实现对象模型到状态模型的转换。

这个就是和上一篇Solidity的思路一样，继承来拓展元模型

因此，本文 Web 代码生成方法整体思路的建模过程中需要解决两个关键问题。第一个关键问题是，既然继承和实例化在建模过程中都可以使用，那么什么情况下使用继承方式，什么情况下使用实例化方式。第二个关键问题是，目标代码平台应用了设计模式，那么如何在模型转换中加入设计模式信息，使得通过模型转换得到的目标代码具有设计模式信息。

3.2.1 继承和实例化两种模型表示的方法

基于 MOF 元建模四层结构的思想，M0 信息层是对客观世界抽象后的结果，M1 模型层是对 M0 信息层的抽象，M2 元模型层是对 M1 模型层的抽象，M3 元元模型层是对 M2 元模型层的抽象。

在客观世界的建模场景中，人们通过上述抽象过程的逆过程完成模型的表示。人们通过实例化 M3 层的元元模型表示 M2 层的元模型，通过实例化 M2 层的元元模型表示 M1 层的模型，对 M1 层的模型实例化，用于表示被抽象后的客观世界。

但在模型的表示过程中可能存在使用元元模型无法表示出元模型或元模型无法表示出模型的问题，导致这种情况的根本原因是元元模型或元模型缺少描当前元模型或模型的语义。因此人们通常采用继承的方式在 M3 层对元元模型或在 M2 层对元模型的语义进行扩展，然后通过实例化扩展后的元元模型或扩展后的元模型的方式完成模型的表示过程。图 3.3 给出了客观世界中人们完成模型的表示和扩展的过程。

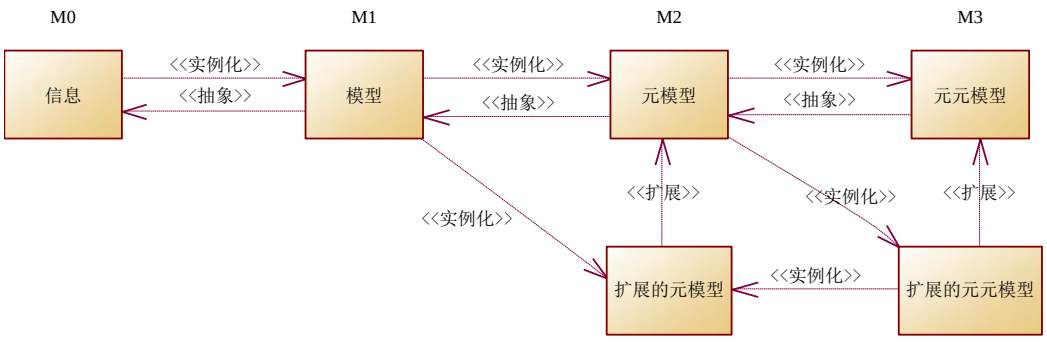


图 3.3 模型的表示和扩展

扩展 M3 层的元元模型后得到扩展的元元模型，实例化扩展的元元模型实际上实例化了扩展前的元元模型。同理，扩展 M2 层的元模型后得到扩展的元模型，实例化扩展的元模型实际上实例化了扩展前的元模型。

上述对客观世界的建模场景中，继承和实例化都可以应用于建模过程。那么在实际的建模过程中，什么情况下使用继承方式和什么情况下使用实例化方式是本文的 Web 代码生成方法中需要解决的一个关键问题。为了解决该问题，下文对 UML 建模工具设计人员和建模人员在建模过程中，使用继承和实例化方式完成工作的场景进行分析。

在模型驱动工程领域，模型转换被认为是管理模型的一种技术。为了提高开发生产力以及模型转换的质量，重用机制是必不可少的，而面向对象思想最根本的内容是实现重用，因此模型驱动架构中应用了面向对象思想。尽管研究者已经认识到重用性的需求，但由于大量的重用机制都显示出来，今天大多数转换设计者仍然按照特定的方式来指导模型的转换。这是因为不同的重用机制的应用场景没有明确的定义[46]。

Web 应用程序开发的实质是将客观世界变为程序代码，即将客观世界中的事物转变为代码中的对象。面向对象思想将客观世界中的事物看作一个个对象。客观世界千差万别且非常复杂，如果人们对每一个对象进行认识将会非常困难。对于编码人员来说，如果对每个对象都进行编码，那工程量会相当庞大。

基于元建模四层架构的思想，M0 信息层是对客观世界的抽象。在 M0 层根据人们认识事物的基本思维方式，采用分类的方式将客观世界中当前关心的范围（即，论域）中具有相同特性的对象分为一类。每一类形成一个集合，集合中的元素就是客观世界中属于当前分类结果的对象。如果分类后形成的集合中的对象仍然难以理解，人们继续将当前集合中的对象按照上述分类方式，将具有相同特性的对象分为一类并形成新的集合。理论上来说，可以对对象进行任意多次的分类，但最终的分类粒度根据实际应用场景决定。

图 3.4 展示了人们采用分类的方式认识客观世界中对象的过程。Δ 表示当前关注的背景，即论域。O_1、O_2 表示客观世界中真实存在的学生对象。O_3、O_4 表示客观世界中真实存在的老师对象。O_5、O_6 表示当前论域中的其他对象。

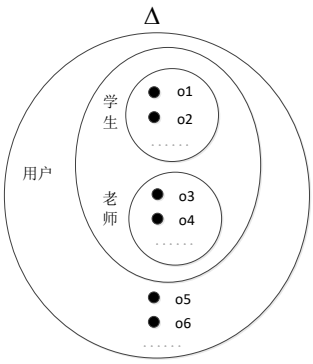


图 3.4 分类客观世界的事物

上述应用场景中当前论域为系统的用户，因此将具有系统用户特征的对象和不具有用户特征的对象分类，并将具有系统用户特征的对象形成的集合使用符号“用户”命名，表示该集合中的所有对象都是系统用户。为了区分系统用户，人们继续将“用户”集合中的对象进行分类，将具有学生特征的对象分为一类并使用符号“学生”命名形成的集合，将具有老师特征的对象分为一类并使用符号“老师”命名形成的集合。上述分类过程完成后，O_1、O_2、对象都具有学生的特性，它们是“学生”集合中的元素。O_3、O_4 对象具有老师的特性，它们是“老师”集合中的元素。

UML 建模人员根据认识客观世界的思维方式对客观世界中的对象进行分类形成 M0 信息层的数学模型，接着将数学模型中的一个个集合抽象为类图中的一个个类、关联和属性等元素。图 3.5 展现了建模人员对客观世界中学生练习场景的抽象过程。

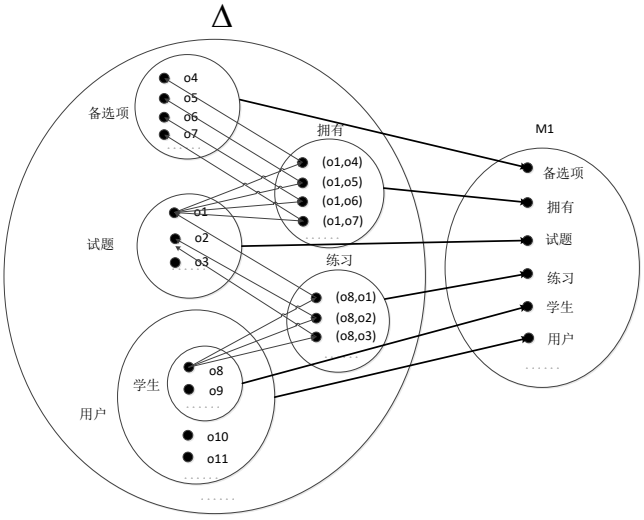


图 3.5 M0 到 M1 的抽象过程

建模人员将 M0 信息层中“学生”、“用户”、“试题”、“备选项”、“学生与试题的练习关系”、“试题与备选项的拥有关系”等集合抽象为 M1 模型层的类图中的类元素和关联元素，并分别使用符号“学生”、“用户”、“试题”、“备选项”、“练习”、“拥有”命名。

对于建模人员来说，在 M1 层创建类图时，可根据 M0 层中集合的包含关系，采用继承方式将类图中某些元素的共同特性进行抽象，形成该元素的一个父元素，或将一个父元素的特性具体化，形成该元素的多个子类元素。继承方式使得建模人员建模子类元素时重用了父类元素的属性和操作。如果在类图中使用了继承关联关系，编码人员在根据模型描述的语义将子类元素实例化形成的代码时，实现了对其父类元素形成的代码的重用。图 3.6 给出了建模人员对 M0 信息层的学生练习场景中的集合抽象后形成的 M1 模型层中的部分类图。

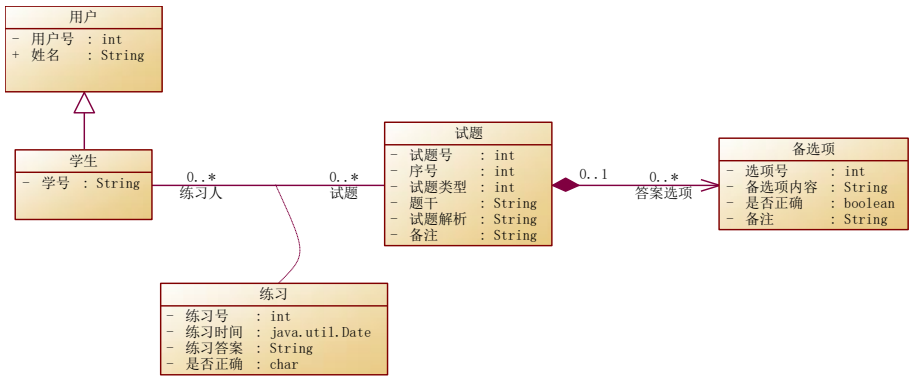


图 3.6 学生练习场景部分类图

学生具有用户的特性，该关系使用继承关联表示，继承使得学生重用了用户的“姓名”属性。学生和试题存在多对多的练习关系，该关系通过“多对多”关联和“练习”关联类表示。试题拥有零个或多个备选项，该关系通过组合关联表示，试题一旦消失，备选项也就跟着消失。

对于编程人员来说，在编码过程中需将类图中的类元素所代表的对象进行编码，形成能够在程序中运行的一个个对象，并最终形成一条一条的数据存入数据库。因此，从概念上来说，编码人员在编码时实际上完成了将类图中的类实例化为程序中的对象的过程，即编码人员将类图中的类元素变为程序中的类语句，通过实例化语句创建程序中的对象来代替为每一个对象进行编码，进而实现了对象的重用。

M2 元模型层是对 M1 模型层的抽象。对类图来说，元模型层中的类图元模型是建模人员创建类图的建模工具。建模人员使用类图描述 M0 层抽象的客观世界时，实际上是对 M2 层类图元模型中的“Class”、“Property”、“Generelization”、“Association”、“DataType”等元元素进行实例化，这些元元素实例化的结果构成的集合即 M1 层的类图。图 3.7 展示了建模人员通过实例化建立类图的过程。

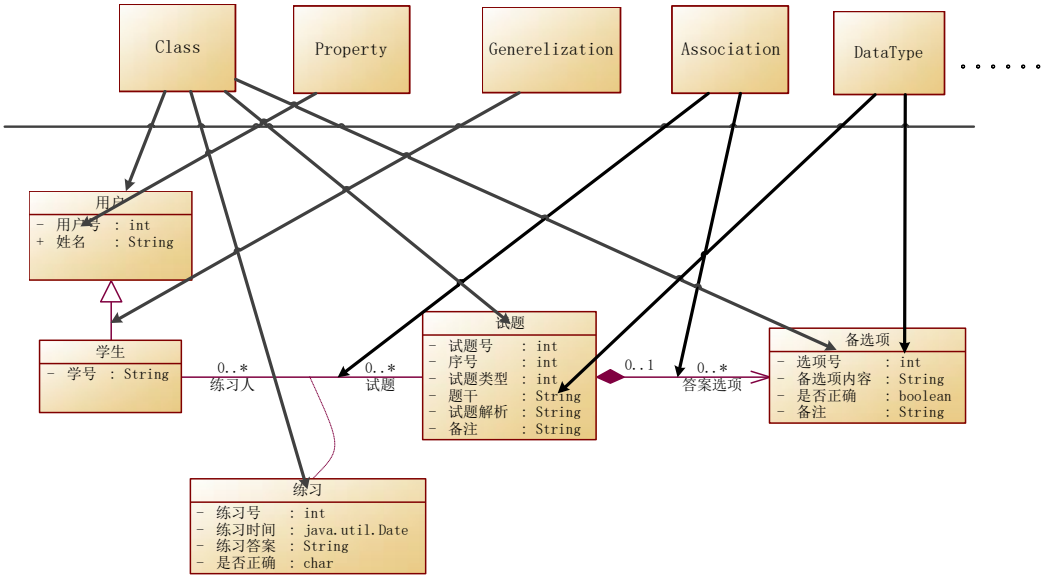


图 3.7 实例化创建类图过程

建模人员实例化 M2 层类图元模型中的“Class”元元素得到 M1 层类图中的“用户”、“学生”、“练习”、“试题”、“备选项”等类。实例化类图元模型中的“Property”元元素得到 M1 层类图中“用户号”、“姓名”、“练习时间”、“是否正确”等属性。实例化类图元模型中的“Generelization”元元素得到 M1 层类图中“用户”类与“学生”类的继承关联关系。实例化类图元模型中的“Association”元元素得到 M1 层类图中“学生”类与“试题”类的多对多关联关系以及“试题”类与“备选项”类的组合关联关系。实例化类图元模型中的“DataType”元元素得到 M1 层类图中“int”、“String”、“boolean”等数据类型。

类似 M1 模型到 M2 元模型的分类和抽象过程，还可将 M2 层中具有相同特性的元元素进行分类，然后抽象得到 M3 层的元元元素。通过这种方式，UML 建模工具设计人员将元模型中的元元素进行分类，得到表 3.1 所示的 M3 元元模型层中的八种元元元素。

表 3.1 元元模型中的元元元素

元元元素名称	作用
Class	描述类
Association	描述类之间关系
Aggregation	描述特殊关联关系行为
DataType	描述数据类型
Package	描述存放分类后模型的空间
Property	描述元元素属性或描述类与数据类型间关系
Operations	描述操作
References	描述引用关系

在 M3 元元模型层中，使用“Class”描述类，使用“Association”描述类之间关系，使用“Aggregation”描述特殊关联关系行为，使用“DataType”描述数据类型的，使用“Package”描述存放分类后模型的空间，使用“Property”描述元元素属性或描述类与数据类型间关系，使用“Operations”描述操作，使用“References”描述引用关系。

对于建模人员来说，元模型不仅是一种建模工具，还是一种建模方法。建模人员根据建模方法完成模型的创建。但对建模人员来说，他们只是建模工具的使用者，他们依赖于建模工具设计人员为其提供建模工具，即提供建模方法。

UML 建模工具设计人员通过实例化 M3 层元元模型中的八种元元元素，构建了属于 M2 层的 UML 中类图、用例图、时序图和状态图等建模工具。对于大多数场景来说，UML 建模人员使用类图、状态图等 UML 建模工具就可以完成自己的建模工作。

但对于特定领域建模，UML 建模能力还不够，需要对其进行相应的扩展[47]。UML 标准提供两种扩展元模型的机制。

第一种扩展机制不会改变元模型的结构。UML Profile 机制用于实现这种扩展，它是一组预定义的构造型、标记值和约束形成的集合，用于支持特定环境的建模。因为通过 Profile 扩展后不形成新的元元素，所以扩展后的元元素的表现力受到约束的限制。建模工具设计人员完成这种类型的扩展时，直接在 M2 层的元元模型中通过继承实现。通过继承的方式，建模工具设计人员重用了原有建模工具的描述能力。

第二种扩展机制会在 M2 层元模型中增加新的元元素，从而导致元模型结构的改变。建模工具设计人员通过实例化 M3 层元元模型中元元元素，在 M2 层元模型

中增加新的元元素。这种扩展机制跨越了元层次的结构，所以这种扩展机制以 MOF 元建模框架为指导。通过实例化的方式，建模工具设计人员实现了创建建模工具的建模单元的重用。

根据上述建模人员和建模工具设计人员在模型的表示过程中的分工场景可知，采用继承的方式时，重用者扮演软件建模工具设计人员的角色，他们在 M2 层通过继承的方式将具有新语义的元元素继承原有的元元素，实现对元模型语义的扩展以及被扩展元元素语义的重用。

采用实例化的方式时，重用者扮演软件建模人员的角色。他们通过实例化 M2 层元模型建立 M1 层的模型，实现使用模型描述客观世界中的对象、对象的特性和对象与对象之间的关系等。因为实例化过程可以使用相同的建模工具描述不同的应用场景，所以实例化实现了建模工具的重用。

对于特殊情况，建模工具设计人员有时首先在 M3 层采用继承的方式扩展元元模型，然后实例化元元模型得到 M2 层的元模型，进而建立建模工具。这种方式中，建模工具被看作一个模型，建模工具设计人员扮演了建模人员的角色，创建建模工具的过程实际上完成了建模人员通过实例化建立模型的过程。

综上所述，当在基于 MDA 的软件开发过程中完成模型的表示过程时，如果要对建模工具进行设计，则开发人员扮演建模工具设计人员角色，使用继承或继承结合实例化的方式实现重用；如果要对某个特定场景进行建模，则开发人员扮演建模人员的角色，使用实例化的方式实现重用。

上述讨论解决了在模型表示过程中继承和实例化两种方式的选择问题，从而解决了本文的 Web 代码生成方法整体思路中如何对模型进行表示的问题。

在基于本文的 Web 代码生成方法中，为了使得对象模型生成的状态模型中每个状态可以表示一个页面并且能够实现对新增、删除、修改和查询页面的区分，使用该代码生成方法的开发人员扮演建模工具设计者的角色，采用继承的方式将页面语义信息扩展到状态图元模型中。

建模工具设计人员通过继承的方式将“Page”、“CreatePage”、“EditPage”、“FindPage”、“ListPage”等属性添加到状态图元模型的“State”元元素中，图 3.8 给出了扩展后的结果。

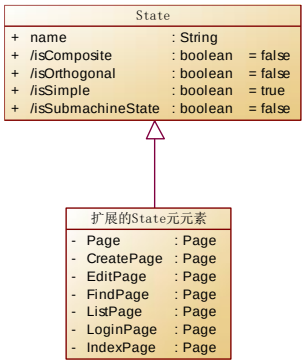


图 3.8 扩展 State 元元素

扩展后的 State 元元素增加了表示界面的“Page”属性，数据类型为“Page”，增加了表示新增界面的“CreatePage”属性，数据类型为“Page”，增加了表示修改界面的“EditPage”属性，数据类型为“Page”，增加了表示查找界面的“FindPage”属性，数据类型为“Page”，增加了表示列表和删除界面的“ListPage”属性，数据类型为“Page”，增加了表示登录界面的“LoginPage”属性，数据类型为“Page”，增加了表示主界面的“IndexPage”属性，数据类型为“Page”。

3.2.2 模型转换中加入表示层设计模式的方法

根据本文基于 MDA 的 Web 代码生成方法整体思路，最终的结果将生成应用 MVC 框架模式的 JSF 表示层框架代码。MVC 框架模式包含策略设计模式、组合设计模式和观察者设计模式，因此在生成 JSF 表示层代码的模型转换过程中需要将策略设计模式、组合设计模式和观察者设计模式的信息添加到转换过程中。那么在实际的建模过程中，如何在模型转换中加入设计模式信息是本文的 Web 代码生成方法中需要解决的另一个关键问题。为了解决该问题，下文针对观察者设计模式讨论，对模型转换过程中使用继承和实例化方式将观察者设计模式信息扩展到类图元模型中的方法进行讨论。

1. 采用继承方式

建模工具设计人员采用继承的方式在类图元模型的“Class”、“Property”、“Operation”、“Interface”元元素中添加拥有“具体目标”、“具体观察者”等语义的属性，从而将观察者设计模式信息扩展到类图元模型中。

(1) 扩展 Class 元元素

为使得建模人员实例化 M2 层的“Class”元元素后得到的 M1 层类元素具有观察者设计模式中“目标”、“具体目标”或“具体观察者”的语义，所以建模工具设计人员通过继承在“Class”元元素中增加“isSubject”、“isConcreteSubject”和“isConcreteObserver”三个元属性，它们的数据类型都是 Boolean。图 3.9 给出在“Class”元元素加入语义后的结果。

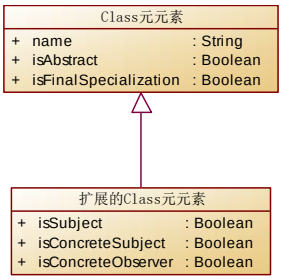


图 3.9 扩展 Class 元元素

当“isSubject”属性值为 True，“isConcreteSubject”和“isConcreteObserver”属性值为 False 时，表示当前“Class”元元素的实例代表一个“目标”。当“isConcreteSubject”属性值为 True，“isSubject”和“isConcreteObserver”属性值为 False 时，表示当前“Class”元元素的实例代表一个“具体目标”。当“isConcreteObserver”属性值为 True，“isSubject”和“isConcreteSubject”属性值为 False 时，表示当前“Class”元元素的实例代表一个“具体观察者”。最后规定三个属性的默认值为 False，三个属性中任意有一个值为 True 时，其余两个属性值必须为 False。

三个扩展属性的语义如表 3.2 所示。

表 3.2 Class 元元素的扩展属性

扩展属性名称	类型	语义	缺省值
isSubject	Boolean	当前元元素是否表示“目标”参与者	False
isConcreteSubject	Boolean	当前元元素是否表示“具体目标”参与者	False
isConcreteObserver	Boolean	当前元元素是否表示“具体观察者”参与者	False

(2) 扩展 Property 元元素

为使得建模人员实例化 M2 层的“Proprty”元元素后得到的 M1 层的属性元素具有观察者设计模式中“观察者集合”、“具体目标状态”或“具体观察者状态”的语义，所以建模工具设计人员通过继承在“Property”元元素中增加“isObserver”、“isSubjectState”、和“isObserverState”三个元属性，它们的数据类型都是 Boolean。图 3.10 给出在“Proprty”中加入语义后的结果。

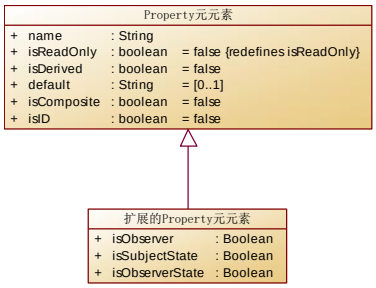


图 3.10 扩展 Class 元元素

当“isObserver”属性值为 True，“isSubjectState”和“isObserverState”属性值为 False 时，表示当前“Property”元元素的实例代表一个“观察者集合”。当“isSubjectState”属性值为 True，“isObserver”和“isConcreteObserver”属性值为 False 时，表示当前“Property”元元素的实例代表一个“具体目标状态”。当“isObserverState”属性值为 True，“isObserver”和“isSubjectState”属性值为 False 时，表示当前“Property”元元素的实例代表一个“具体观察者状态”。

最后规定三个属性的默认值为 False，并且三个属性中任意有一个值为 True 时，其余两个属性值必须为 False。

三个扩展属性的语义如表 3.3 所示。

表 3.3 Property 元元素的扩展属性

扩展属性名称	类型	语义	缺省值
isObserver	Boolean	当前元元素是否表示“观察者集合”属性	False
isSubjectState	Boolean	当前元元素是否表示“具体目标状态”属性	False
isObserverState	Boolean	当前元元素是否表示“具体观察者状态”属性	False

(3) 扩展 Operation 元元素

为使得建模人员实例化 M2 层的“Operation”元元素后得到的 M1 层的操作元素具有观察者设计模式中“绑定观察者”、“解绑观察者”、“通知观察者进行状态改变”、“获得目标自身状态”、“设置目标自身状态”和“更新自身状态”的语义，所以建模工具设计人员通过继承在“Operation”元元素中增加“isAttach”、“isDetach”、“isNotify”、“isGetState”、“isSetState”和“isUpdate”六个元属性，它们的数据类型都是 Boolean。图 3.11 给出在“Operation”元元素中加入语义后的结果。

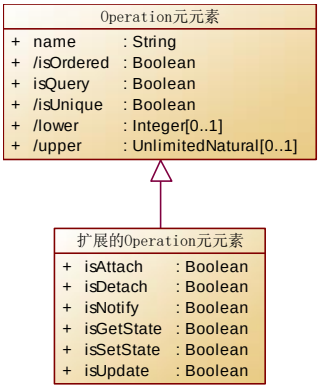


图 3.11 扩展 Operation 元元素

当“isAttach”属性值为 True，“isDetach”、“isNotify”、“isGetState”、“isSetState”和“isUpdate”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“绑定观察者”操作。当“isDetach”属性值为 True，“isAttach”、“isNotify”、“isGetState”、“isSetState”和“isUpdate”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“解绑观察者”操作。当“isNotify”属性值为 True，“isAttach”、“isDetach”、“isGetState”、“isSetState”和“isUpdate”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“通知观察者进行状态改变”操作。当“isGetState”属性值为 True，“isAttach”、“isDetach”、“isNotify”、“isSetState”和“isUpdate”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“获得目标自身状态”操作。当“isSetState”属性值为 True，“isAttach”、“isDetach”、“isNotify”、“isGetState”和“isUpdate”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“设置目标自身状态”操作。当“isUpdate”属性值为 True，“isAttach”、

“isDetach”、“isNotify”、“isGetState”和“isSetState”属性值为 False 时，表示当前“Operation”元元素的实例代表一个“更新自身状态”操作。

最后规定六个属性的默认值为 False，并且六个属性中任意有一个值为 True 时，其余五个属性值必须为 False。

六个扩展属性的语义如表 3.4 所示。

表 3.4 Operation 元元素的扩展属性

扩展属性名称	类型	语义	缺省值
isAttach	Boolean	当前元元素是否表示“绑定观察者”操作	False
isDetach	Boolean	当前元元素是否表示“解绑观察者”操作	False
isNotify	Boolean	当前元元素是否表示“通知观察者改变状态”操作	False
isGetState	Boolean	当前元元素是否表示“获得目标自身状态”操作	False
isSetState	Boolean	当前元元素是否表示“设置目标自身状态”	False
isUpdate	Boolean	当前元元素是否表示“更新自身状态”操作	False

(4) 扩展 Interface 元元素

为使得建模人员实例化 M2 层的“Interface”元元素后得到的 M1 层的接口元素具有观察者设计模式中“观察者”的语义，所以建模工具设计人员通过继承在“Interface”元元素中增加“isObserver”元属性，数据类型是 Boolean。图 3.12 给出在“Interface”元元素加入语义的结果。

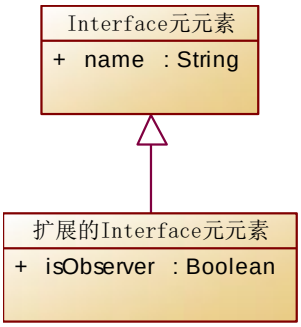


图 3.12 扩展 Interface 元元素

当“isObserver”属性值为 True，表示当前“Interface”元元素的实例代表一个“观察者”接口。当“isObserver”属性值为 False，表示当前“Interface”元元素的实例代表一个普通接口。该属性默认值为 False。

扩展属性的语义如表 3.5 所示。

表 3.5 Interface 元元素的扩展属性

扩展属性名称	类型	语义	缺省值
isObserver	Boolean	当前元元素是否表示“观察者”接口	False

2. 采用实例化方式

继承的方式使得建模工具设计人员可以将观察者、策略和组合等设计模式信息添加到类图元模型中，从而创建供建模人员建模使用的设计模式建模工具。但客观世界中存在各种各样的设计模式，采用继承的方式将导致建模工具设计人员必须在类图建模工具中为每一种设计模式添加一次模式信息，这必然增加了建模工具设计人员的工作量。

为了解决这个问题，建模工具设计人员将每种设计模式都拥有的“参与者”概念抽象出来，并在 M3 元元模型层采用继承的方式扩展“Class”元元元素，最终得到具有“参与者”语义的“Role”元元元素。当需要创建设计模式建模工具时，直接实例化 M3 层的元元模型，获得具有参与者语义的“Role”元元元素的实例。通过这种实例化方式，可以将一个“Role”元元元素实例化为各种设计模式中的参与者，实现了对象的重用。

图 3.13 给出了建模工具设计人员采用继承方式对 M3 元元模型层的 CMOF 元元模型[14]扩展后的结果。图中的框表示扩展的具体内容，建模工具设计人员扩展“Class”元元元素并得到具有“参与者”语义的“Role”元元元素。

“Role”元元元素是“Class”元元元素的子类。通过复用和扩展“Class”的定义，“Role”在抽象语法结构上拥有了多个属性（Property）和多个操作（Operation）。在语义方面，“Role”元元素表示在类图元模型中用于定义设计模式参与者信息的元元素。

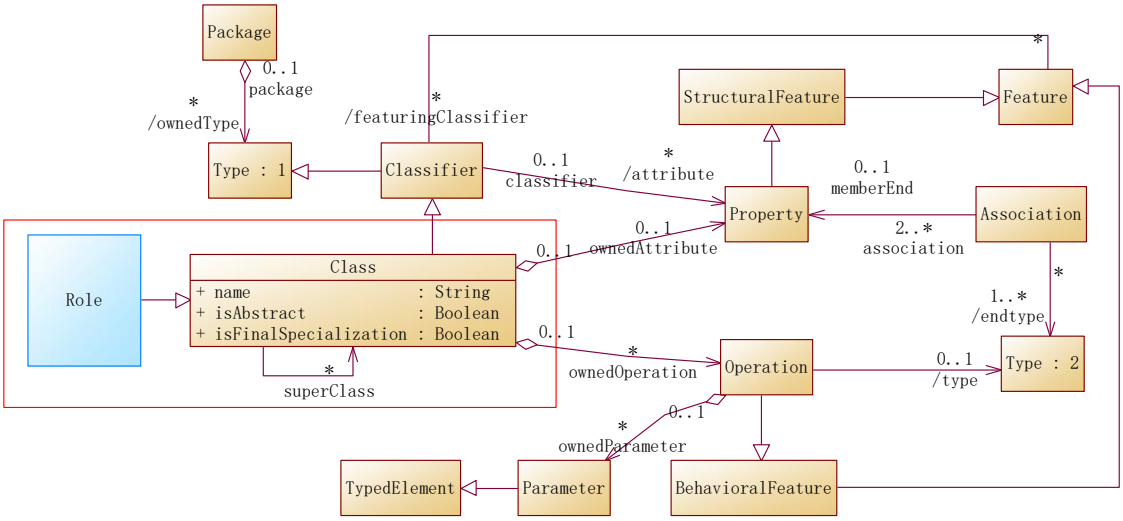
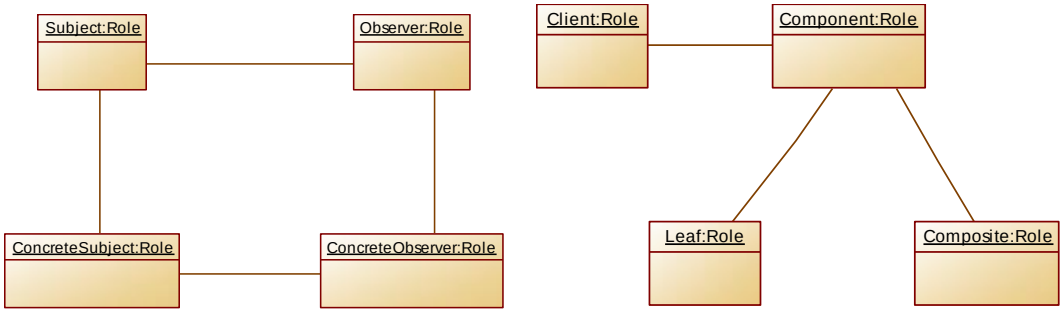


图 3.13 扩展的 CMOF 元元模型

图 3.14 通过对象图的方式给出了当建模工具设计人员需要创建设计模式建模工具时，直接实例化 M3 层的元元模型，获得具有参与者语义的“Role”元元元素的实例，进而创建各种设计模式中参与者的建模工具。



(a) 组合设计模式中的参与者 (b) 观察者设计模式中的参与者

图 3.14 Role 元元元素的实例化结果

对于观察者设计模式，建模工具设计人员将 M3 层“Role”元元元素实例化成了 M2 层“Subject”、“ConcreteSubject”、“Observer”和“ConcreteObserver”四个具有观察者设计模式参与者信息的元元素。对于组合设计模式，建模工具设计人员将 M3 层“Role”元元元素实例化成了 M2 层“Component”、“Leaf”、“Composite”和“Client”四个具有组合设计模式参与者信息的元元素。

3.2.3 对象模型生成状态模型的方法

本文基于 Web 的代码生成方法整体思路中，要实现对象模型到状态模型的转换，首先需要基于类图和状态图的元模型定义对象模型到状态模型的转换规则，然后对转换过程进行描述并在描述中指明转换规则的执行顺序，最后实现将类图表示的对象模型转换到状态图表示的状态模型。下面给出使用 QVT 模型转换描述语言描述的对象模型到状态模型的转换规则。

1. “类”到“状态”的转换规则

```
1.  //获得表示新增、删除、修改和查找页面的状态元素
2. mapping Class:: Class2StartPage ():StartPage
3. {
4.   result.name:= ‘Start_1’;
5.   result.code:= ‘Start_1’;
6. }
7. mapping Class:: Class2FindPage ():FindPage
8. {
9.    result.name:= self.code+ ‘Find’;
10.   result.code:= self.code+ ‘Find’;
11.   result.classifier:= self;
12.   result.stereotype:= ‘FindPage’;
13. }
14. mapping Class:: Class2ListPage ():ListPage
15. {
16.   result.name:= self.code+ ‘List’;
17.   result.code:= self.code+ ‘List’;
18.   result.classifier:= self;
19.   result.stereotype:= ‘ListPage’;
20. }
21. mapping Class:: Class2CreatePage ():CreatePage
22. {
23.   result.name:= self.code+ ‘Create’;
24.   result.code:= self.code+ ‘Create’;
25.   result.classifier:= self;
```

```
26.    result.stereotype:= ‘CreatePage’;
27. }
28. mapping Class:: Class2EditPage ():EditPage
29. {
30.    result.name:= self.code+ ‘Edit’;
31.    result.code:= self.code+ ‘Edit’;
32.    result.classifier:= self;
33.    result.stereotype:= ‘EditPage’;
34. }
```

如果类图表示的对象模型中的元素类型是 Class 元元素，则该元素将生成一个表示状态开始的 State 元素，然后根据类缺省的新增、删除、修改和查找四个操作生成状态图中四个 State 元素。表示状态开始的 State 元素的 name 和 code 属性值为“Start_1”。根据新增、删除、修改和查找操作，Class 元素的 code 属性值分别拼接“Create”、“List”、“Edit”和“Find”字符串，并分别转换为四个 State 元素的 name 和 code 属性值。四个 State 元素的 classifier 属性值被设置为对象模型中的当前对象。四个 State 元素的 stereotype 属性值被分别设置为“CreatePage”、“ListPage”、“EditPage”和“FindPage”字符串。

2. “类”到“转移”的转换规则

```
1.    //获得表示页面跳转的转移元素
2. mapping Class:: Class2StartToFind(in startPage:page, in findPage:page):StartToFind
3. {
4.    result.object1:= startPage;
5.    result. object2:= findPage;
6.    result.name:= startPage.code+ ‘to’+self.code+ ‘Find’;
7.    result. code:= startPage.code+ ‘to’+self.code+ ‘Find’;
8.}
9. mapping Class:: Class2FindToList(in findPage:page, in listPage:page):FindToList
10. {
11.    result. object1:= findPage;
12.    result. object2:= listPage;
13.    esult1.name:= self.code+ ‘Find’ + ‘to’+self.code+ ‘List’;
14.    result1.code:= self.code+ ‘Find’ + ‘to’+self.code+ ‘List’;
```

```
15.  result1.triggerEvent:= self.map Class2FindEvent();
16.}
17. mapping Class:: Class2ListToCreate(in listPage:page, in createPage:page): ListToCreate
18. {
19.  result. object1:= listPage;
20.  result. object2:= createPage;
21.  esult1.name:= self.code+ 'List' + 'to'+self.code+ 'Create';
22.  result1.code:= self.code+ 'List' + 'to'+self.code+ 'Create';
23.  result1.triggerEvent:= self.map Class2ListCreateEvent();
24.}
25. mapping Class:: Class2ListToEdit(in listPage:page, in editPage:page): ListToEdit
26. {
27.  result. object1:= listPage;
28.  result. object2:= editPage;
29.  esult1.name:= self.code+ 'List' + 'to'+self.code+ 'Edit';
30.  result1.code:= self.code+ 'List' + 'to'+self.code+ 'Edit';
31.  result1.triggerEvent:= self.map Class2ListEditEvent();
32.}
33. mapping Class:: Class2CreateToList (in createPage:page, in listPage:page): CreateToList
34. {
35.  result. object1:= createPage;
36.  result. object2:= listPage;
37.  esult1.name:= self.code+ 'Create' + 'to'+self.code+ 'List';
38.  result1.code:= self.code+ 'Create' + 'to'+self.code+ 'List';
39.  result1.triggerEvent:= self.map Class2CrateEvent();
40.}
41. mapping Class:: Class2EditToList (in editPage:page, in listPage:page): EditToList
42. {
43.  result. object1:= editPage;
44.  result. object2:= listPage;
45.  esult1.name:= self.code+ 'Edit' + 'to'+self.code+ 'List';
46.  result1.code:= self.code+ 'Edit' + 'to'+self.code+ 'List';
47.  result1.triggerEvent:= self.map Class2EditEvent();
```

48.}

如果类图表示的对象模型中的元素类型是 Class 元元素，则该元素将生成表示开始状态到查找状态转移的转移元素，并根据类缺省的新增、删除、修改和查找四个操作，生成状态图中“查找状态到列表状态”、“列表状态到新增状态”、“列表状态到修改状态”、“新增状态到列表状态”、“修改状态到列表状态”五个转移元素。因为是针对列表中一条条记录进行删除，所以本文中“列表”状态也代表了“删除”状态。

开始状态元素的 code 属性值拼接“to”字符串，再拼接 Class 元素的 code 属性值，再拼接“Find”转换为“开始状态到查找状态”转移元素的 name 和 code 属性值。“开始状态到查找状态”转移元素的 objec1 和 objec2 属性分别设置为开始状态和查找状态。

Class 元素的 code 属性值拼接“Find to”字符串，再拼接 code 属性值，再拼接“List”字符串转换为“查找状态到列表状态”转移元素的 name 和 code 属性值。“查找状态到列表状态”转移元素的 objec1 和 objec2 属性分别设置为“开始”状态和“删除”状态。

Class 元素的 code 属性值拼接“List to”字符串，再拼接 code 属性值，再拼接“Create”字符串转换为“列表状态到新增状态”转移元素的 name 和 code 属性值。“列表状态到新增状态”转移元素的 objec1 和 objec2 属性分别设置为“列表”状态和“新增”状态。

Class 元素的 code 属性值拼接“List to”字符串，再拼接 code 属性值，再拼接“Edit”字符串转换为“列表状态到修改状态”转移元素的 name 和 code 属性值。“列表状态到修改状态”转移元素的 objec1 和 objec2 属性分别设置为“列表”状态和“修改”状态。

Class 元素的 code 属性值拼接“Create to”字符串，再拼接 code 属性值，再拼接“List”字符串转换为“新增状态到列表状态”转移元素的 name 和 code 属性值。“新增状态到列表状态”转移元素的 objec1 和 objec2 属性分别设置为“新增”状态和“列表”状态。

Class 元素的 code 属性值拼接“Edit to”字符串，再拼接 code 属性值，再拼接“List”字符串转换为“修改状态到列表状态”转移元素的 name 和 code 属性值。

“修改状态到列表状态”转移元素的 objec1 和 objec2 属性分别设置为“修改”状态和“列表”状态。

每个“类”到“转移”的转换规则中将调用对应的“类”到“事件”的转换规则，用以生成触发当前转移的事件元素。下面给出“类”到“转移”转换规中调用的“类”到“事件”的转换规则。

3 “类”到“事件”的转换规则

```
1.  //获得表示触发页面跳转的事件元素
2. mapping Class:: Class2FindEvent ():FindEvent
3. {
4.  result.name:= ‘Find’;
5.  result.code:= ‘Find’;
6.  result.stereotype:= ‘FindAction’;
7. }
8. mapping Class:: Class2 ListCreateEvent ():ListCreateEvent
9. {
10.  result.name:= ‘ListCreate’;
11.  result.code:= ‘ListCreate’;
12.  result.stereotype:= ‘ListCreateAction’;
13. }
14. mapping Class:: Class2ListEditEvent (): ListEditEvent
15. {
16.  result.name:= ‘ListEdit’;
17.  result.code:= ‘ListEdit’;
18.  result.stereotype:= ‘ListEdit Action’;
19. }
20. mapping Class:: Class2List CrateEvent (): CrateEvent
21. {
22.  result.name:= ‘Crate’;
23.  result.code:= ‘Crate’;
24.  result.stereotype:= ‘CrateAction’;
25. }
26. mapping Class:: Class2ListEditEvent (): EditEvent
27. {
```

```
28.    result.name:= ‘Edit’;
29.    result.code:= ‘Edit’;
30.    result.stereotype:= ‘Edit Action’;
31. }
```

如果类图表示的对象模型中的元素类型是 Class 元元素，则该元素将根据类缺省的新增、删除、修改和查找四个操作生成状态图中五个 Event 元素。根据新增操作，Event 元素的 name 和 code 属性值设置为“Create”字符串。根据删除操作，Class 元元素实例将生成两个 Event 元素，一个 Event 元素的 name 和 code 属性值设置为“ListCreate”，另一个 Event 元素的 name 和 code 属性值设置为“ListEdit”。根据修改操作，Event 元素的 name 和 code 属性值设置为“Edit”。根据查找操作，Event 元素的 name 和 code 属性值设置为“Find”。

4 转换过程描述

定义对象模型到状态模型的转换规则后，需要对转换过程进行描述。过程描述中阐述了转换规则的执行顺序，进而实现对象模型到状态模型的转换。下面给出对象模型到状态模型的转换过程描述。

```
1. transformation ClassDiagramToStateDiagram (in  srcModel: ClassDiagram, out  tarModel:
StateDiagram);
2. main() {
3.    //调用“类”到“状态”的转换规则
4.    var startPage = srcModel.objectsOfType(Class)->map Class2StartPage();
5.    var findPage = srcModel.objectsOfType(Class)->map Class2FindPage();
6.    var listPage = srcModel.objectsOfType(Class)->map Class2ListPage();
7.    var createPage = srcModel.objectsOfType(Class)->map Class2CreatePage();
8.    var editPage = srcModel.objectsOfType(Class)->map Class2EditPage();
9.    //调用“类”到“转移”的转换规则
10.    srcModel.objectsOfType(Class)->map Class2StartToFind(startPagee, findPage);
11.    srcModel.objectsOfType(Class)->map Class2FindToList(findPage, listPage);
12.   srcModel.objectsOfType(Class)->map Class2ListToCreate(listPage, createPage);
13.   srcModel.objectsOfType(Class)->map Class2ListToEdit(listPage, editPage);
14.   srcModel.objectsOfType(Class)->map Class2CreateToList(createPage, listPage);
15.   rcModel.objectsOfType(Class)->map Class2EditToList(editPage, listPage);
16. }
```

3.3 生成表示层代码的方法

表示层框架代码的生成过程对应了本文 Web 代码生成方法整体思路中状态模型到表示层代码的转换过程。

为了告诉计算机需要生成的表示层代码的样貌，首先给出基于 MVC 框架模式的表示层代码的框架结构，然后定义状态模型到 JSF 表示层框架代码的转换规则，接着根据转换规则使用 GTL 创建需要生成的表示层代码的 GTL 模板，最后在转换过程中通过应用 GTL 模板实现根据转换规则转换状态模型到表示层框架代码。

3.3.1 表示层代码框架

JavaServer Faces(简称 JSF)是在 Sun 公司提出的 Web 开发标准，旨在推动基于 Java 的 Web 用户界面开发的简易性，它包含一组用于表示用户界面组件并管理其状态、处理事件等内容的应用程序编程接口(Application Programming Interface, API)和一个用于在 JSP 页面中表示 JSF 组件的自定义标签库[48]。JSF 表示层框架应用了 MVC 框架模式，其中 JSF 框架与 MVC 框架模式的关系如图 3.15 所示。

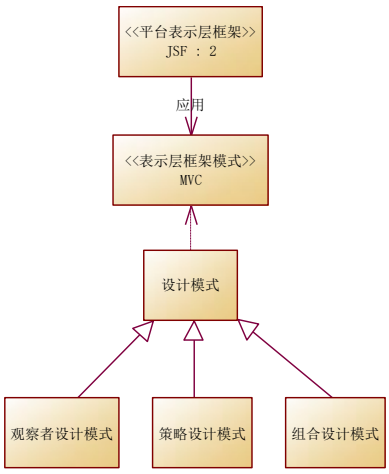


图 3.15 JSF 框架与设计模式的关系

JSF 框架应用了表示层 MVC 框架模式，MVC 框架模式应用了观察者、策略和组合设计模式，因此添加表示层框架信息的过程完成了设计模式信息的添加。

既然 JSF 表示层框架应用了 MVC 框架模式，那么基于本文的 Web 代码生成方法生成的 JSF 表示层代码包括视图代码、控制代码和模型代码。视图代码包括表示

新增页面、删除页面、修改页面和查找页面的 JSF 代码。控制代码包括新增页面、删除页面、修改页面和查找页面分别对应的 PageBean 代码。模型代码包括通过领域模型生成的 Java 类代码。图 3.16 给出生成的具有 MVC 框架模式的表示层代码的框架。

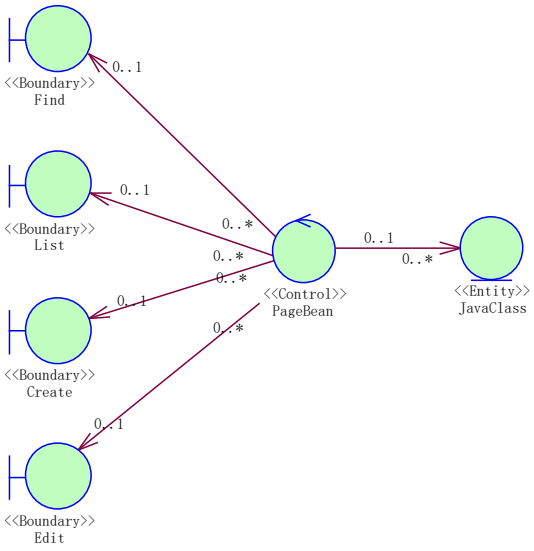


图 3.16 表示层代码框架

“Find”、“List”、“Create”和“Edit”四个边界类分别表示生成的“查找页面”、“删除页面”、“新增页面”和“修改页面”，它们对应了 MVC 框架模型中的“View”。“PageBean”控制类用于处理用户与上述页面交互过程中触发的事件，它对应 MVC 框架模型中的“Control”。“JavaClass”实体类表示页面数据的来源以及用于处理页面业务逻辑，它对应了 MVC 框架模型中的“Model”。

“JavaClass”实体类代码的生成过程并不是以状态模型为入口，它的生成过程对应着本文 Web 代码生成方法整体思路中“领域模型到对象模型”和“对象模型到关系模式”的转换过程。如前文所述，建立 O/R 映射得到关系模式后，通过关系模式可以进一步生成 SQL 代码文件。有了 SQL 代码文件后，可以直接在数据库管理系统中应用 SQL 文件完成数据库表的建立。由于该过程涉及 O/R 映射、数据访问和存储等技术，所以“模型”代码的生成过程与数据访问层的代码生成有关，本文不作重点讨论。本节的表示层代码生成方法重点讨论“视图”代码和“控制”代码的生成。

3.3.2 生成视图代码的方法

本文基于 Web 的代码生成方法整体思路中，要实现状态模型到表示层“视图”代码的转换，首先需要定义状态模型到 JSF 表示层视图代码的转换规则，后对转换过程进行描述并在描述中指明转换规则的执行顺序，最终实现将状态模型生成 JSF 表示层视图代码。下面以状态模型中表示“修改页面”的状态元素为例，给出使用 QVT 模型转换描述语言描述的状态模型到 JSF 表示层视图代码核心部分的转换规则。

```
1.  //生成表单
2.  mapping  State:: State2Form (): Form
3. {
4.  var classifier:= self.classifier;  //获取学生对象
5.  result:=self.map Class2Form (classifier);  //转换学生对象到表单
6. }
7. mapping  Class:: Class2Form (in classifier):Form
8. {
9.  result.id:= classifier.code+ ‘EditForm’;  //设置表单id
10.  self.ownedElement->map Attr2InputText ();  //遍历学生对象中的属性并将属性依次转换为输入框
11. }
12.  //生成输入框
13.  mapping  Attribute:: Attr2InputText ():InputText
14. {
15.  result.value:= ‘#{ sTUDENTEditPageBean. sTUDENT.’+ self.code+‘}’;  //设置输入框的内容
16.  result.id:= self.code;  //设置输入框的id
17. }
18.  //生成超链接
19. mapping  State:: State2CommandLink():CommandLink
20. {
21.  var classifier:= self.classifier;  //获取学生对象
22.  result:=self.map Class2CommandLink(classifier);  //转换学生对象到超链接
23. }
24. mapping  Class:: Class2CommandLink(in classifier): CommandLink
```

```
25. {
26.     result.action:= '#{'+ classifier.code +'EditPageBean.invoke_update' +'}';   //设置超链接的跳转路径
27. }
```

State 元素将转换为<h:form>、<h:inputText>、<h:commandLink>等标签。查找 State 元素的“Classifier”属性值，根据属性值获取对象模型中生成该 State 元素的 Class 元素，将 Class 元素的 code 属性值拼接“EditForm”字符串转换为<h:form>标签的 id 属性值。Class 元素拥有的 Attribute 元素的 code 属性值转换为<h:iutputText>标签的 value 属性值，属性值的形式为 EL 语言表示的“#{%PageBean 名称%.%实体名称%.%code 属性值%}”，其中%code 属性值%将被替换。Class 元素的 code 属性值转换为<h:iutputText>标签的 id 属性值。Class 元素的 code 属性值转换为<h:commandLink>标签的 action 属性值，值的形式为 EL 语言表示的“#{%code 属性值%EditPageBean.invoke_update}”，其中%code 属性值将被替换。

这在代码逻辑里也看不出来

定义状态模型到 JSF 表示层视图代码的转换规则后，需要对转换过程进行描述。过程描述中阐述了转换规则的执行顺序，进而实现状态模型到 JSF 表示层视图代码的转换。下面给出状态模型到“修改页面”JSF 表示层视图代码核心部分的转换过程描述。

```
1. transformation StateDiagramToJSFCode (in   srcModel: StateDiagram, out   tarModel: JSFCode);
2. main() {
3.     //调用“状态”到“JSF表示层视图代码”的转换规则
4.     srcModel.objectsOfType(State)->map State2Form ();
5.     srcModel.objectsOfType(State)->map State2CommandLink();
6. }
```

3.3.3 生成控制代码的方法

本文基于 Web 的代码生成方法整体思路中，要实现状态模型到表示层“控制”代码的转换，首先需要定义状态模型到 JSF 表示层控制代码的转换规则，然后对转换过程进行描述并在描述中指明转换规则的执行顺序，最终将状态模型生成 JSF 表示层控制代码。下面以状态模型中表示“修改页面”的状态元素为例，给出使用 QVT 模型转换描述语言描述的状态模型到 JSF 表示层控制代码的转换规则。

```
1.  //生成PageBean类
2. mapping  State:: State2PageBeanClass ():PageBeanClass
3. {
4.  result.visibility:= 'public';  //设置可见性
5.  var classifier:= self.classifier;  //获取学生对象
6.  result.className:= classifier.code+' EditPageBean';  //设置类名
7.  result.superClassName:= classifier.code+' PageBeanBase';  //设置父类名
8. }
9.  //生成get方法
10. mapping  State:: State2Get():Get
11. {
12.  result.visibility:= 'public';  //设置可见性
13.  var classifier:= self.classifier;  //获取学对象
14.  result.returnType:= classifier.code;  //设置get方法返回类型
15.  result.name:= 'get'+ classifier.code;  //设置get方法名称
16. }
17.  //生成更新方法
18. mapping  State:: State2Update():Update
19. {
20.  result.visibility:= 'public';  //设置可见性
21.  result.returnType:= 'String';  //设置update方法返回类型
22.  result.name:= 'invoke_update';  //设置update方法名称
23. }
24.  //生成取消方法
25. mapping  State:: State2Cancel():Cancel
26. {
27.  result.visibility:= 'public';  //设置可见性
28.  result.returnType:= 'String';  //设置cancel方法返回类型
29.  result.name:= 'invoke_cancel';  //设置cancel方法名称
30. }
```

State 元素将转换为一个 PageBean 类代码。查找 State 元素的“Classifier”属性值，根据属性值找到对象模型中生成该 State 元素的 Class 元素，Class 元素的 code

属性值拼接字符串“EditPageBean”转换为 PageBean 类的类名。Class 元素的 code 属性值拼接字符串“PageBeanBase”转换为 PageBean 类的父类名称。Class 元素的 code 属性值转换为 get 方法的返回类型。字符串“get”拼接 Class 元素的 code 属性值转换为 get 方法的方法名，Class 元素的 code 属性值转换为 get 方法的返回类型。另外，转换还将生成名为“invoke_update”和“invoke_cancel”

定义状态模型到 JSF 表示层控制代码的转换规则后，需要对转换过程进行描述。过程描述中阐述了转换规则的执行顺序，进而实现状态模型到 JSF 表示层控制代码的转换。下面给出状态模型到“修改页面”JSF 表示层控制代码的转换过程描述。

```
1. transformation StateDiagramToPageBean (in srcModel: StateDiagram, out    tarModel: PageBean);
2. main() {
3.     //调用“状态”到“JSF表示层控制代码”的转换规则
4.     srcModel.objectsOfType(State)->map State2PageBeanClass ();
5.     srcModel.objectsOfType(State)->map State2Get ();
6.     srcModel.objectsOfType(State)->map State2Update();
7.     srcModel.objectsOfType(State)->map State2Cancel ();
8. }
```

3.4 本章小节

本章首先结合 MDA 中模型表示和模型转换思想，介绍了本文的 Web 代码生成方法思路。然后针对建模方法，介绍了建模人员建模和建模工具设计人员设计建模工具两个过程中继承和实例化在实现重用上的区别，并结合继承和实例化两种方式讨论了在模型转换中加入设计模式信息的方法。最后基于类图和状态图的元模型，给出了对象模型到状态模型、状态模型到表示层框架代码的转换规则和转换过程描述。

第 4 章 Web 代码生成方法的应用

本章以程序自主学习系统为例，介绍了如何在 PowerDesigner 中实现使用类图表示的程序自主学习系统领域模型到使用状态图表示的状态模型的转换，介绍了如何将状态模型转换到应用了 JSF 框架的表示层代码。

4.1 程序设计自主学习场景简介

《C++程序设计教程》是我校大部分理工科学生的公共基础课程，其教学目的是培养学生的计算思维能力和读程序基本能力。但通过任课老师课堂授课方式难以满足学生学习的要求并且目前使用的自主学习软件在体现我校教学的特色和重点上还存在不足。因此，根据我校学生的实际情况设计一个程序设计自主学习系统。程序设计自主学习系统包含学生学习、任课教师授课、教学管理员管理这三个业务场景。本文重点对学生学习场景中的学生练习进行讨论。

4.2 程序设计自主学习系统建模

程序设计自主学习系统建模分为“程序自主学习系统领域模型”、“标记分析模型”和“生成设计模型”三个部分。

1. 建立程序自主学习系统领域模型

领域模型是 Web 代码自动生成的入口，在系统分析阶段根据需求分析阶段对学生练习用例分析结果，将每个用例描述中涉及的实体、实体之间的关系以及基本的业务进行抽象，得到如图 4.1 所示的学生练习的领域模型，完成 PIM 的创建。

2. 标记分析模型

领域模型是建模人员在系统业务分析或系统分析阶段对客观世界中的对象抽象后得到的结果。在系统设计阶段，客观世界中的对象需要转换为计算机世界中的对象。

本文的 Web 代码生成方法整体思路中，领域模型需根据标记转换为对象模型。如图 4.2 所示，为了完成上述过程，建模人员使用 PowerDesigner 实现平台提供的

Java 语言平台信息标记领域模型，进而指导领域模型转换为具有 Java 语言的数据类型等信息的对象模型。

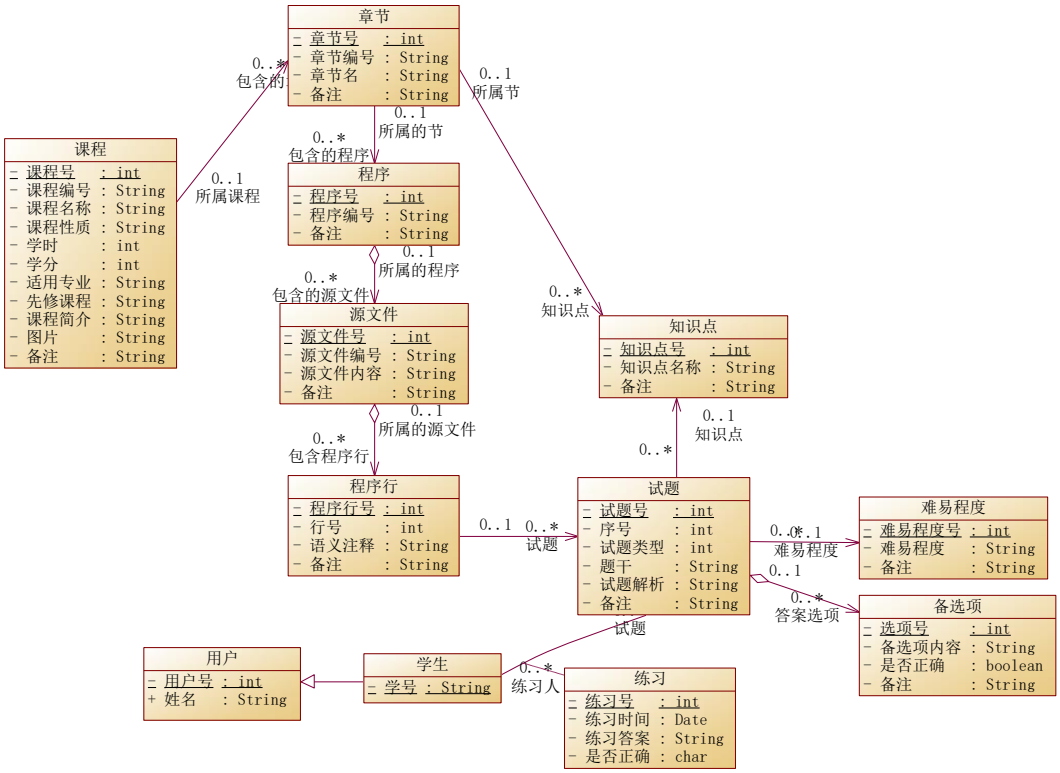


图 4.1 学生练习领域模型

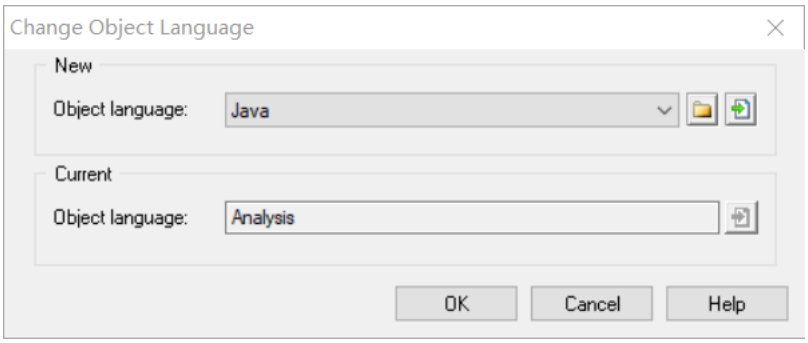


图 4.2 Java 语言标记

3. 生成设计模型

生成设计模型包含“生成对象模型”、“生成关系模型”、“扩展状态元模型”、“生成状态模型”和“创建表示层设计模型”五个部分。

(1) 生成对象模型

每种编程语言的名称作为标记的名称，上述选择语言的过程完成了使用标记标记 PIM 的过程。描述客观世界中学生练习场景的领域模型模型被标记后形成了描述程序设计自主学习系统中学生练习的对象模型，进而完成了被标记的 PIM 的创建。图 4.3 展示了得到的对象模型。

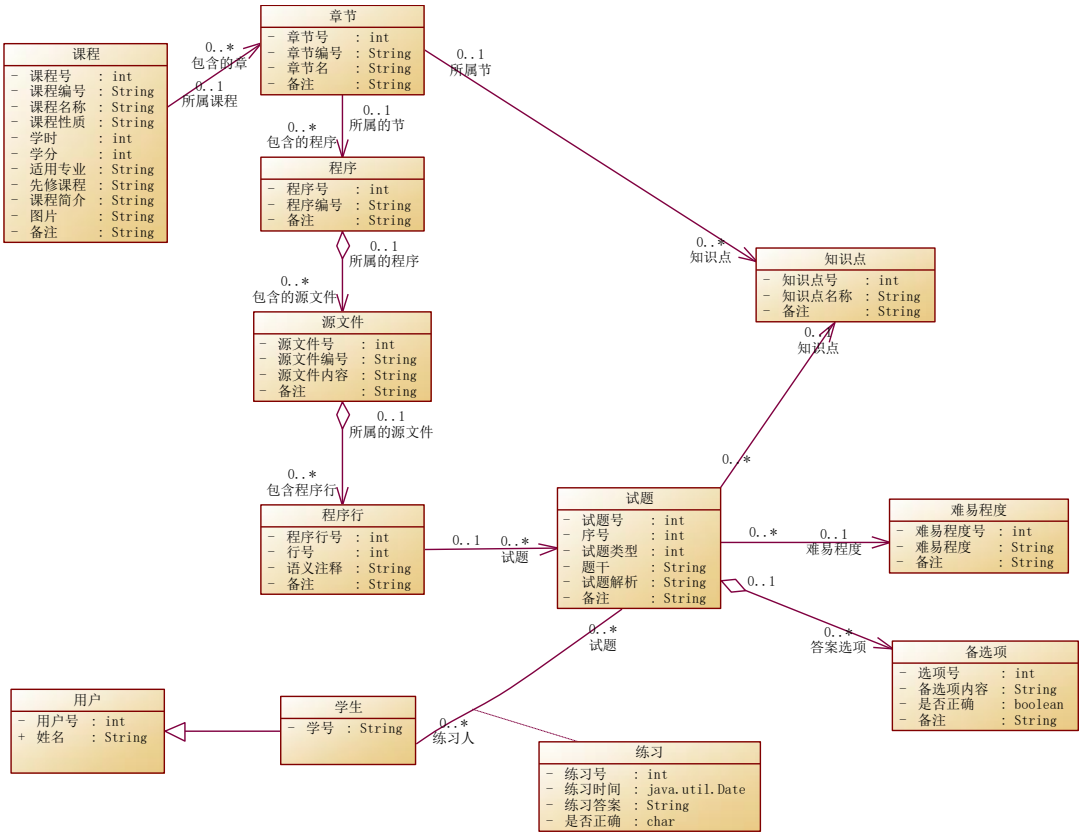


图 4.3 学生练习对象模型

其中“练习”类中的“练习时间”属性的数据类型从平台无关的“Date”类型变为了与 Java 语言相关的“java.util.Date”类型。

(2) 生成关系模型

表示层的对象通过业务逻辑层的对象获取数据，而业务逻辑层的对象通过数据访问层的对象从数据库中获取数据。数据访问层的对象要获取数据库中的数据需建立 O/R 映射关系。如图 4.4 所示，建模人员使用 PowerDesigner 工具的“Generate Physical Data Model”功能，生成 O/R 映射关系。

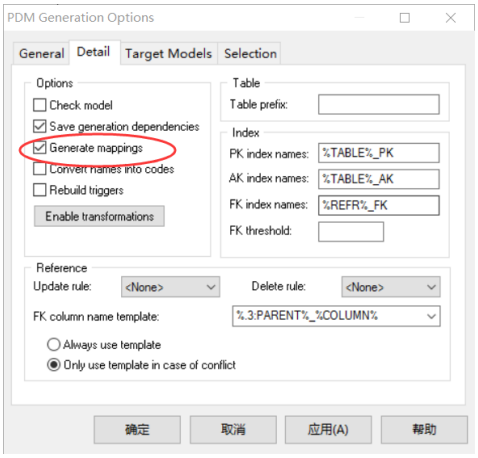


图 4.4 设置 O/R 映射

图 4.5 是生成的关系模型。建模人员可以在 PowerDesigner 工具中将关系模型直接生成数据库 SQL 文件，然后在数据库管理工具中运行生成的 SQL 文，最后完成数据库表的创建。

本文的研究重点在表示层代码的生成，而上述过程主要涉及数据访问层的操作，因此不是本文研究的重点，下文不具体阐述。

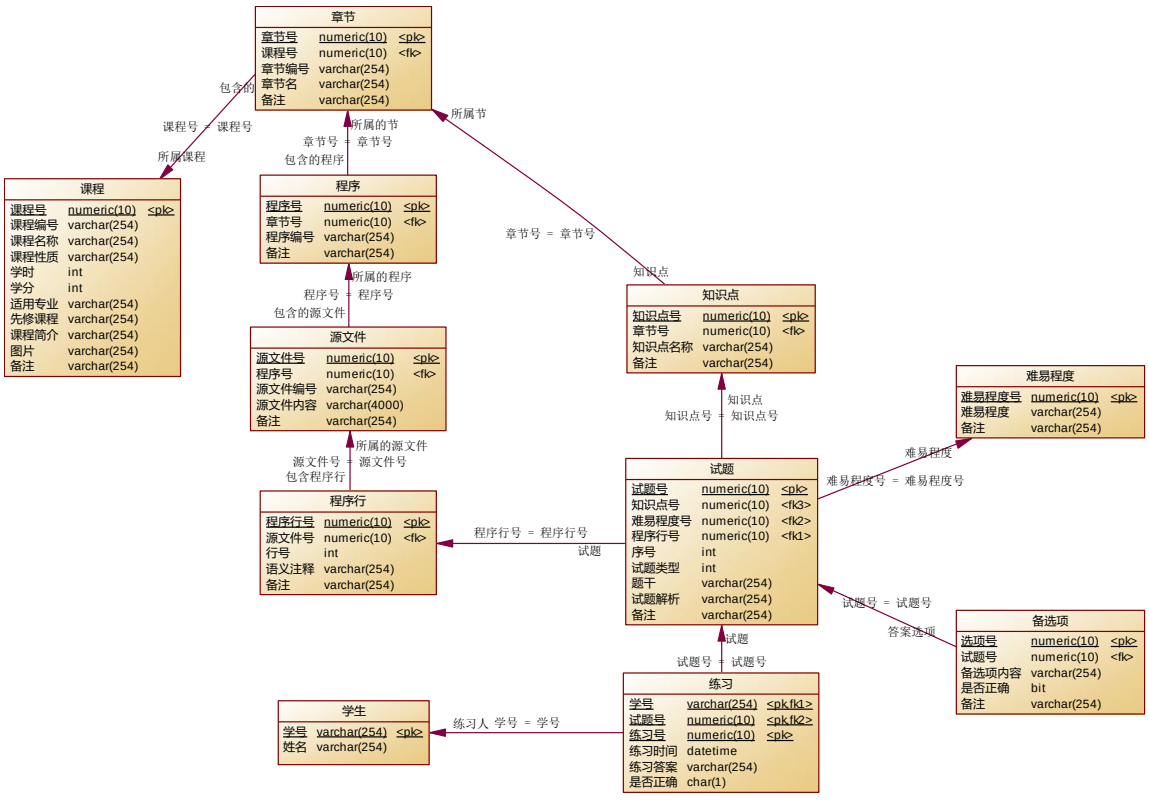


图 4.5 学生练习关系图

(3) 扩展状态图元模型

本文的 Web 代码生成方法整体思路中，对象模型需根据类图到状态图的转换规则生成状态模型。因此，学生练习对象模型作为被标记的平台无关模型，根据类图到状态图的映射规则转换为使用状态图表示的状态模型。

如图 4.6 和图 4.7 所示，为了将 JSF 表示层框架信息、JSF 表示层框架应用的设计模式信息和数据访问层信息扩展到状态图元模型中，使得转换对象模型得到的状态模型包含 JSF 框架、设计模式等信息的状态模型，建模人员使用 PowerDesigner 工具添加 JSF 扩展文件和 Hibernate 扩展文件。

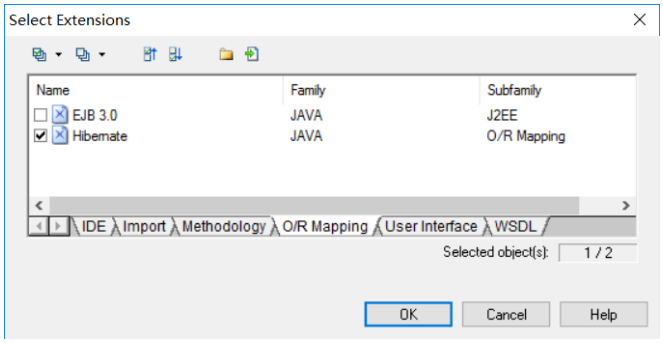


图 4.6 添加数据访问层框架

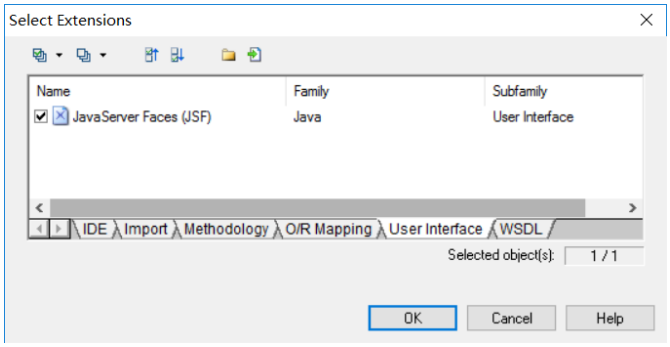


图 4.7 添加表示层框架

为使得实例化状态图元模型生成的状态图中每个状态元素表示一个表示层的页面并且能够清晰地区分各个页面。

首先，建模人员通过 PowerDesigner 中实现的 UML Profile 扩展机制，在资源文件编辑器中的“State”元元素下创建表示页面的“Page”构造型。

然后，在“Page”构造型下创建表示新增页面的“CreatePage”、表示修改页面的“EditPage”、表示查找页面的“FindPage”、表示删除和列表页面的“ListPage”、表示引导页面的“FrontPage”、表示的登录页面的“LoginPage”和表示主页面的“IndexPage”构造型。

最后，为每个表示页面语义的构造型设置图形符号，完成具有页面语义的构造型的创建工作。

创建结果如图 4.8 所示。创建构造型的过程完成了状态图元模型的状态元元素语义的扩展。

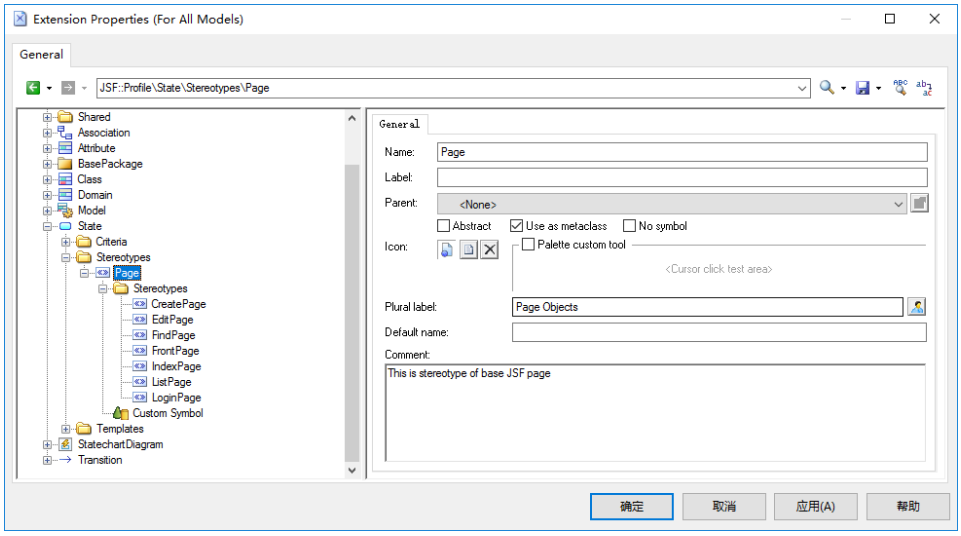


图 4.8 创建构造型

(4) 生成状态模型

为了在 PowerDesigner 实现平台中将对象模型转换为状态模型，建模人员根据类图到状态图的转换规则，在 PowerDesigner 中编写 VBScript 脚本并执行。下面的 VBScript 脚本代码展示了在 PowerDesigner 工具中，如何根据不同构造型的值，在状态模型中创建具有不同页面语义的状态元素。

```
//根据不同的构造型的值创建状态元素
Function GeneratePage(pageflowDiagram, pageName, stereotype)
    Dim page
    //根据状态元素名称获取状态元素
    Set page = GetPage(pageflowDiagram.Package, pageName)
    //如果该元素不存在
```

```
    If page is Nothing Then
        Set page = CreatePage(pageflowDiagram.Package, pageName, stereotype)   //新建
        Call pageflowDiagram.AttachObject(page)   //将新建的状态元素增加到状态图中
    Else
        //判断当前状态元素是否加入到了状态图中
        If Not IsAttachedPage(pageflowDiagram, page) Then
            Call pageflowDiagram.AttachObject(page)
        End If
    End If
    Set GeneratePage = page
End Function

//创建一个新的状态元素
Function CreatePage(package, pageName, stereotype)
        Set CreatePage = package.CreateObject(PdOOM.cls_State)   //创建状态元素
        CreatePage.Name = pageName   //设置状态元素 name 属性值
        CreatePage.Code = pageName   //设置状态元素 code 属性值
        CreatePage.Stereotype = stereotype   //设置状态元素表示的界面类型
End Function
```

建模人员在 PowerDesigner 中使用 VBScript 脚本实现类图到状态图的转换规则后，将对象模型自动生成状态模型。每个对象模型中的对象生成一张状态图，其中每个对象根据默认的增加、删除、修改、查找四个方法生成状态模型中“CreatePage”、“EditPage”、“FindPage”和“ListPage”四个状态以及状态之间的转移等。图 4.9 给出了学生练习对象模型中“学生”对象生成的状态模型。

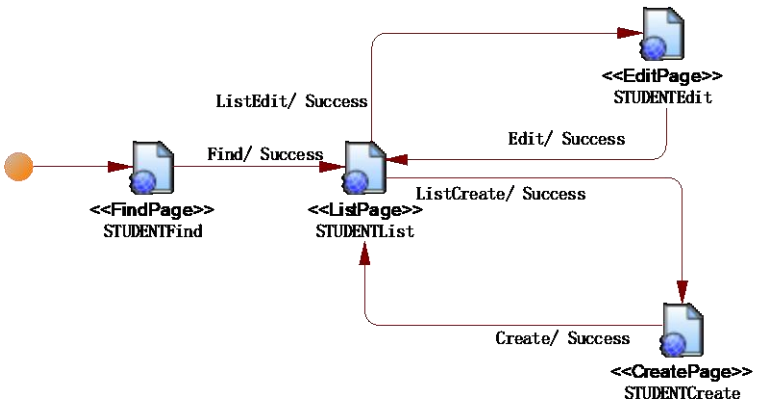


图 4.9 学生状态模型

构造型“CreatePage”表示当前状态元素代表一个新增学生界面、“EditPage”表示当前状态代表一个修改学生界面、“FindPage”表示当前状态代表一个查找学生界面、“ListPage”表示当前状态代表一个学生列表和删除学生界面。

(5) 创建表示层设计模型

Web 表示层应用了 MVC 框架模式，建模人员在系统设计阶段需要构建应用 MVC 框架思想的表示层设计模型。它包括用于向用户展示数据的视图设计模型、用于实现数据交互的控制设计模型和用于实现数据持久化的实体设计模型。

图 4.10 以“学生练习”领域模型中“学生”对象为例，给出在系统设计阶段创建的应用 MVC 框架模式的 Web 表示层代码设计模型。

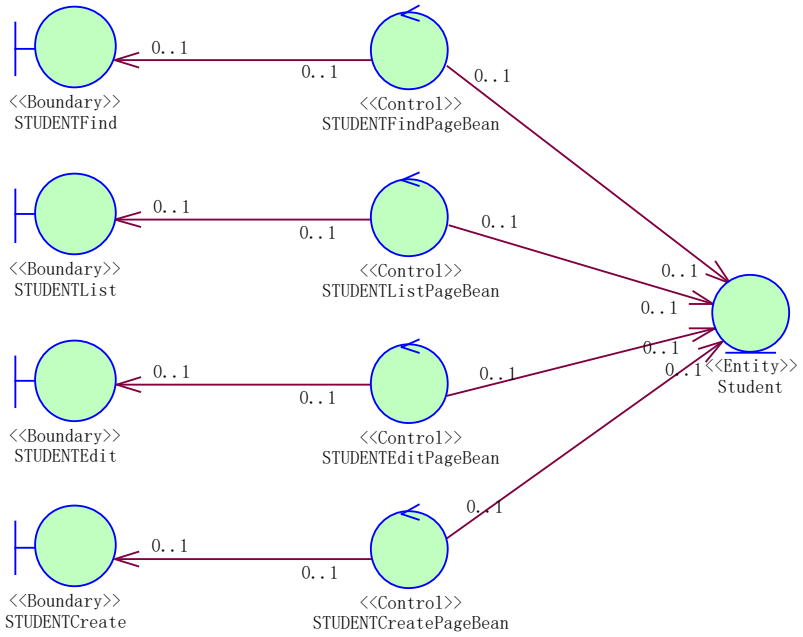


图 4.10 学生 Web 表示层代码设计模型

“STUDENTFind”、“STUDENTList”、“STUDENTCreate”和“STUDENTEdit”四个边界类分别表示生成的“查找学生页面”、“删除学生页面”、“新增学生页面”和“修改学生页面”，它们对应了 MVC 框架模型中的“View”。

“STUDENTFindPageBean”、“STUDENTListPageBean”、“STUDENTEditPageBean”和“STUDENTCreatePageBean”控制类表示用于处理用户与上述页面交互过程中触发的事件，它对应 MVC 框架模型中的“Control”。

“Student”实体类表示学生信息的来源以及页面业务逻辑的处理，它对应了 MVC 框架模型中的“Model”。

图 4.11 所示的“修改学生”控制类的细化。

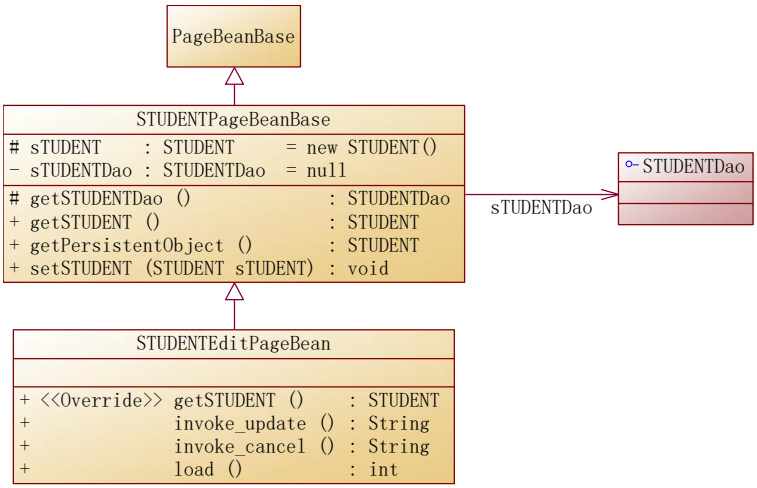


图 4.11 修改学生控制设计模型

该类图表示“STUDENTEditPageBean”控制类继承“STUDENTPageBeanBase”，“STUDENTPageBeanBase”又继承“PageBeanBase”。“STUDENTEditPageBean”控制类中包含用于获取学生实例的“getSTUDENT”方法、用于实现学生更新的“invoke_update”方法、用于取消更新操作的“invoke_cancel”方法等。

图 4.12 给出了使用类图表示的“学生”实体类的细化。

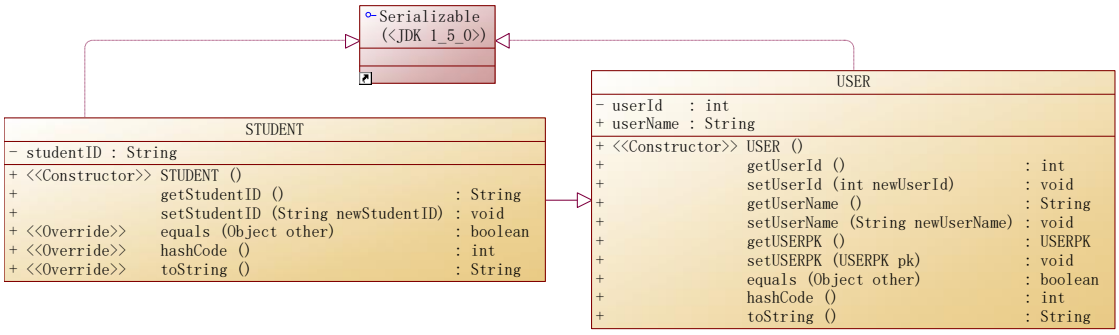


图 4.12 学生实体设计模型

学生实体继承用户实体，用户和学生都实现了“Serializable”接口。学生实体中除了包含通过继承用户实体获得的“userId”和“userName”属性外，还包含自

已特有的“studentID”属性，它包含对父类中“equals”、“hashCode”、“toString”方法的重写外，还拥有自己特有的用于实例化学生的“STUDENT”构造函数、用于获取学生 ID 的“getStudentID”、用于设置学生 ID 的“setStudentID”方法。

图 4.13 展示了基于 MVC 框架模式的“修改学生”表示层中视图、控制和实体三个组件间的交互过程。

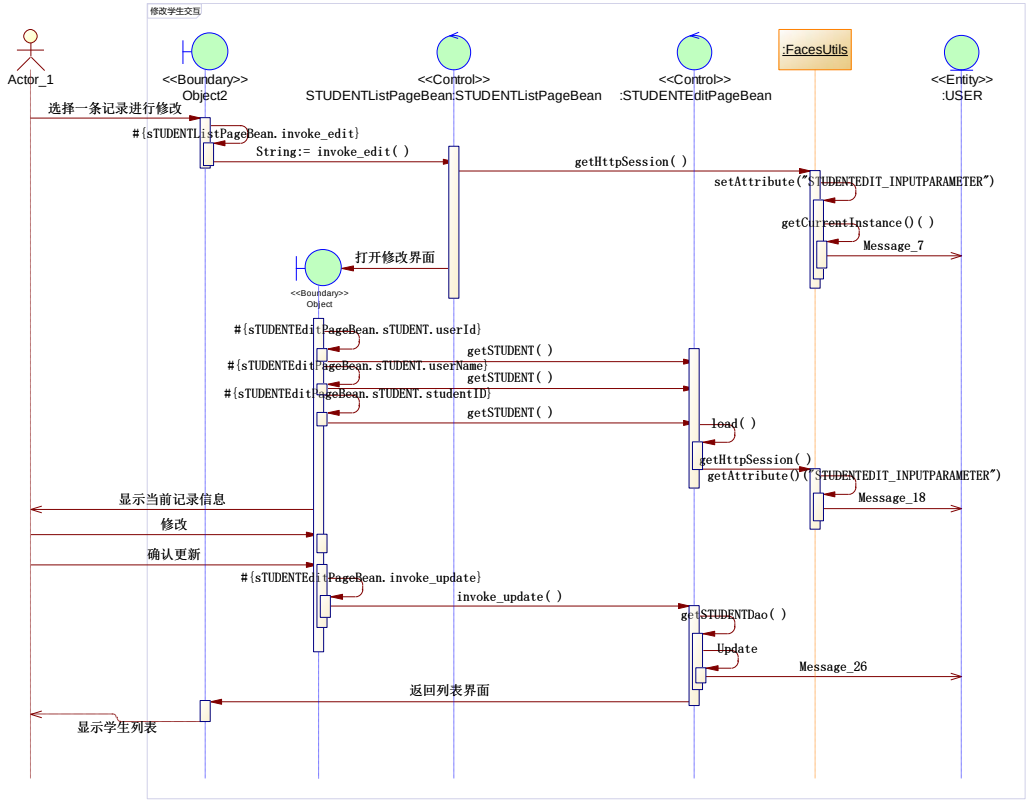


图 4.13 修改学生表示层组件交互模型

用户在学生列表界面选择某个学生记录后，点击修改按钮。“STUDENTList PageBean”控制器调用其中的“invoke_edit”方法，该方法通过调用“FacesUtils”中的“getHttpSession”方法获取会话对象，会话对象记录了每一位用户的交互数据并与其它用户进行了区分。通过调用会话对象中的“setAttribute”以及“getCurrent Instance()”方法，将从数据库获取的当前学生记录放到会话对象中。“STUDENTList PageBean”控制器打开学生修改界面，界面通过 EL 表达式语言完成与 PageBean 控制器的交互。通过“#{sTUDENTEditPageBean.sTUDENT.userId}”、“#{sTUDENTEdit

PageBean.sTUDENT.userName}”、“#{sTUDENTEditPageBean.sTUDENT.studentID}”分别调用“STUDENTPageBean”中的“getSTUDENT”方法，该方法调用“FacesUtils”中的“getHttpSession”方法获取会话对象，然后通过调用会话对象中的“setAttribute”方法获取存储在对话中的学生实体以及学生实体中“userId”、“userName”和“studentID”的值并填充到界面标签中，从而完成数据的显示。当用户修改完学生信息，点击确认按钮后，界面通过 EL 表达式语言“#{sTUDENTEditPageBean.invoke_update}”调用“STUDENTPageBean”中的“invoke_update”方法完成学生信息的持久化操作。

4.3 程序设计自主学习系统表示层代码生成

在系统设计阶段，通过对基于 MVC 框架模式的表示层构建三种设计模型，计算机在代码生成阶段就可以根据设计模型包含的信息，重用设计模型实现表示层代码的自动生成。根据系统设计阶段创建的表示层设计模型，生成的表示层代码包括用于显示数据的 JSF 标签代码、用于响应用户事件的 PageBean 代码和用于实现数据持久化操作的 Java 实体类代码。下文主要讨论表示层显示数据的 JSF 标签代码和用于响应用户事件的 PageBean 代码生成过程。

领域模型到对象模型的转换中加入了 Java 平台的编程语言信息，对象模型到状态模型的转换中加入了 JSF 表示层框架和 Hibernate 数据访问层框架信息，这使得状态模型具备了生成表示层代码的语义。生成代码过程需要首先根据状态图到表示层代码的映射规则，使用 GTL 表示出表示层代码生成模板，然后通过 PowerDesigner 的代码生成机制自动生表示层代码。

下文通过状态模型中“STUDENTEdit”状态元素到表示层代码的生成过程为例，给出代码生成过程。

建模人员使用 GTL 模板语言表示“学生修改页面”的 GTL 模板中的核心代码如下。

```
<h:form id="%editPageName%Form">
    <table class="formPage" width="100%">
        <tr>
            <td></td>
            <td align="left">
```

```
                    <h:outputText value="Edit %.FU:Name%" styleClass="headerText"/>
                </td>
            </tr>
            .foreach_item(Attributes,,,%isAttributeMapped%)
                <tr>
                    <td align="right" width="%Model.FormLabelWidth%" class"controlLabel">
                        <h:outputText value="%.FU:Name%:"/>\n
                        .if (%Mandatory%) and (%_pageState% != "find")
                            <h:outputText value=" *" style="color: red"/>\n
                        .endif
                    </td>
                    <td align="left" width="*">\n
                        <h:inputText value="#{%beanName%.%instance%.%Code%}"
id="%Code%"
                        required="true"  readonly="true">
                            <f:validateLength minimum="1"/>
                        </h:inputText>
                        <h:message for="%Code%" styleClass="errorMessage"/>
                    </td>
                </tr>
            .next(\n)
            <tr>
                <td height="50"></td>
                <td align="left">
                    <div class="commandBar">
                        <span>
                            <h:commandLink styleClass="commandLink"
action="#{%qualifiedCode%EditPageBean.invoke_update}">
                                <h:outputText value="%updateTitle%"/>
                            </h:commandLink>
                        </span>
                        <span>
                            <h:commandLink styleClass="commandLink"
action="#{%qualifiedCode%EditPageBean.invoke_cancel}"
                            immediate="true">
                                <h:outputText value="%cancelTitle%"/>
                            </h:commandLink>
```

```
                        </span>\n
                    </div>
                </td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <h:messages styleClass="errorMessage" globalOnly="true"/>
                </td>
            </tr>
        </table>
    </h:form>
```

“% %”语法用于在代码生成过程读取模型中元素的属性值并进行替换。转换过程中计算机根据转换规则和重用“修改学生”界面设计模型，首先查找“STUDENTEdit”状态元素的“Classifier”属性，得到其属性值为“学生”，然后根据属性值找到该状态元素对应的对象模型中的“学生”类。最后将对象模型中的“学生”类的“Code”属性值取出替换 GTL 模板中%Code%或%qualifiedCode%，将对象模型中的“学生”类的属性的“Name”属性值取出替换 GTL 模板中%FU：Name%等占位符，从而完成表示层修改学生界面代码的生成。

同理，根据转换规则，下面给出使用 GTL 表示的“修改学生”PageBean 代码的结果。

```
public class %Code%EditPageBean extends    %Code%PageBeanBase
{
    public %Code% get%Code%()
    {
        try {
            this.%.FL:Code%                                                    =
get%.FU:Code%Dao().load((%pkClassCode%)FacesUtils.getHttpSession(false).getAttribute("%Code
%Edit_INPUTPARAMETER"));
        } catch (Exception e) {
            FacesUtils.addErrorMessage(e.getCause().getMessage());
        }
        return %.FL:Code%;
    }
```

```
    public String invoke_update()
    {
        try
        {
            get%.FU:Code%Dao().update(this.%.FL:Code%);
            return ACTION_UPDATE;
        }
        catch (Exception e)
        {
            FacesUtils.addErrorMessage(e.getCause().getMessage());
            return STATUS_FAIL;
        }
    }

    public String invoke_cancel()
    {
  FacesUtils.getHttpSession(false).setAttribute("%Code%Edit_INPUTPARAMETER", null);
        return ACTION_CANCEL;
    }\n
}
```

计算机根据转换规则和重用“修改学生”控制设计模型，首先通过查找“STUDENTEdit”状态元素的“Classifier”属性值“学生”并找到该状态元素对应的对象模型中的“学生”类，然后将对象模型中学生类的“Code”属性值取出替换 GTL 模板中%Code%等占位符，从而完成表示层修改学生 PageBean 控制代码的生成。

4.4 本章小节

本章以程序设计自主学习系统为例，首先介绍了在 PowerDesigner 中标记类图表示的领域模型后得到具有语言平台信息的对象模型的过程，然后介绍了在对象模型到状态模型的转换中标记对象模型，使得转换得到包含 JSF 框架和设计模式信息的状态模型的过程。最后介绍了通过状态模型生成 JSF 表示层代码的过程。通过将 Web 代码生成方法应用于基于 MDA 思想开发的程序自主学习系统上，验证了本文提出的 Web 代码生成方法的可行性。

第 5 章 总结与展望

5.1 本文工作总结

本文介绍了基于 MDA 思想的软件开发过程涉及的基础知识以及元建模机制和统一建模语言等相关标准。根据 MDA 模型表示和转换思想，给出了本文的 Web 代码生成方法思路。将领域模型作为代码生成的入口，使用 Java 语言名称标记领域模型，得到具有 Java 语言信息的对象模型。使用 JSF 框架名称标记对象模型，使得对象模型转换得到的状态模型中包含 JSF 框架和设计模式信息。根据状态图到 JSF 表示层代码的映射规则，将状态模型生成表示层代码。为解决建模过程中存在“如何选择实现重用的方式”和“转换中如何加入设计模式”两个关键问题，研究了建模人员建模和建模工具设计人员设计建模工具两个过程，并给出了在实现重用上继承和实例化的区别。然后结合继承和实例化两种方式研究了在模型转换中加入设计模式信息的方法。通过研究 PowerDesigner 中用于实现模型转换的 VBScript 脚本语言和用于代码生成的 GTL 等技术，将 Web 代码生成方法应用于基于 MDA 思想开发的程序设计自主学习系统中，并在 PowerDesigner 中进行实现，从而验证了本文提出的 Web 代码生成方法的可行性。

本文的研究结论为：

1. 提出了首先创建类图表示的领域模型，然后标记领域模型得到具有编程语言信息的对象模型，其次标记对象模型得到具有表示层框架和设计模式信息的状态模型，接着根据类图到状态图的转换规则将对象模型转换到状态模型，最后根据状态图到表示层代码的转换规则将状态模型生成表示层代码的 Web 代码生成方法。

2. 在 MDA 的软件开发过程中，当设计建模工具时，开发人员作为建模工具设计人员，采用继承方式扩展元模型并实现对象重用；当需要使用建模工具建模时，开发人员作为建模人员，采用实例化方式完成建模任务并实现对象重用。

3. 根据实例化和继承两种实现重用的方式，分别给出在基于 MDA 的模型转换中加入设计模式的方法。

4. 将提出的基于模型驱动的 Web 代码生成方法应用于程序设计自主学习系统场景中，验证了该方法的可行性。

5.2 后续研究工作

本文讨论了实例化和继承两种实现重用的方式以及使用两种方式将设计模式信息扩展到元模型中的方法。该方法虽然支持模型可视化的创建和扩展，但在模型的形式化分析与验证上缺少相应的支持，无法保证通过实例化和继承扩展得到的模型的可靠性。另外，本文主要研究了 Web 表示层代码的自动生成。基于三层架构的思想，业务逻辑层和数据访问层的代码自动生成也具有实际的研究价值。基于以上原因，本文后续的研究工作如下：

1. 研究一种模型形式化方法，为软件开发者提供一种用于验证模型可靠性的软件模型形式化验证机制。

2. 继续研究模型与模型的转换规则，在完善 Web 表示层代码自动生成的同时，进一步研究 Web 业务逻辑层和数据访问层代码的自动生成。

参考文献

[1] 尹彦均. Web 应用代码自动生成平台中代码生成系统的研究与实现[D]. 北京: 北京工业大学, 2007.

[2] 郑斌. 基于 ASP.NET Web 应用的代码生成技术的研究与实现[D]. 长沙: 中南大学, 2011.

[3] Mukhtar M A O, Mohd F B H, Bin Jaafar J, et al. WSDMDA: an enhanced Model Driven Web engineering methodology[C]// Control System, Computing and Engineering (ICCSCE), 2014 IEEE International Conference on. Batu Ferringhi: IEEE Press, 2014: 484-489.

[4] OMG. ormsc/08-09-16. Model Driven Architecture (MDA) [S]. Needham: OMG, 2008-09.

[5] Sacevski I, Veseli J. Introduction to Model Driven Architecture(MDA)[D]. Salzburg: Department of Computer Science University of Salzburg, 2007.

[6] Ciccozzi F, Malavolta I, Selic B. Execution of UML models: a systematic review of research and practice[J]. Software & Systems Modeling, 2018(3): 1-48.

[7] Favre L, Duarte D. Formal MOF metamodeling and tool support[C]// International Conference on Model-Driven Engineering and Software Development. Rome: IEEE Press, 2017: 99-110.

[8] Zhu Z, Lei Y, Zhu Y, et al. Cognitive behaviors modeling using UML Profile: design and experience[J]. IEEE Access, 2017(99): 21694-21708.

[9] 杨凌云. 基于列控系统的扩展 UML 模型设计及故障树求解算法[D]. 北京: 北京交通大学, 2015.

[10] Al-Alshuhai A, Siewe F. An extension of UML Activity Diagram to model the behaviour of Context-Aware systems[C]// IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing. Liverpool: IEEE Press, 2015: 431-437.

[11] 胡洁, 王青. 一种软件特征模型扩展和演化分析方法[J]. 软件学报, 2016, 27(5): 1212-1229.

[12] 王永涛, 刘勇. 基于 MDA 的模型转换研究与应用[J]. 计算机工程, 2011, 37(16): 84-85.

[13] 张行知. 模型驱动体系结构研究[J]. 信息通信, 2016(5): 16-17.

[14] 张天, 张岩, 于笑丰, 等. 基于 MDA 的设计模式建模与模型转换[J]. 软件学报, 2008, 19(9): 2203-2217.

[15] 曾一, 周吉, 孙政, 等. 支持 MDA 的设计模式建模与模型转换方法研究[J]. 计算机工程与应用, 2012, 48(1): 76-80.

[16] 马丽, 毋国庆, 黄勃, 等. BDL 模型到 UML 状态图的可视化方法研究[J]. 计算机科学, 2015, 42(7): 38-43.

[17] Rhazali Y, Hadi Y, Chana I, et al. A model transformation in model driven architecture from business model to web model[J]. Iaeng International Journal of Computer Science, 2018, 45(1): 104-117.

[18] Agustin J L H. Model-driven web applications[C]// Science and Information Conference. London: IEEE Press, 2015: 954-964.

[19] Zouhaier L, Hlaoui Y B, Ayed L J B. Users interfaces adaptation for visually impaired users based on Meta-Model Transformation[C]// Computer Software and Applications Conference. Turin: IEEE Press, 2017: 881-886.

[20] Wan Jiancheng, Lu Xudong, Lu Lei. A model of user interface design and it's code generation[C]// IEEE International Conference on Information Reuse and Integration. Las Vegas: IEEE Press, 2007: 128-133.

[21] 蔡奎, 卢雷, 王帅强, 等. 基于 Web 界面设计模式的复杂行为建模及其代码生成方法[J]. 计算机应用, 2009, 29(4): 1139-1142.

[22] 杨鹤标, 侯仁刚, 田青华. 支持界面自动生成的模型研究[J]. 计算机工程, 2010, 36(3): 79-82.

[23] 王金恒, 王普, 李亚芬. 面向 MDA 的业务逻辑模型到代码的转换方法研究[J]. 计算机技术与发展, 2012, (11): 36-40.

[24] Selimi B，Luma A. Specification of document structure and code generation for Web content management[J]. International Journal of Computer Science Issues, 2015, ISSN (Print): 1694-0814.

[25] Roubi S, Erramdani M, Mbarki S. Modeling and generating graphical user interface for MVC Rich Internet Application using a model driven approach[C]// International

Conference on Information Technology for Organizations Development. Fez: IEEE Press, 2016: 1-6.

[26] Kövesdán G, Lengyel L. Meta3: a code generator framework for domain-specific languages[J]. Software & Systems Modeling, 2018: 1-19.

[27] Czarnecki K, Helsen S. Classification of Model Transformation Approaches[J]. The Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003, 61(2): 217-219.

[28] OMG. ormsc/03-05-01. MDA Guide revision 1.0 [S]. Needham: OMG, 2003-05.

[29] OMG. ormsc/14-06-01. MDA Guide revision 2.0 [S]. Needham: OMG, 2014-06.

[30] Kleppe A, Warmer J, Bast W. MDA explained: the Model Driven Architecture: practice and promise[M]. Upper Saddle River: Addison-Wesley, 2003: xi-xiii.

[31] Misbhauddin M, Alshayeb M. Extending the UML use case metamodel with behavioral information to facilitate model analysis and interchange[J]. Software & Systems Modeling, 2015, 14(2): 813-838.

[32] OMG. formal/02-04-03. Meta Object Facility (MOF) Specification version 1.4[S]. Needham: OMG, 2002-04.

[33] Fernández-Sáez A M, Chaudron M R V, Genero M. An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles[J]. Empirical Software Engineering, 2018:1-65.

[34] Na H S, Choi O, Lim J E. A method for bBuilding domain Ontologies based on the transformation of UML models[C]// International Conference on Software Engineering Research, Management and Applications. Seattle: IEEE Press, 2006: 332-338.

[35] OMG. Formal/11-08-05. Unified Modeling Language (UML) Infrastructure specification version 2.4.1[S]. Needham: OMG, 2011-08.

[36] OMG. formal/11-08-06. Unified Modeling Language (UML) Superstructure specification version 2.4.1[S]. Needham: OMG, 2011-08.

[37] Amrani M, Dingel J, Lambers L, et al. Model transformation intents and their properties[J]. Software & Systems Modeling, 2016, 15(3): 647-684.

[38] Craig Larman. UML 和模式应用[M]. 李洋, 郑奕, 译. 北京: 机械工业出版社, 2006: 201-202.

[39] Mak J K H, Choy C S T, Lun D P K. Precise Modeling of Design Patterns in UML[J]. 2004: 252-261.

[40] Hardyanto W, Purwinarko A, Sujito F, et al. Applying an MVC framework for the system development life cycle with Waterfall Model extended[C]// Journal of Physics Conference Series. Semarang: IOP, 2017: 012007.

[41] 谭云杰. 大象-Thinking in UML(第二版)[M]. 北京: 中国水利水电出版社, 2012: 152-153.

[42] OMG. formal/2016-06-03. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification [S]. Needham: OMG, 2016-06.

[43] Possatto M A. Automatically propagating changes from reference implementations to code generation templates[J]. Information and Software Technology, 2015, 67(C): 65-78.

[44] Petković D, Jovic S, Golubovic Z. Object-oriented modeling approach of universal education software[J]. Computer Applications in Engineering Education, 2017: 1-16.

[45] Khelifi N Y, Śmiałek M, Mekki R. Generating database access code from domain models[C]// Computer Science and Information Systems. Lodz: IEEE Press, 2015: 991-996.

[46] Kusel A, Schönböck J, Wimmer M, et al. Reuse in model-to-model transformation languages: are we there yet?[J]. Software & Systems Modeling, 2015, 14(2): 537-572.

[47] 韩德帅, 杨启亮, 邢建春. 一种软件自适应 UML 建模及其形式化验证方法[J]. 软件学报, 2015, 26(4): 730-746.

[48] Prokhorenko V, Choo K K R, Ashman H. Web application protection techniques: A taxonomy[J]. Journal of Network & Computer Applications, 2016, 60: 95-112.

致谢

光阴似箭，日月如梭。转眼三年的研究生学习生涯即将画上句号，迎接自己的将是一个全新的世界。此刻的我不仅对全新的社会生活充满好奇，而且对即将离开的校园生活充满了回忆。

在本文即将完成之际，不得不感谢的是张力生教授。张老师是一位拥有丰富工程经验和优秀的逻辑思维的好老师。在指导研究生的过程中，他总是为我们灌输“先学做人，再学做事”的思想，强调做好人比搞好学术更为重要。张老师还强调“诚信和责任”的重要性，教导我们做人要有诚信，做事要有责任感，做一个对社会有用的人而不是做一个仅仅具有一本毕业证的无能力者。另外，张老师还强调搞学术要具有“工匠精神”，要一步一个脚印，踏踏实实做好做实每一件小事，小事做踏实了，大事才有保障。在大四下学期，张老师就热心地指导了我的本科毕业设计，该过程使得我真正体会到了什么是软件开发过程和软件建模思想，从此对软件分析和设计产生了浓厚的兴趣。读研后，张老师安排我首先采用读程序的方法读程序，其目的是为编码打下着实的基础。然后组织我和实验室的同学一起参与了重庆市“领导干部考试系统”的项目开发。通过参与项目，我将读程序过程中学到的知识应用到了编码过程，将理论与实践进行了结合。为解决一个编码难题，张老师专门腾出宝贵的时间为我进行了细心的辅导，最终解决了难题。我在软件项目开发中，根据“做中学”的思想接触了 MDA 思想，知道模型和模型转换等基础知识，为我研究生阶段的研究工作积累了宝贵的经验。

其次，感谢我的同门们。感谢研究同一方向的年欢、刘本静师姐为实验室留下的关于软件建模和模型转换的宝贵资料。感谢王晗师姐在研究学习的方法和成果上做出的榜样。感谢杨小刚师兄在我垂头丧气时为我加油打气，让我重新静下心来投入研究工作。感谢同级的杨卓卿、周宇等在软件开发和平日研究工作中日日夜夜的陪伴。感谢魏天、宛佳明师弟和张悦、钟钰璐师妹在我研究工作中提供的帮助。

最后，对参加论文评阅和答辩工作的老师们致以衷心的感谢！

攻读硕士学位期间从事的科研工作及取得的成果

发表及完成的论文：

[1] **罗异**. 模型表示中继承和实例化方法的研究与应用[J]. 现代计算机. 2018(已录用，待发表).

软件著作权：

[1] 张力生，刘本静，**罗异**，周宇[Z]，考试资源管理系统V1.0, 2016-07-22.（已授权）