软件体系结构
(Software Architecture)

四、模型驱动的体系结构
MDA (Model Driven Architecture)

北京大学软件工程国家工程研究中心
NATIONAL ENGINEERING RESEARCH CENTER FOR
SOFTWARE ENGINEERING OF PEKING UNIVERSITY

- 从西西弗斯说起——

- 现代版西西弗斯
  ——项目进展

内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

MDA简介

- Model Driven Architecture
  - Model？客观事物的抽象表示
  - Model-Driven？使用模型完成软件的分析、设计、构建、部署、维护等各开发活动
  - Architecture？构成系统的部件、连接件及其约束的规约
  - MDA起源于分离系统规约和平台实现的思想
  - MDA的主要目标：

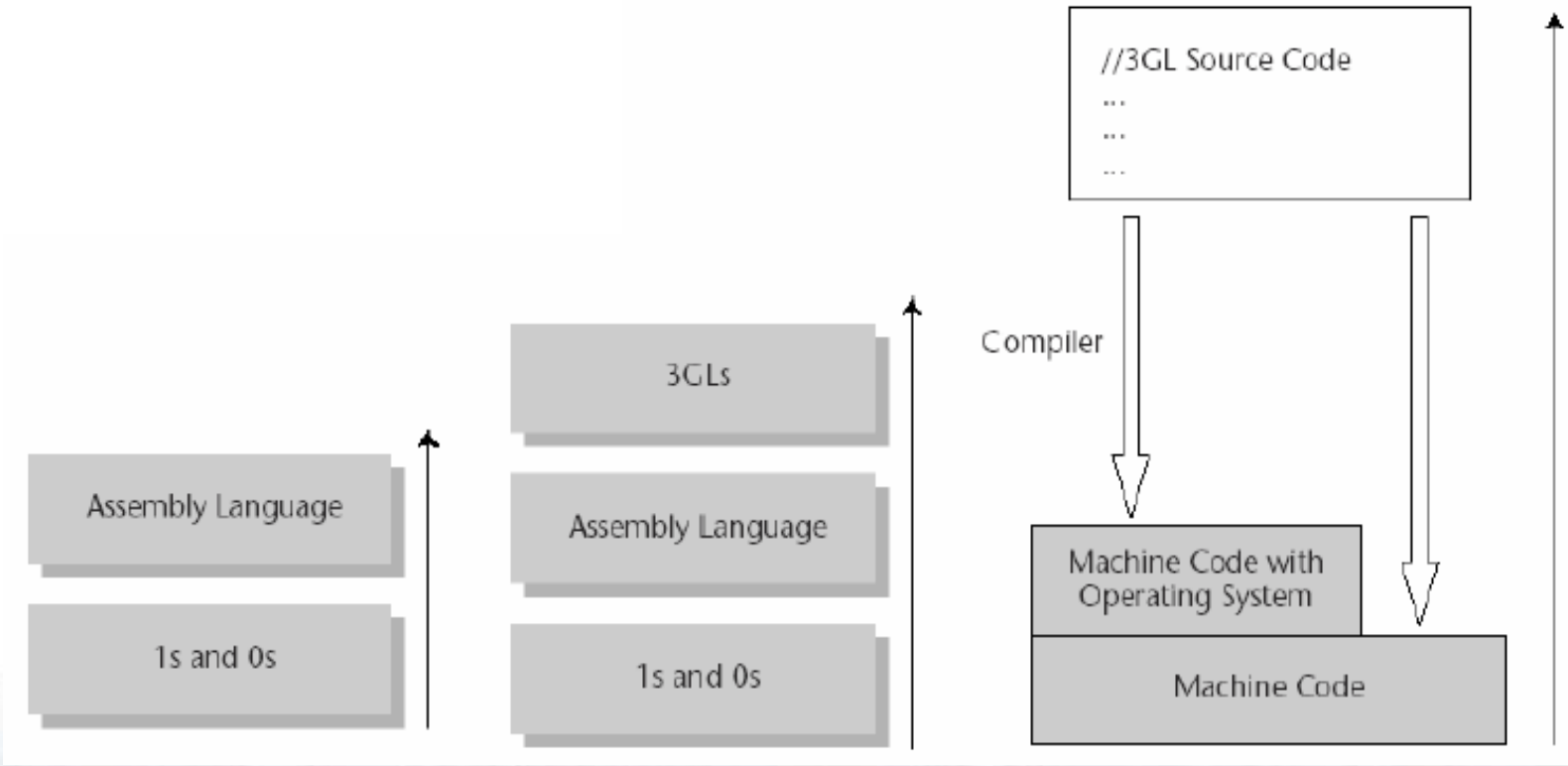
    portability, interoperability, reusability

MDA简介

- MAD产生背景
  - OMG 标准化OMA CORBA
  - OMG 标准化对象建模技术UML, 1995
  - OMG 采纳MDA作为第二个软件框架, 2001
    - MDA不是实现分布式系统的框架，而是在软件开发中使用模型的指导方法
    - MDA是从软件工艺迈向软件工程化的一步

MDA简介

- 以机器为中心的计算
- 以应用为中心的计算



不断提高的抽象层次

MDA简介

- 以企业为中心的计算
  - 基于构件的开发（CBD）
  - 体系结构风格、设计模式
  - 分布式计算
  - 中间件：提升平台抽象层次、提升变成抽象层次
  - 说明性规约（数据库、WYSIWYG）
  - 企业体系结构和关注点分离（大型机、C/S、3-tier、n-tier）
  - 企业应用集成（EAI）：集成遗产系统
  - 契约式设计：建立可靠软件系统的方法
  - 4GL（数据库访问、GUI生成）

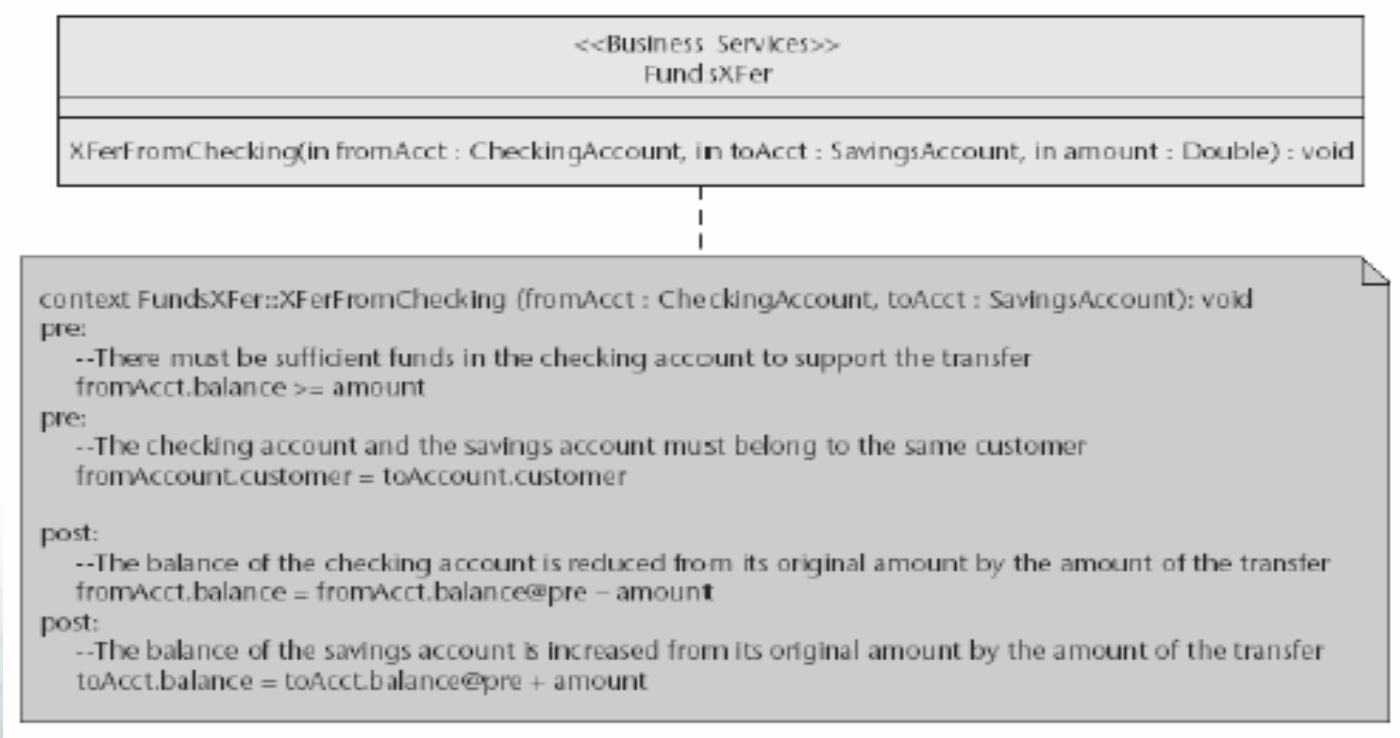
MDA简介

- 以企业为中心的计算面临的压力
  - 应用的复杂性（B2Bi）
  - 生产成本的压力（需求的变动）
  - 质量的压力（文档、形式化）
  - 软件生命期的压力（平台的易变性）

COMPANY	THEN	NOW
Sun	Java language	J2SE, J2EE
W3C	XML DTD	XML Schema, WSDL, etc.
OMG	CORBA	CORBA Component Model
Microsoft	MTS	COM+/.NET
Sun and OMG	RMI vs. IIOP	RMI over IIOP

MDA简介

- 将以模型为中心的思想引入EAI、B2Bi
  - 模型作为开发制品，而非设计制品
  - 模型富含语法抽象和语义抽象
  - 结合Web Service的应用
  - 定义抽象的业务服务 =〉将业务服务模型映射到WSDL



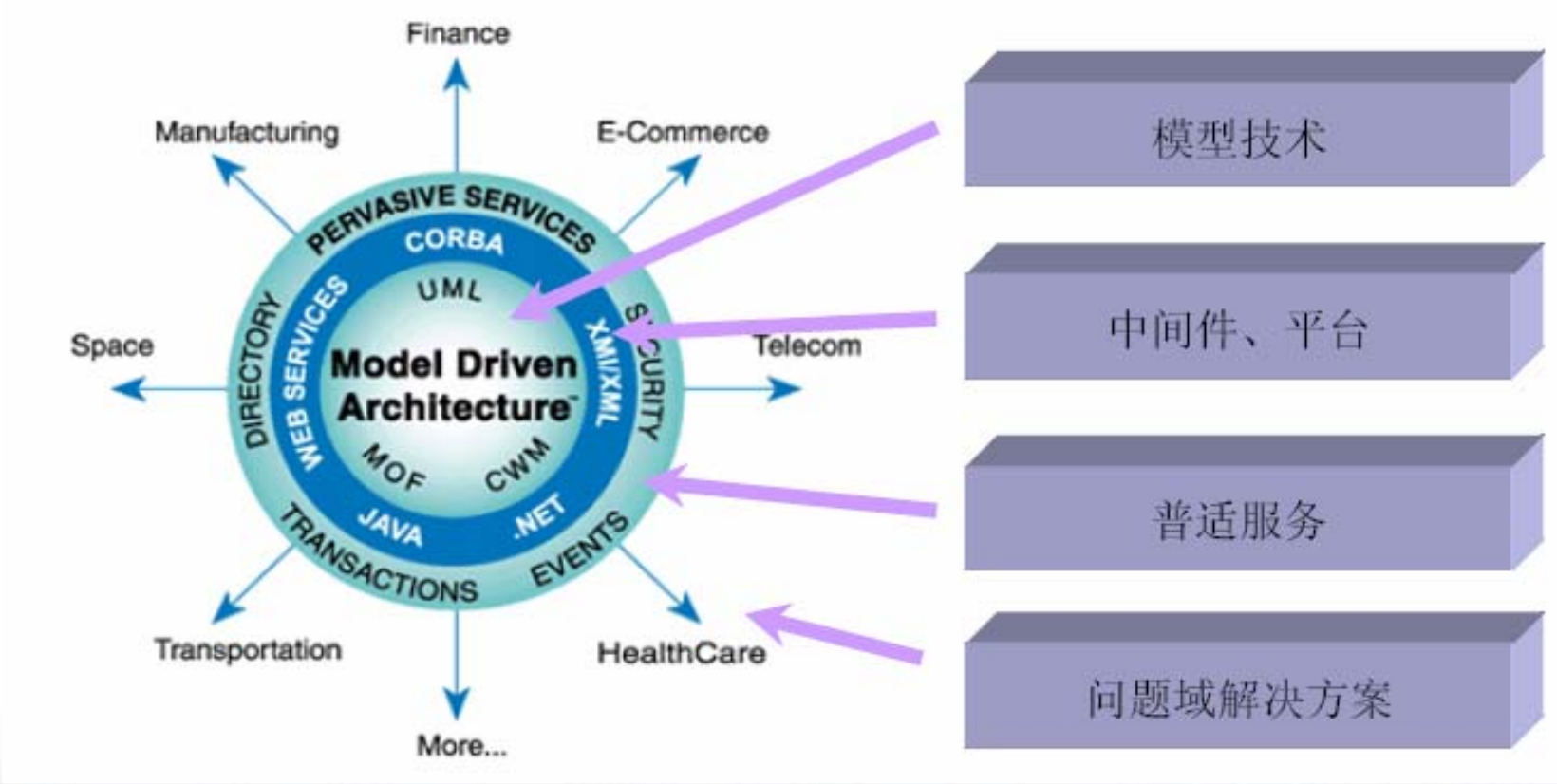
MDA简介

- OMG提出解决上述问题的有效途径—MDA
  - 模型的角色
  - 软件开发中的难点——分离关注点

  - 模型是问题域的抽象，模型中的元素对应问题域中的概念
  - 模型中使用客户熟悉的概念/术语描述问题，避免与计算机实现相关的细节问题
  - 沟通业务和技术的有效途径
  - 模型有不同的抽象级别，可以从高层抽象向底层模型进行映射

MDA简介

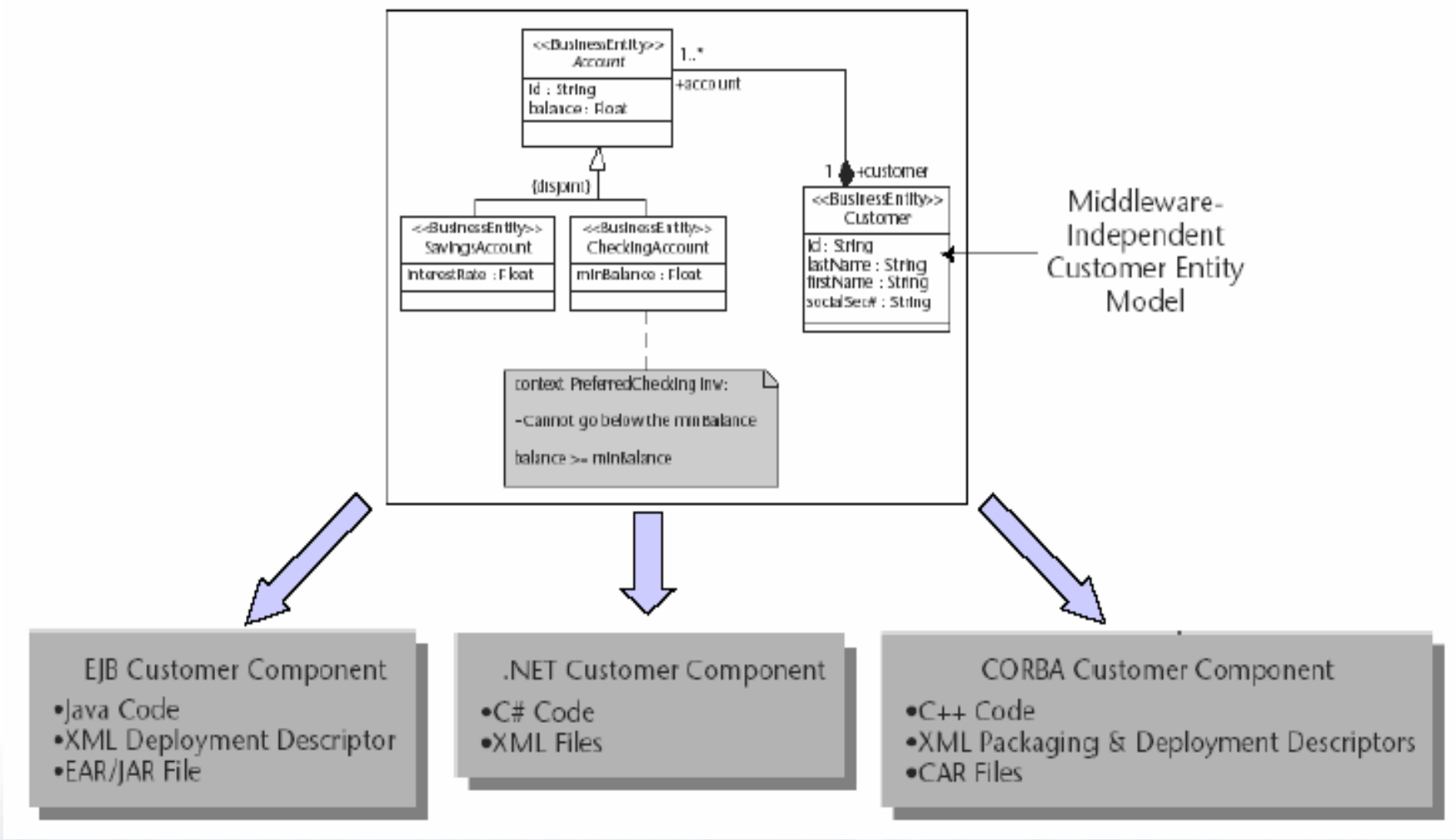
- MDA的体系结构图



MDA简介

- 核心部分：OMG的建模技术——UML、CWM、MOF等，可通过UML profiles进行扩充，独立于具体平台
- 平台层：将独立于平台的模型映射到各平台，使用映射标准和工具自动或半自动的执行
  - 平台相关的模型一般也采用UML表达，通过UML方言表达特定平台的元素
  - 平台无关模型中的语法和语义信息被映射到平台相关的模型中
- 实现层：平台相关的模型被映射产生代码，使用工具自动或半自动的生成
  - 平台相关模型中包含的语义信息越丰富，代码的生成质量越高
  - 从平台无关模型到平台相关模型再到代码，体现一致性

MDA简介


<<BusinessEntity>>
Account
Id : String
balance : Float
1..*
+account
1
+customer
{disjoint}
<<BusinessEntity>>
SavingsAccount
interestRate : Float
<<BusinessEntity>>
CheckingAccount
minBalance : Float
<<BusinessEntity>>
Customer
Id : String
lastName : String
firstName : String
socialSec# : String
context PreferredChecking inv:
-Cannot go below the minBalance
balance >= minBalance
Middleware-
Independent
Customer Entity
Model
EJB Customer Component
•Java Code
•XML Deployment Descriptor
•EAR/JAR File
.NET Customer Component
•C# Code
•XML Files
CORBA Customer Component
•C++ Code
•XML Packaging & Deployment Descriptors
•CAR Files

MDA简介

- 互操作性
  - 核心模型是平台无关的，便于向MDA中添加不同平台，只要增加模型到特定平台概念和元素的映射
  - 相应的，平台元素之间的交互逐渐标准化，bridge、gateway、mapping等能够自动或半自动的产生
  - 集成遗产系统，通过包装与核心模型一致的代码层，并且建立包装器的模型，与其他平台的交互可以通过映射机制生成，内部实现可以调用遗产系统完成，最终集成到MDA框架中

MDA简介

- 普适服务
  - 业务系统一般依赖一组关键服务：目录服务、事件处理、数据持久化、事务、安全等
  - 业务系统可能需要平台的非功能性属性：可扩展性、实时、容错性、环境限制等
  - 在平台独立的模型中提供普适服务的支持，映射到平台相关的模型后再利用特定平台提供的服务
  - 在平台独立的模型中，以高层抽象的形式看待服务；在平台相关的模型以及实现中，以本地代码的形式调用平台提供的服务

MDA简介

- 标准化领域模型
  - OMG认识到领域模型的重要性，成立了Domain TaskForce，对特定领域市场所提供的服务和设施进行标准化
  - 包括面向领域的平台独立模型、至少一个平台相关模型、接口描述文件

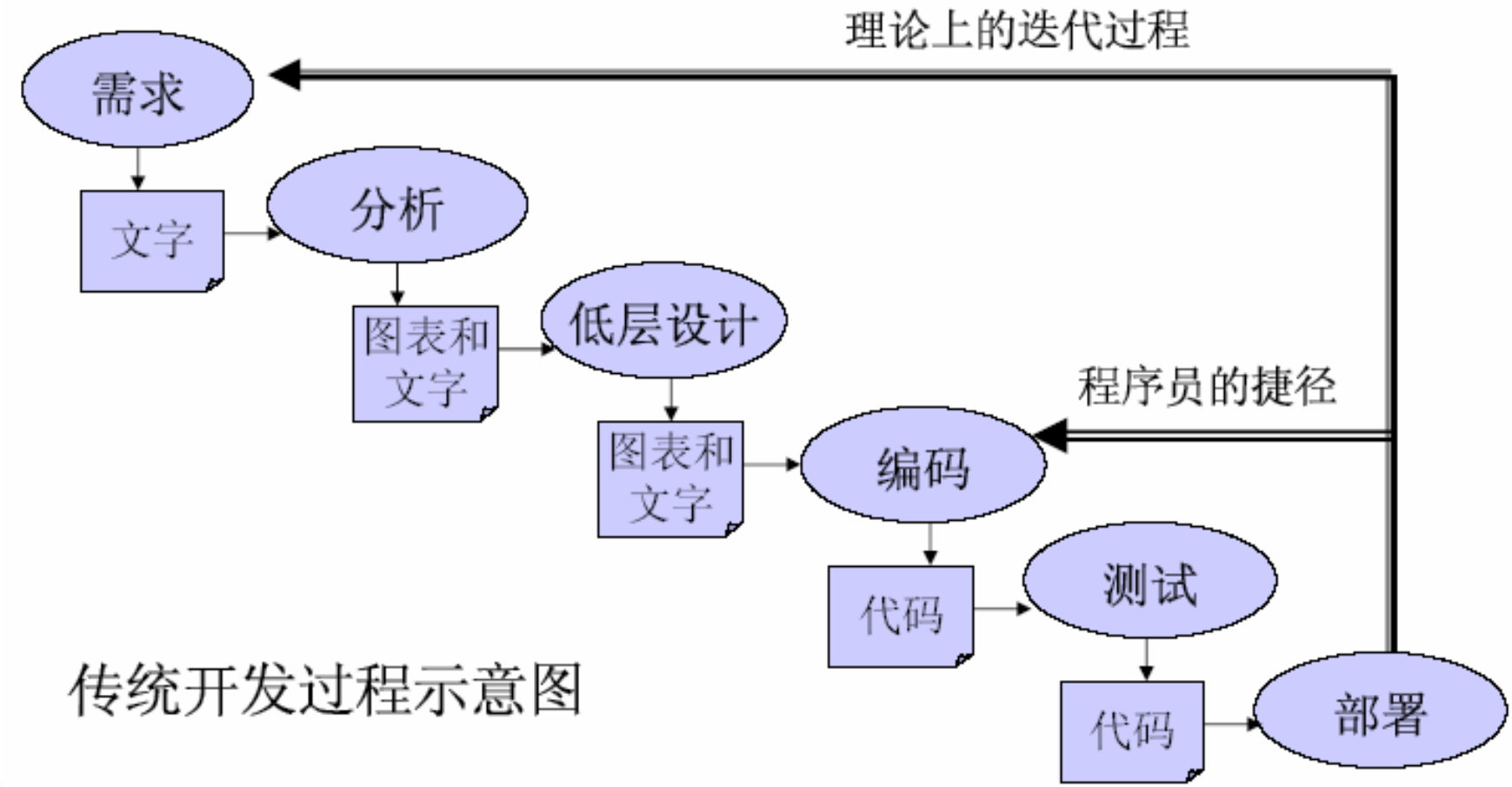
MDA简介

- MDA对企业计算的主要影响
  - 再次提升编程环境的抽象级别
  - 使用形式化模型驱动代码生成器
  - 基于不同规约语言的元数据集成
  - 在生成器中封装了设计模式的知识
  - 形成按需生产构件的方式
  - 使用DBC来驱动断言检测、异常处理和测试框架的生成
  - 形成了“The Global Information Appliance”

内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

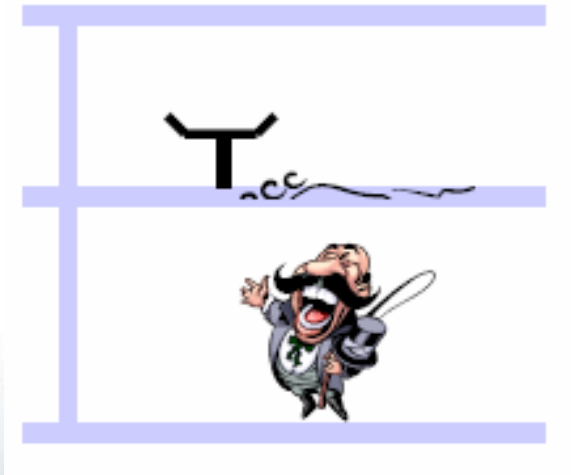
MDA开发过程



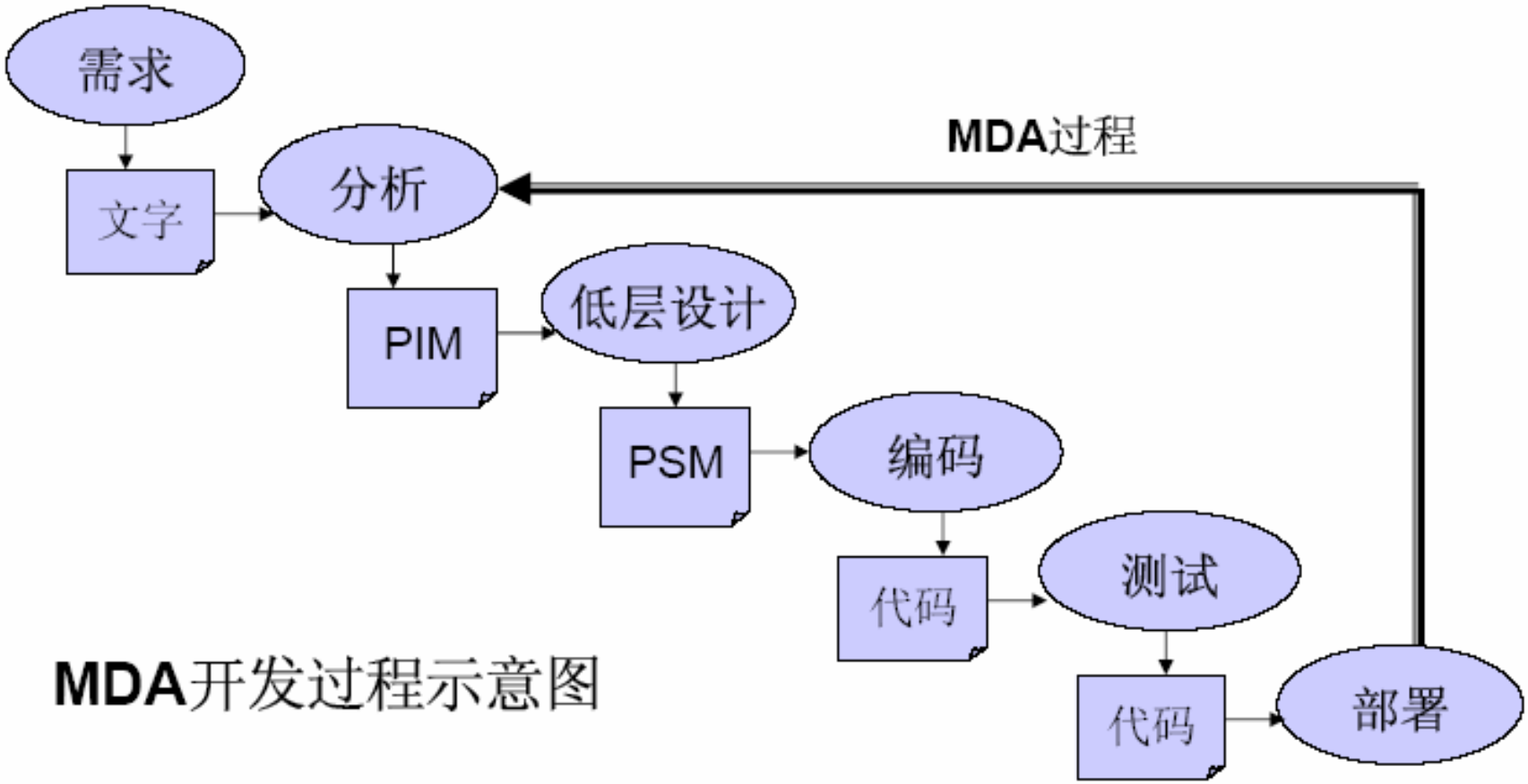
传统开发过程示意图

MDA开发过程

- 传统开发过程带来的问题
  - 生产效率问题：重视代码、维护软件时有问题
  - 可移植性：技术更新换代快
  - 互操作性问题：前端系统和后端系统
  - 维护与文档问题：后期补文档



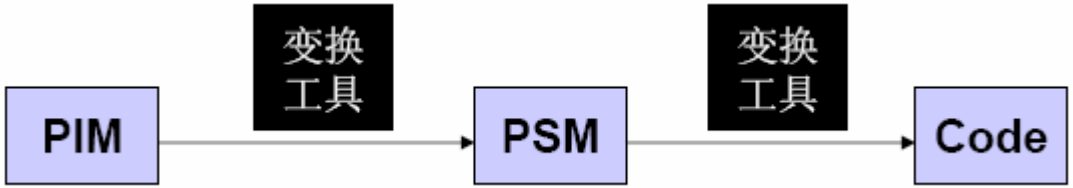
MDA开发过程



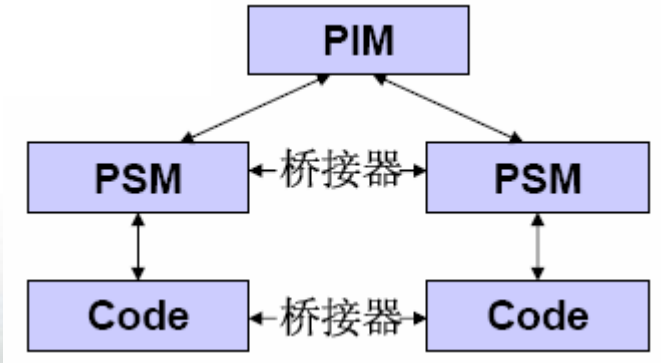
MDA开发过程示意图

MDA开发过程

- 变换自动完成



- 相比传统软件开发过程
  - 生产效率（模型是焦点、PIM开发者轻松）
  - 可移植性（PIM到多个PSM的变换）
  - 互操作性
  - 维护与文档

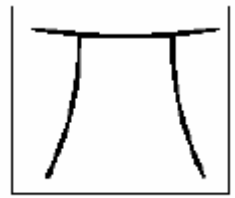


MDA开发过程

- 模型对开发过程的影响：允许定义机器可读的应用及数据模型，用以支持灵活的
  - 实现：结合基础设施，由设计模型映射生成
  - 集成：自动化生成平台之间的桥接器和连接件
  - 维护：可以直接回到系统规约——模型阶段进行维护，然后向下生成系统
  - 测试和模拟：模型根据需求进行验证，在基础设施之上进行测试，模拟系统的行为
- 此前，模型对开发人员来说只短暂存在而MDA中模型之间进行演化，相当于高层次的编译过程

基本概念

- Architecture: 描述了软件开发中可能使用到的若干模型，这些模型如何建立，以及它们之间的关系
- Viewpoint: 为关注系统的特定部分而使用一套概念和规则进行抽象的技术
  - 计算无关的视点：关注系统需求，忽略系统的结构和实现技术
  - 平台无关的视点：关注系统如何操作，忽略特定平台的细节
  - 平台相关的视点：关注系统如何利用平台进行实现
- Platform:通过接口或模式提供一组功能的系统或技术的集合，使得使用平台的应用程序不用关心这些功能如何实现

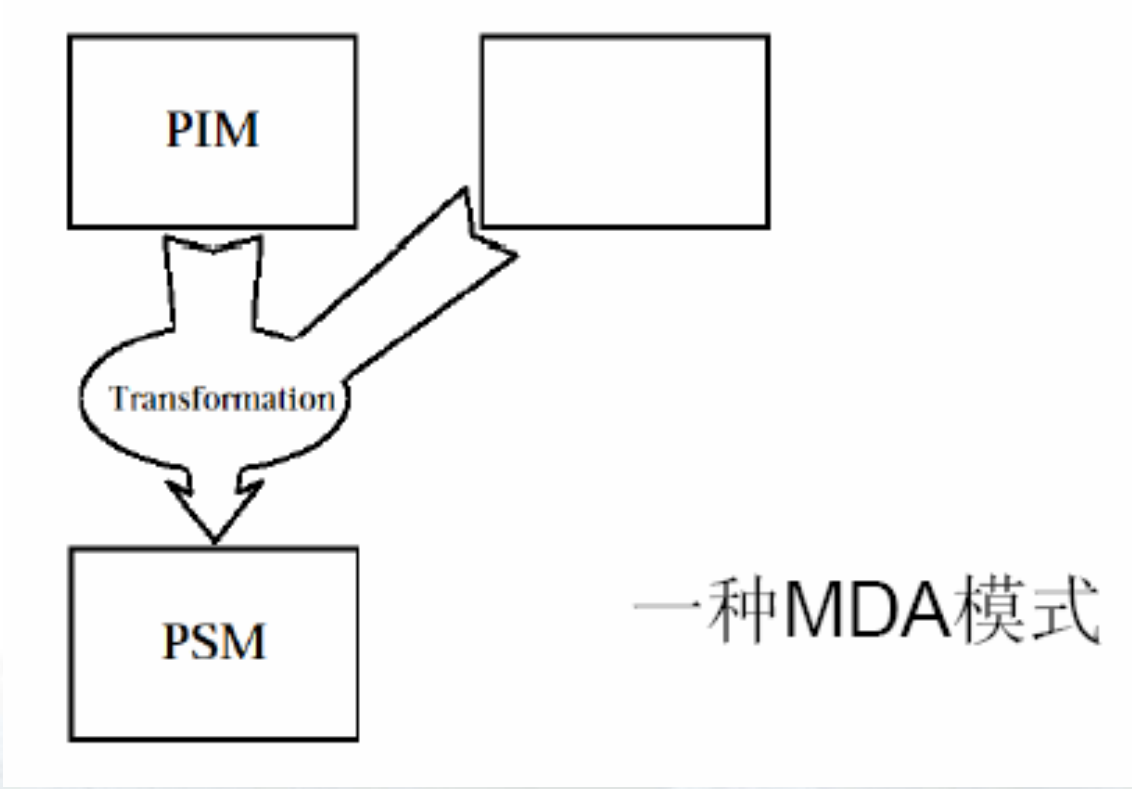


基本概念

- Computation Independent Model: 使用计算无关视点建立的系统模型
  - 被称为领域模型或业务人员的词汇表
  - 其用户是业务人员，不了解用以实现需求的建模技术
  - 在业务专家和技术专家，业务需求和设计、构建方法之间建立桥梁
- Platform Independent Model: 使用平台无关视点建立的系统模型
  - 表达了某种程度的平台无关性，适用于不同的平台上
  - 通用技术是将模型建立在一个技术中立的虚拟机上，该“平台”提供的功能和服务可以在不同平台上实现
- Platform Specific Model: 使用平台相关视点建立的系统模型，将PIM中的规约联系到平台上如何实现

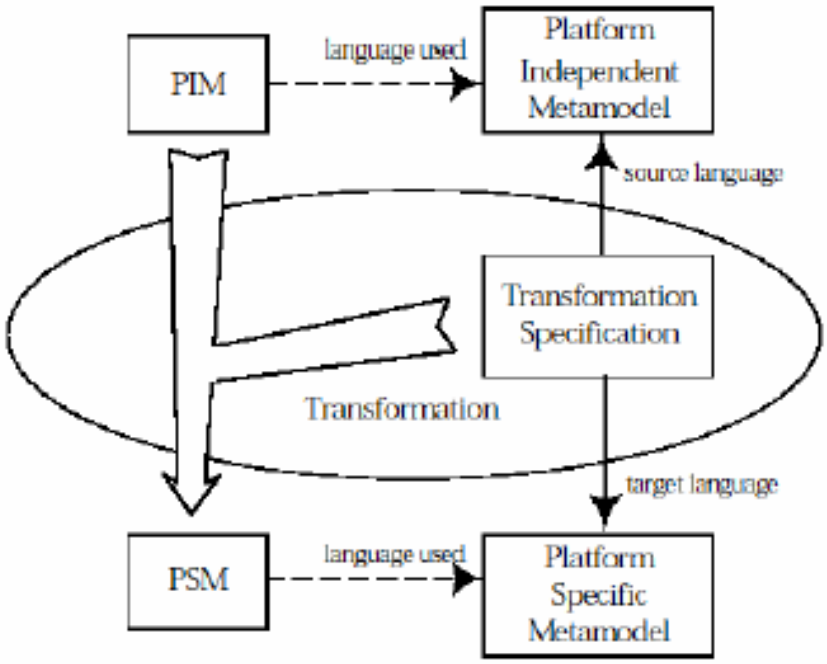
模型变换

- 模型变换：将同一系统的一个模型转换为另一个模型的过程

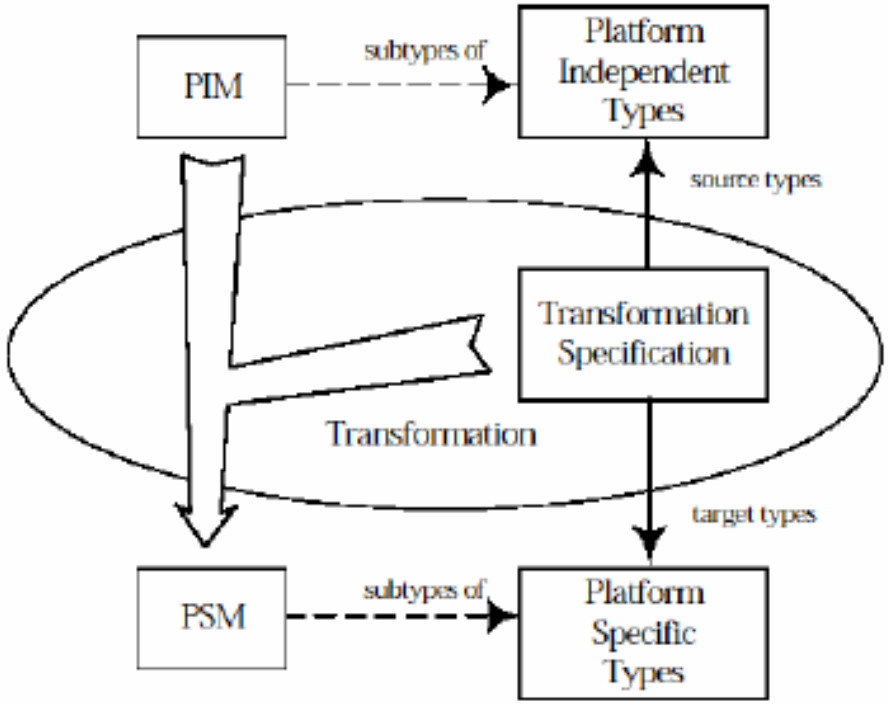


模型变换

- 基于元模型的变换

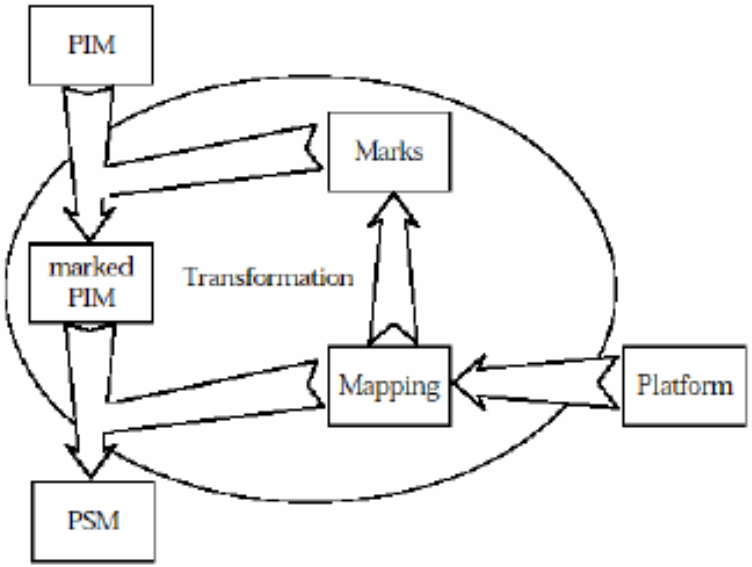


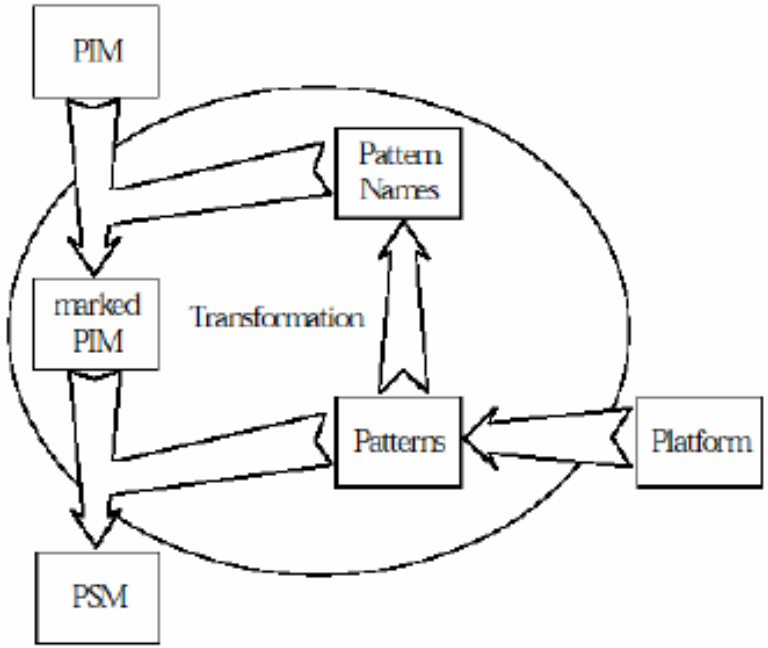
- 基于类型的模型变换



模型变换

- 基于标记的模型变换
- 使用模式的模型变换

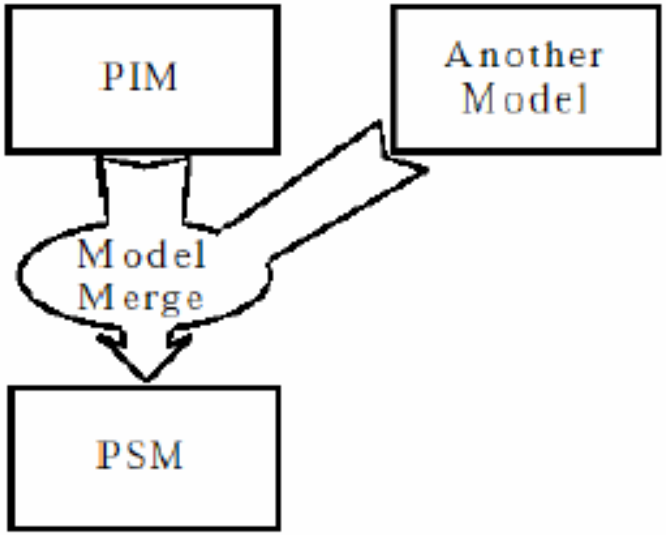




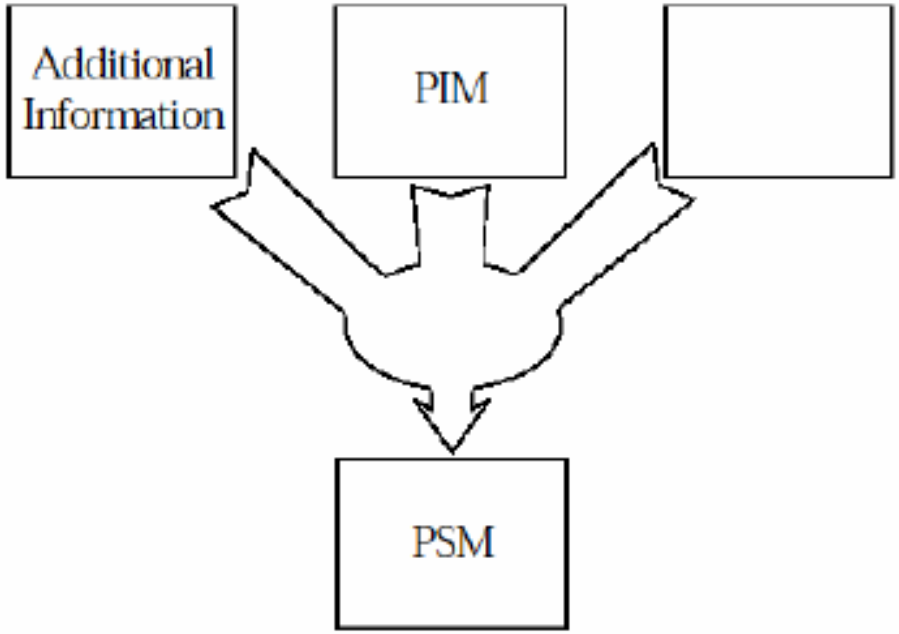
Mark: 表达了PSM中的概念，被应用于PIM中的一个元素，从而指出PIM如何被变换为PSM

模型变换

- 模型合并

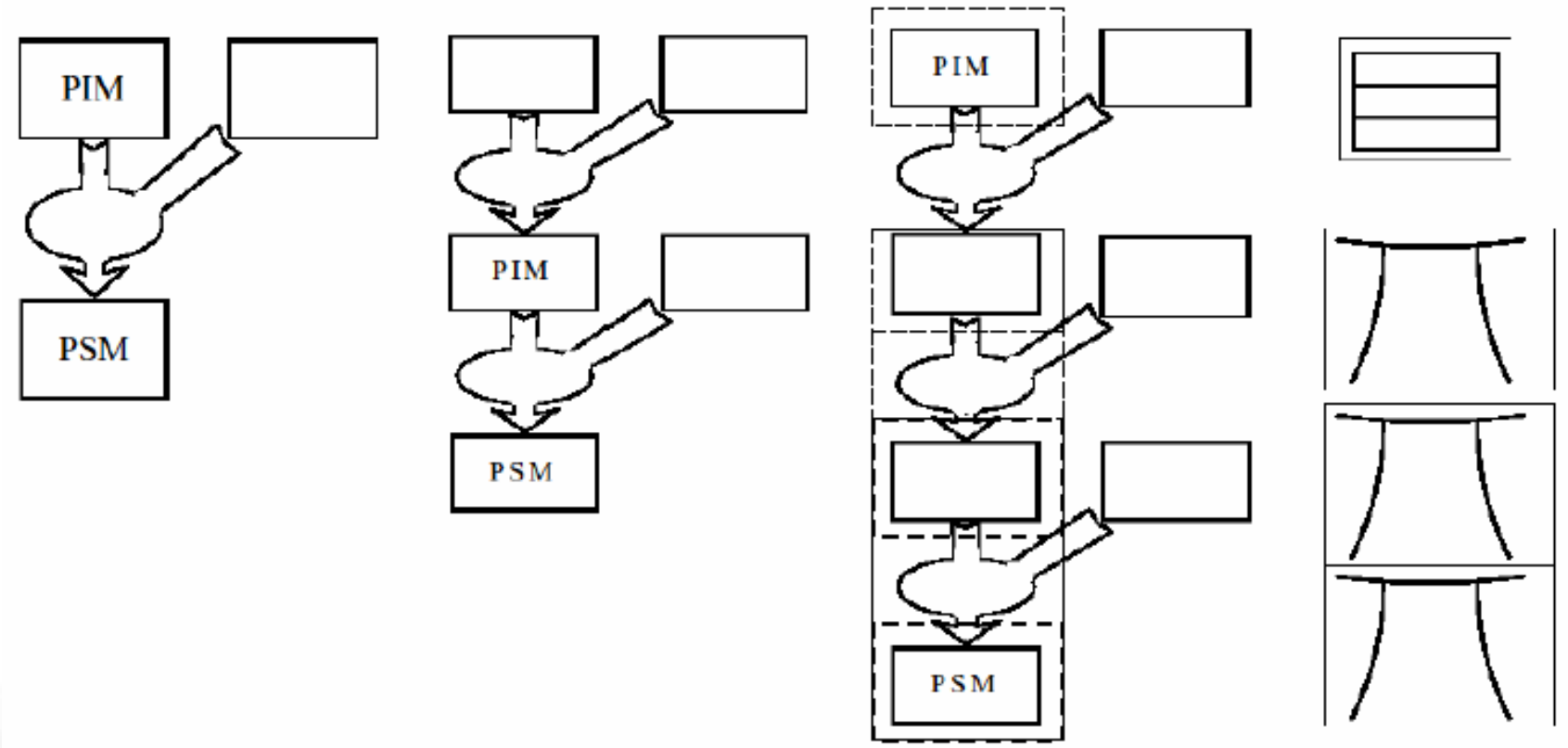


- 使用附加信息的模型变换



模型变换

- 对平台的理解

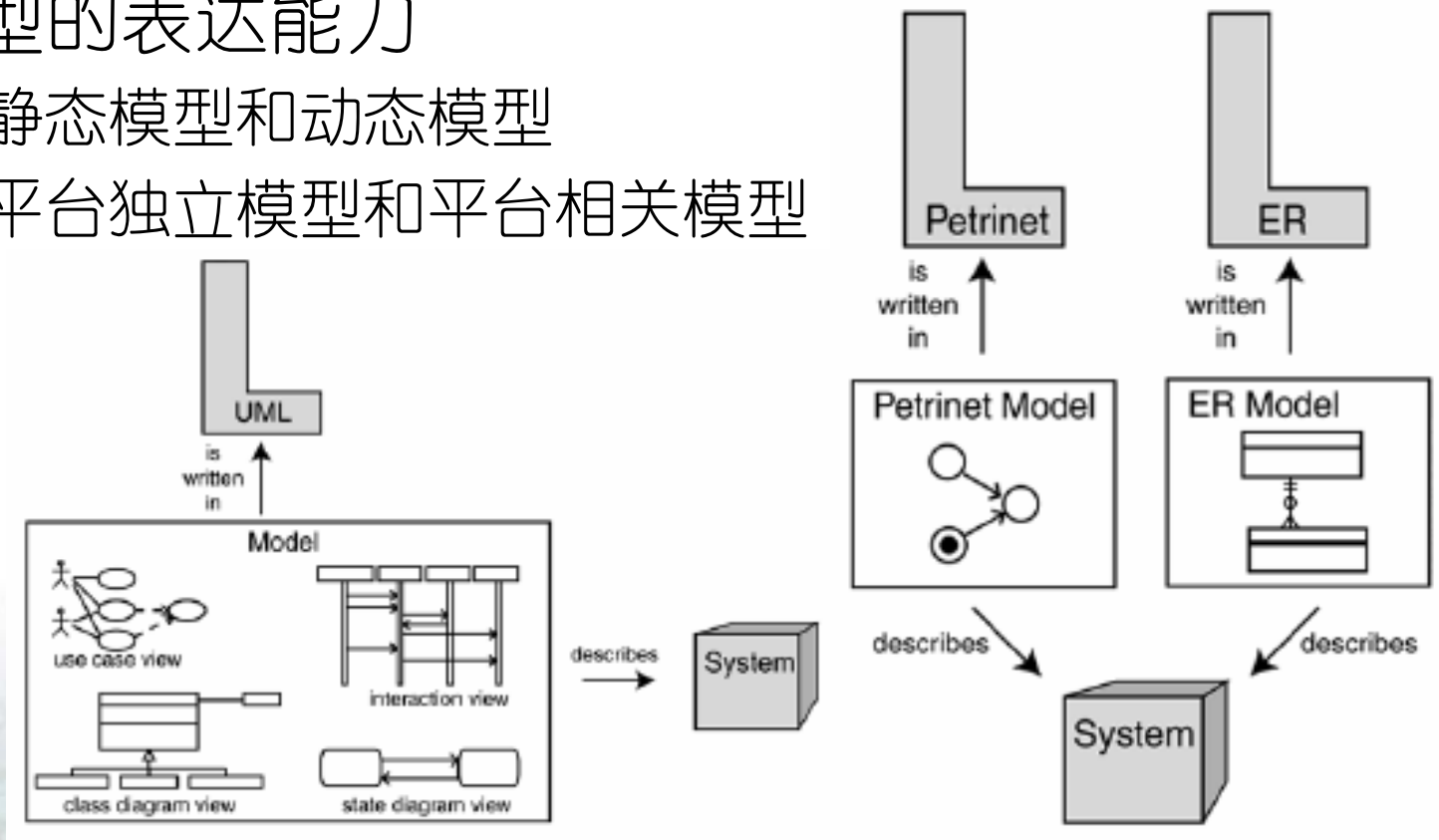


内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

MDA框架

- 模型：以精确定义的语言对系统作出的描述
- 精确定义的语言：具有精确定义的形式(语法)和含义(语义)的语言，以适合计算机自动解释
- 模型的表达能力
  - 静态模型和动态模型
  - 平台独立模型和平台相关模型

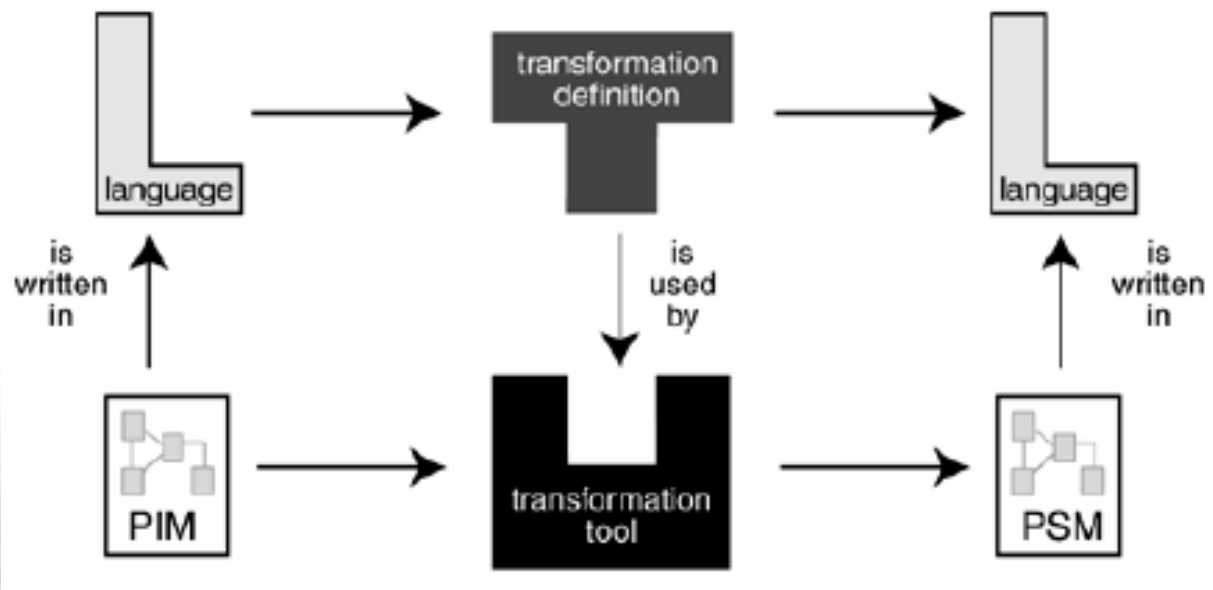


MDA框架

- 变换(Transformation)：模型之间转换的过程
  - 变换定义：一组变换规则，描述用源语言表述的模型
  - 如何变换为用目标语言表述的模型
  - 变换规则：描述源语言中的一个元素如何进行变换为目标语言中的一个元素
- 变换中的输入和输出
  - 可在同一种语言中进行（重构、ER模型的规范化）
  - 可在不同语言中进行（PIM到PSM）

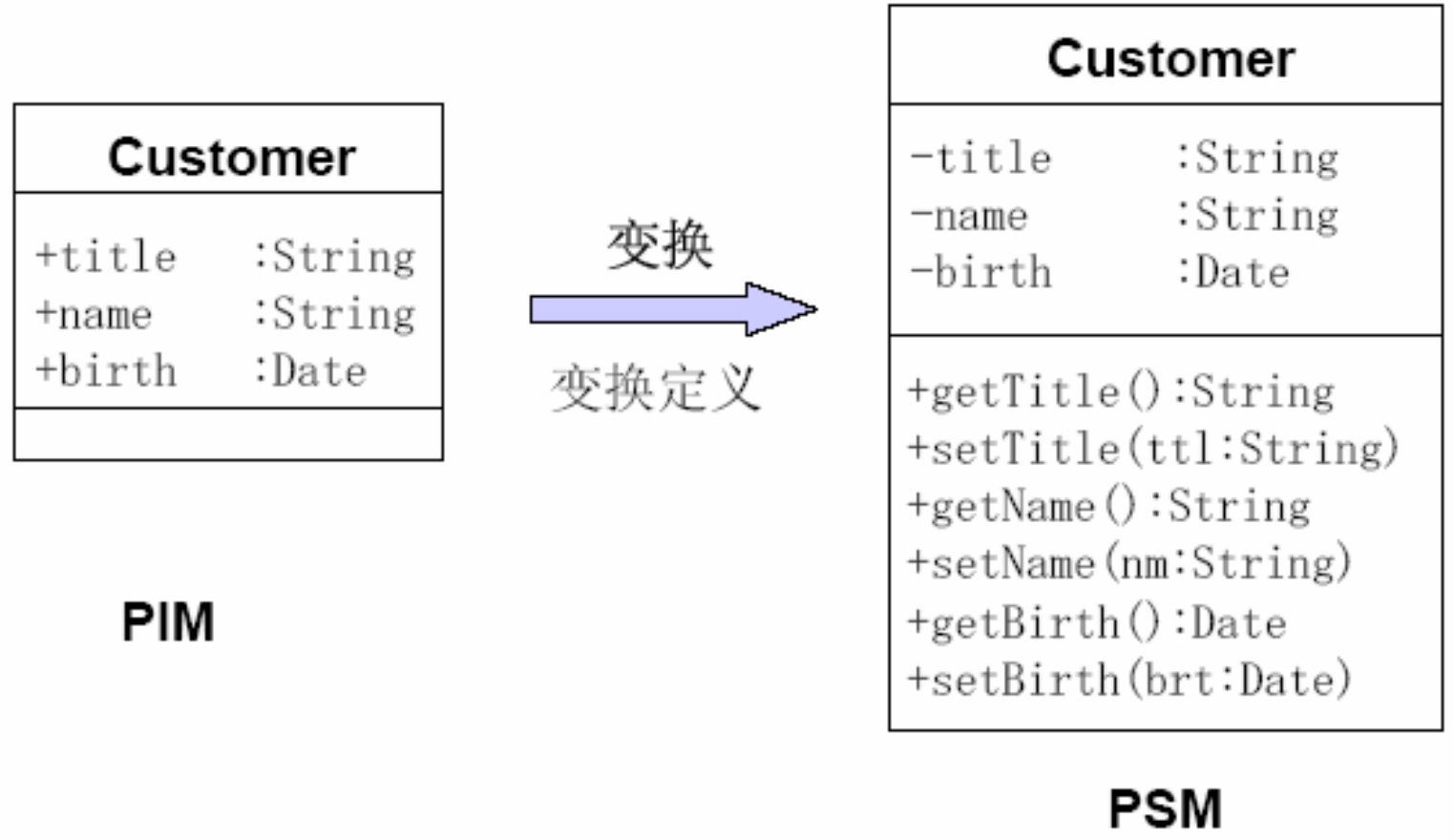
MDA框架

- 基本的MDA框架
  - 模型：PIM PSM
  - 语言：精确定义的
  - 变换定义：变换规则的集合
  - 变换工具：执行源模型到目标模型的变换

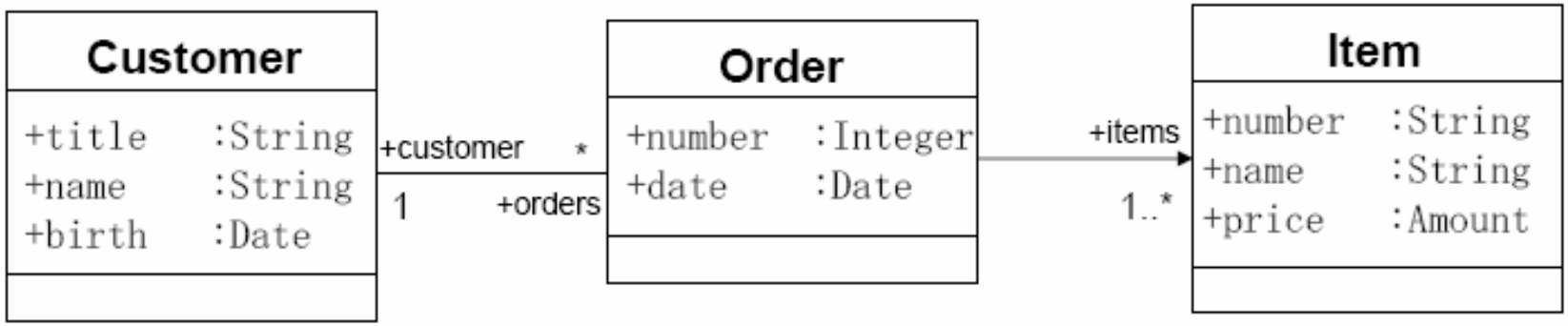


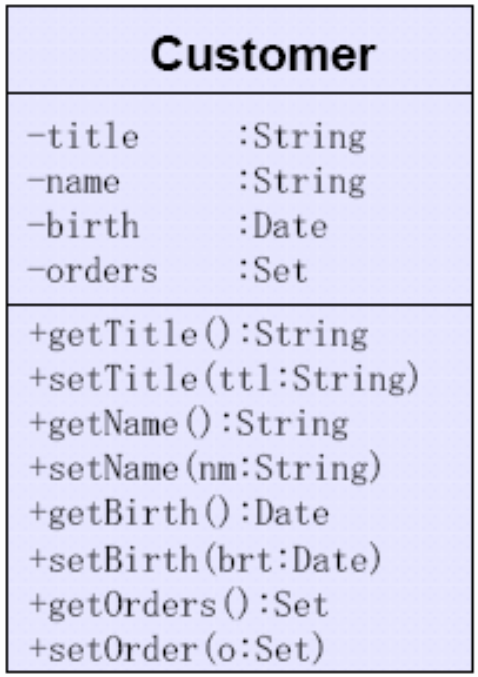
MDA框架

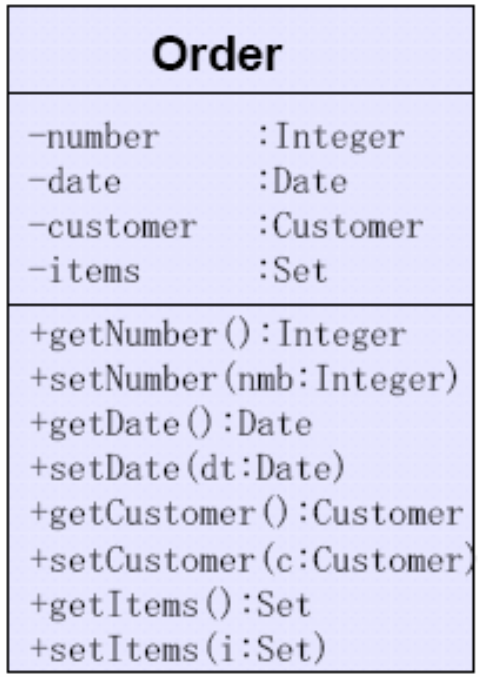
- 小例子

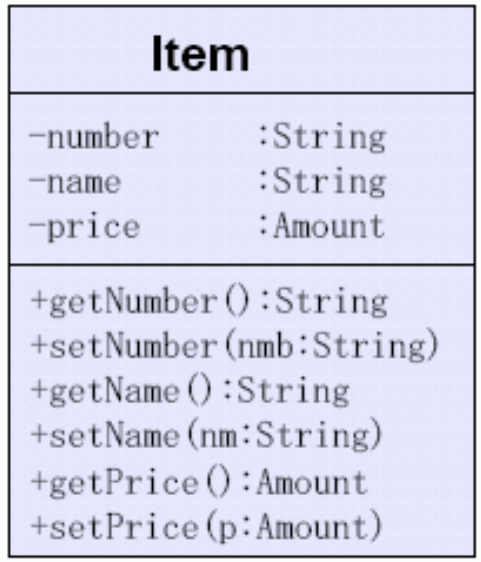

Customer
+title :String
+name :String
+birth :Date
变换
变换定义
PIM
Customer
-title :String
-name :String
-birth :Date
+getTitle():String
+setTitle(ttl:String)
+getName():String
+setName(nm:String)
+getBirth():Date
+setBirth(brt:Date)
PSM

MDA框架


Customer
+title :String
+name :String
+birth :Date
+customer
*
1
+orders
Order
+number :Integer
+date :Date
+items
1..*
Item
+number :String
+name :String
+price :Amount


Customer
-title :String
-name :String
-birth :Date
-orders :Set
+getTitle():String
+setTitle(ttl:String)
+getName():String
+setName(nm:String)
+getBirth():Date
+setBirth(brt:Date)
+getOrders():Set
+setOrder(o:Set)


Order
-number :Integer
-date :Date
-customer :Customer
-items :Set
+getNumber():Integer
+setNumber(nmb:Integer)
+getDate():Date
+setDate(dt:Date)
+getCustomer():Customer
+setCustomer(c:Customer)
+getItems():Set
+setItems(i:Set)


Item
-number :String
-name :String
-price :Amount
+getNumber():String
+setNumber(nmb:String)
+getName():String
+setName(nm:String)
+getPrice():Amount
+setPrice(p:Amount)

MDA框架

- MDA不要求特定的软件开发过程
- 和敏捷软件开发结合：模型被提升为交付软件的一部分
- 和极限编程结合：增量、测试、极限建模
- 和Rational统一过程结合：利用UML、既保持控制又提高
- 效率

敏捷MDA

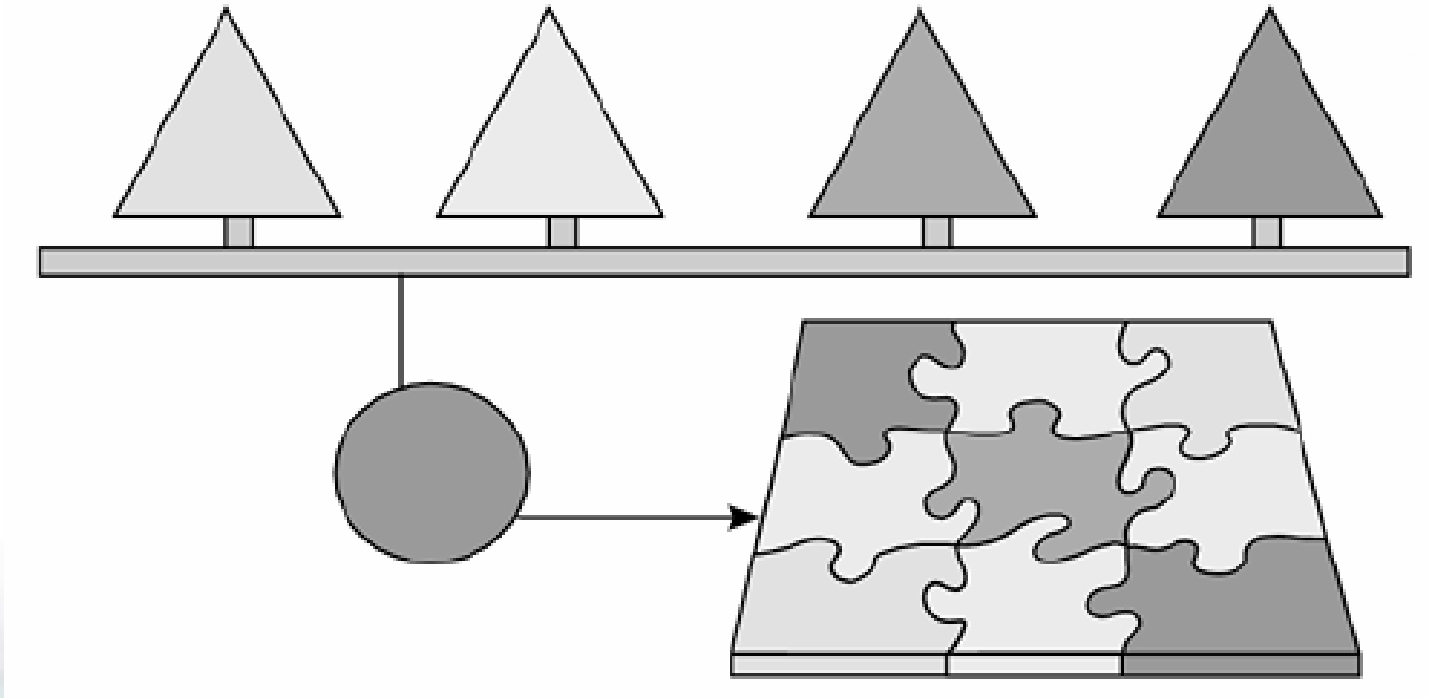
- 基于可执行的模型与代码在执行方面一致，敏捷人士提出了敏捷的MDA方法，即可执行模型以增量或迭代的形式被快速创建、运行、测试和修改
- 敏捷与建模的根本差距之一就是“确认的间隔”——不能执行的模型需要时间和人力转化为代码，模型的准确度、转化的正确性、客户需求的变化等因素使得最终系统存在问题；敏捷方法强调短时间内交付小片的可执行代码，开发结果随时对客户可见，从而缩短了“确认的间隔”
- 敏捷与建模的差别并没有最初看上去那么大，敏捷方法中的很多思想在模型的上下文中同样有效，只要将“代码”替换为“可执行的模型”
- 模型的语言更接近于客户，使其更容易与开发者交互和增量、迭代开发

敏捷MDA

- 2001年，敏捷联盟成立，提出敏捷宣言
  - 个体和交互　胜过　过程和工具
  - 工作软件　　胜过　可理解的文档
  - 与客户协作　胜过　合同谈判
  - 响应变化　　胜过　追随计划
- 模型的三个层次
  - Sketch: 表达系统设计的思想，不必精确和完整，便于理解和交流，应用广泛
  - Blueprint: 描述构造实际系统的文档，将模型作为软件开发过程中的一阶制品，通过手动或自动化手段可以转化为实际系统，要求模型中包含足够精确的信息
  - Program Language: 可以执行，经过编译和执行能满足用户各项需求，就像程序设计语言一样，“可执行的模型”

敏捷MDA

- 设计时互操作总线：模型具有相同的元模型时，模型之间的连接非常简单，新增模型可以直接插在总线上



敏捷MDA

- 敏捷：创建测试用例，编写代码，使用语言编译器编译代码，运行测试用例，不断向用户交付系统的片段
- 敏捷MDA：创建测试用例，编写可执行的模型，使用模型编译器编译模型，运行测试用例，不断向用户交付系统的片段，直接收到客户的反馈
- 可执行的模型 vs. 代码　重量级相当
- 统一过程软件开发 vs. 极限编程

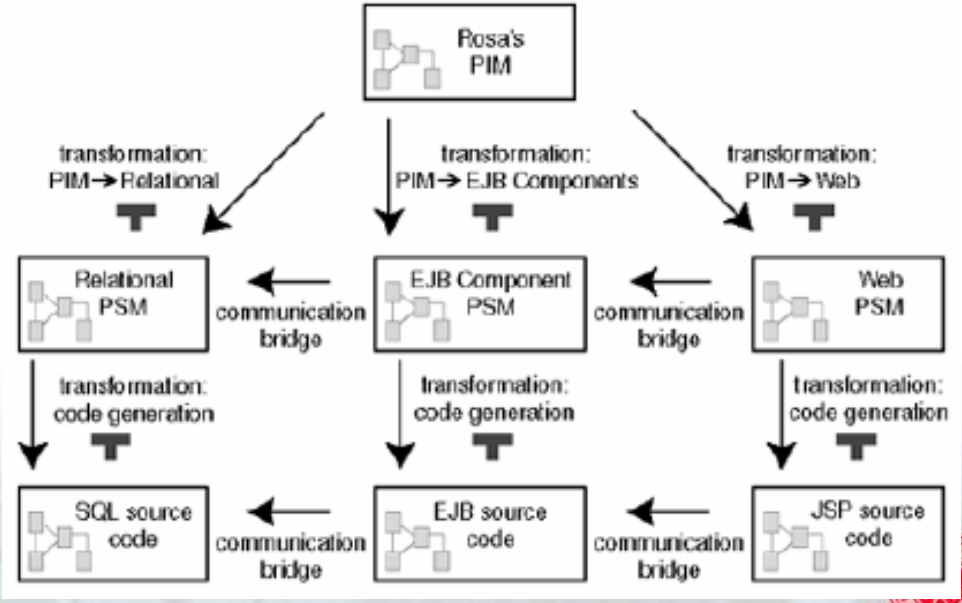
重量级不同

内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

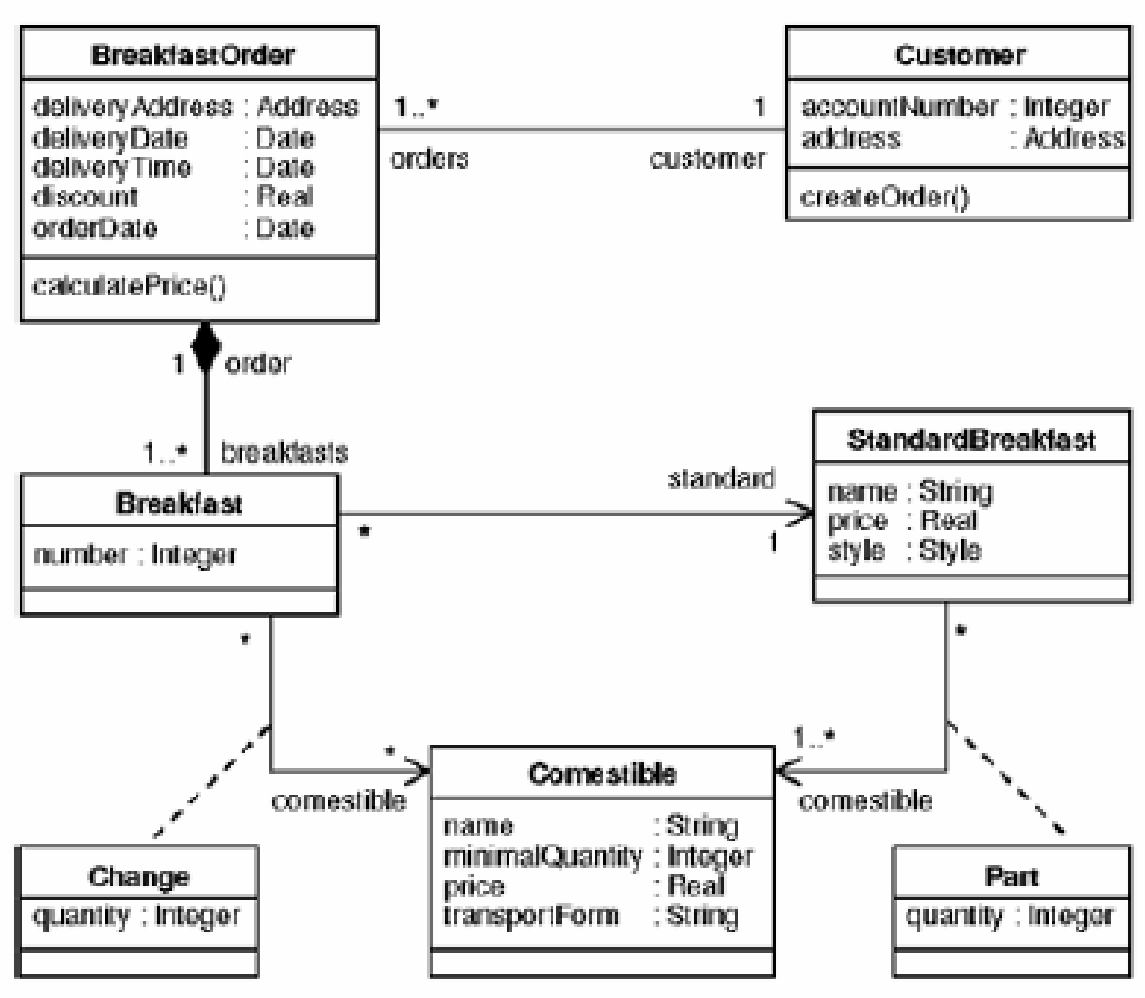
MDA应用案例

- Rosa早餐服务系统：
  - 提供早餐订购服务，有若干种套餐：如法式早餐、英式早餐等
  - 客户可以在套餐基础上调整食物
  - 记录客户信息以便得到配送信息和打折
- 设计决策：采用基于Web的三层应用



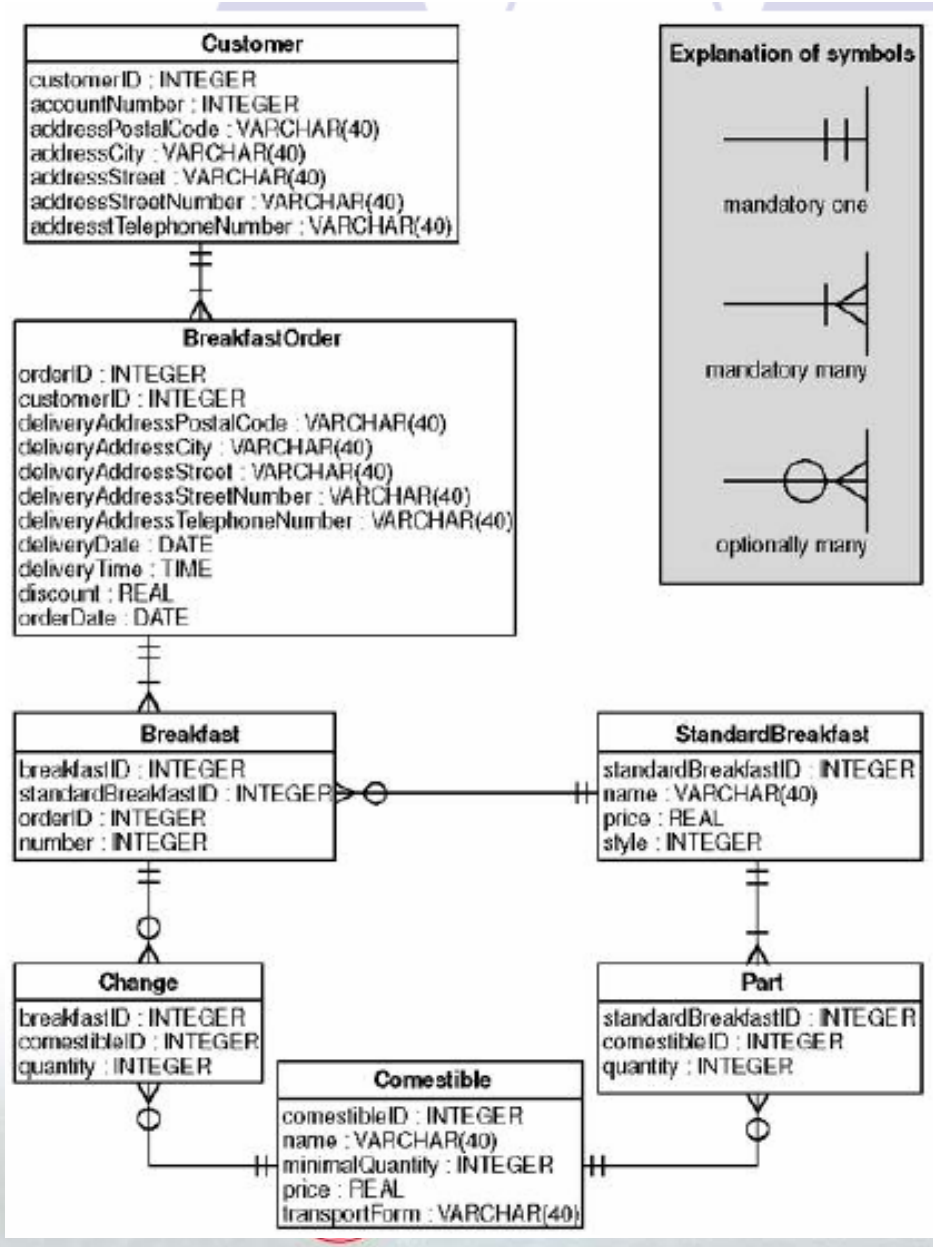
MDA应用案例

- Rosa系统PIM



MDA应用案例

- PIM到关系PSM的变换
  - 每个类变换成一 个表
  - 当属性的类型不是基本数据类型而是类时，对应字段应该包含外键
  - 设计到多重性问题，需要在多一端设置到另一端的关联；当多对多的情况，需要设置关联类

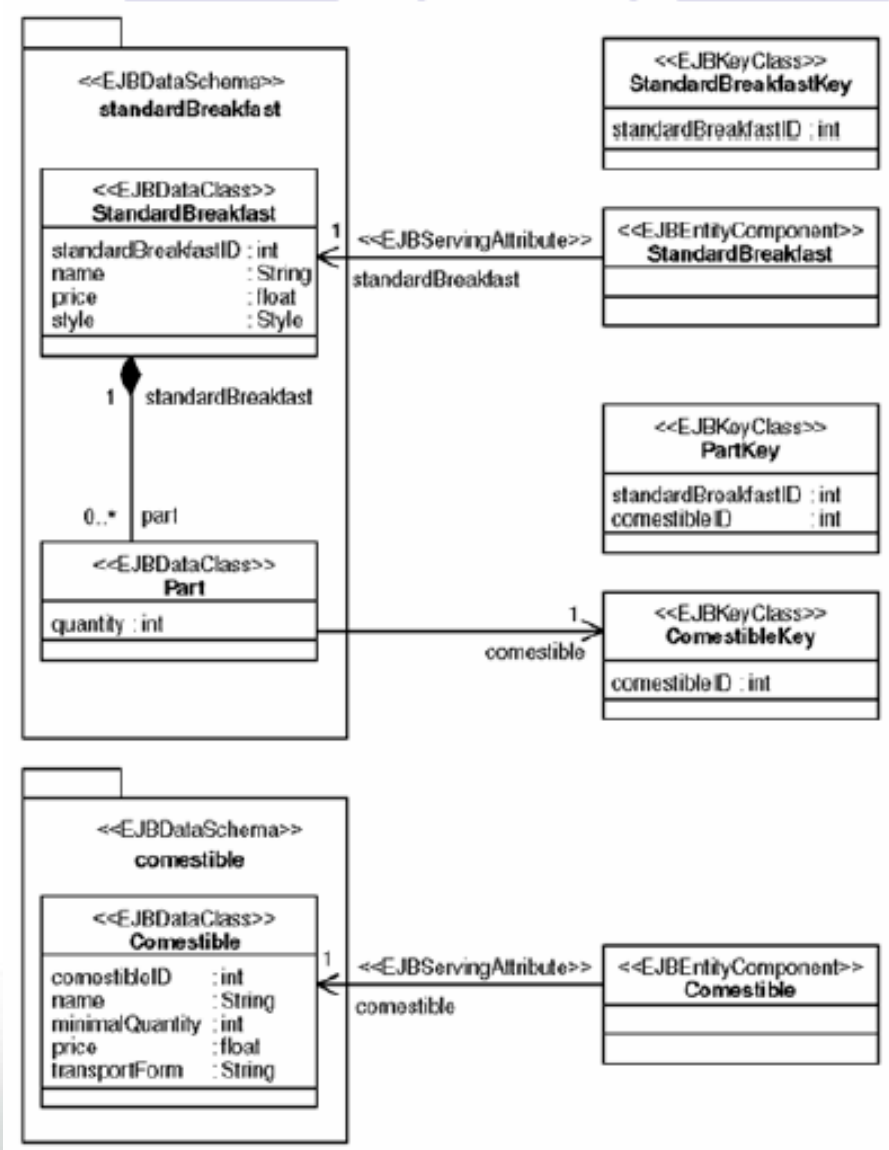


MDA应用案例

- PIM到EJB PSM的变换
  - 粗粒度的构件模型：构件较大，交互频率较低，每次交互传递大量数据
  - EJB数据schema：一组类、属性、关联，在数据交互时它们被EJB构件作为整体来对待
  - EJB数据类：保存EJB数据，是数据模式的一部分
  - EJB数据对象：EJB数据类的实例
  - EJB主键类：保存用来区分EJB数据对象的数据

MDA应用案例

- 部分EJB PSM
  - 类=〉主键类
  - 非组合成份 =〉EJB构件和数据模式
  - 类=〉数据类和归入数据模式
  - 关联=〉EJB关联和归入数据模式
  - 关联类=〉数据类和2个EJB关联
  - 属性=〉数据类属性
  - 操作=〉 EJB构件操作

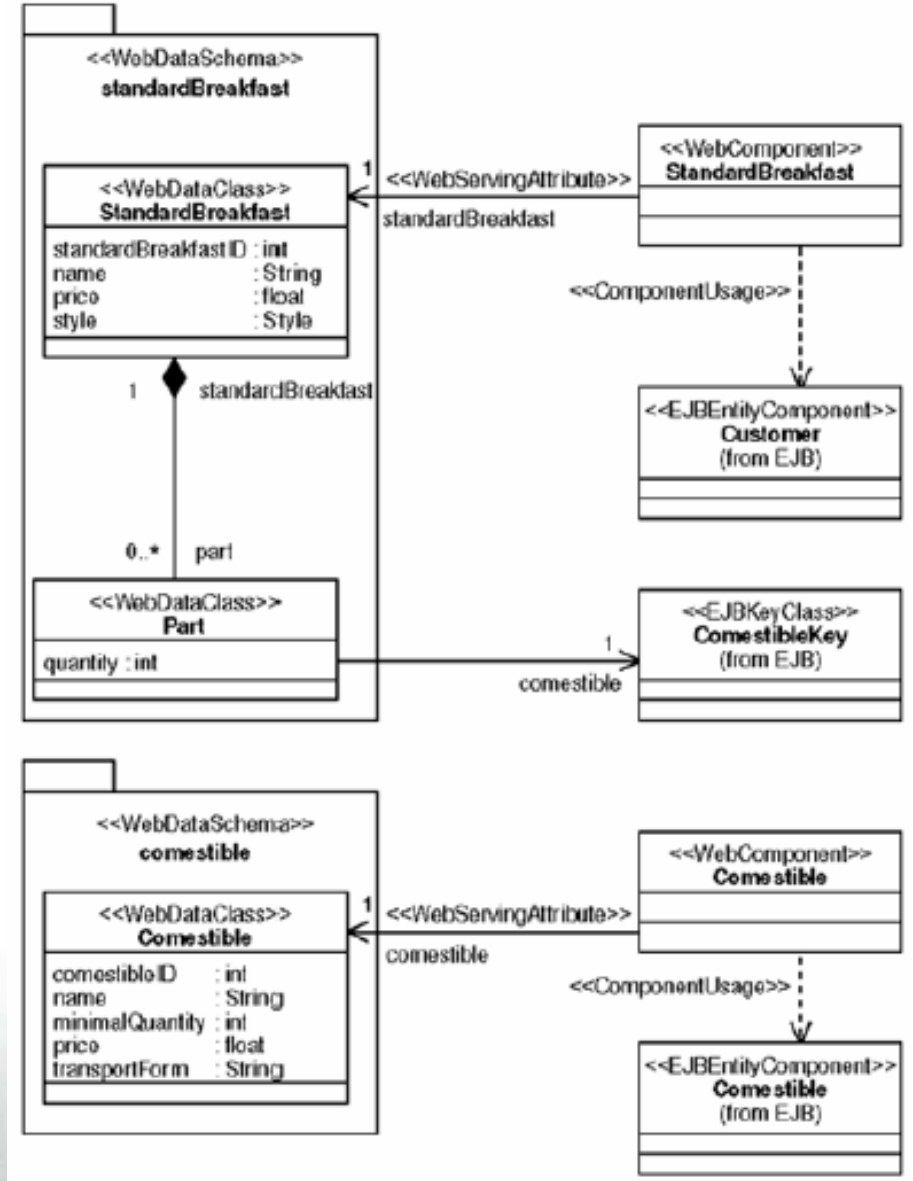


MDA应用案例

- PIM到Web PSM的变换
- 与EJB PSM的不同
  - Web模型的数据类型定义了用户表示和交互细节
  - Web模型中没有主键类，引用EJB中的主键类
  - 增加了Web动作，定义最终用户可以引发的动作
- 变换规则与EJB变换类似

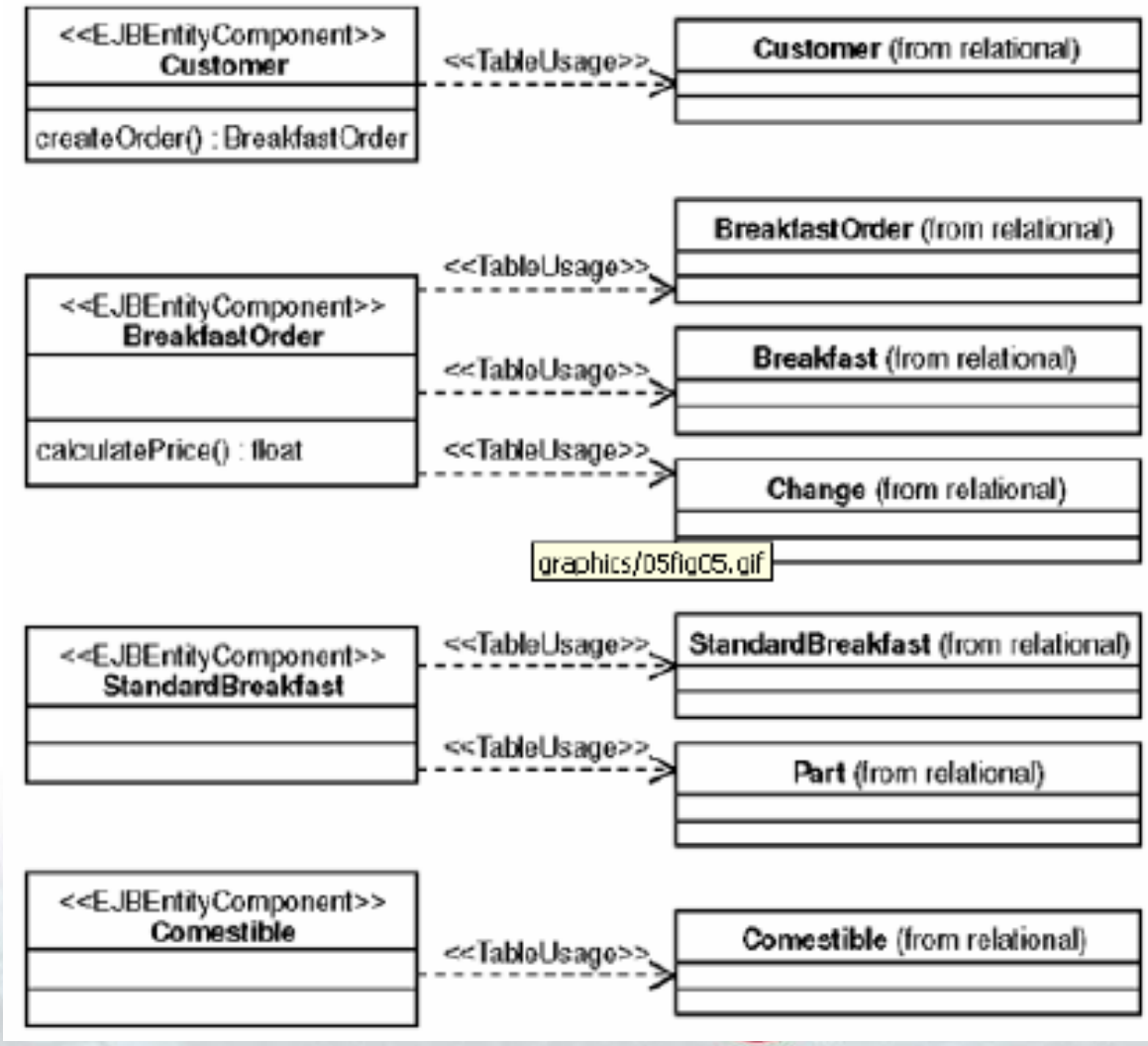
MDA应用案例

- 部分EJB PSM
  - 非组合成份 =〉构件和数据模式
  - 类=〉数据类和归入数据模式，由最外组合类生成
  - 关联=〉关联和归入数据模式
  - 关联类=〉数据类和2个关联类
  - 属性=〉数据类属性
  - 操作=〉 web构件操作



MDA应用案例

- 通信桥接器：关系PSM、EJB PSM、Web PSM



MDA应用案例

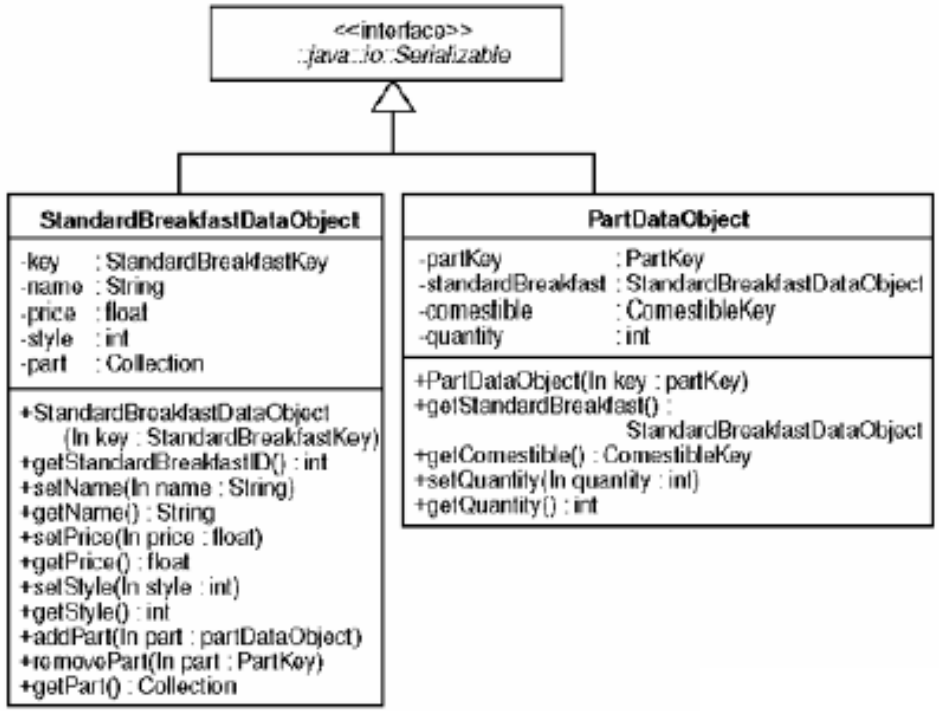
- 关系PSM到代码
  - 表　CREATE TABLE{…}
  - 列　COLUMN TYPE NOT NULL
  - 主键列　PRIMARY KEY(…)

```
CREATE TABLE StandardBreakfast {
  standardBreakfastID  INTEGER  NOT NULL,
  name                 STRING   NULL,
  price                REAL     NULL,
  style                INTEGER  NULL,
  PRIMARY KEY (standardBreakfastID)
};
CREATE TABLE Part {
  standardBreakfastID  INTEGER  NOT NULL,
  comestibleID         INTEGER  NOT NULL,
  quantity             INTEGER  NULL,
  PRIMARY KEY(standardBreakfastID, comestibleID)
};
```

MDA应用案例

EJB PSM到代码的变换——

EJB构件PSM到EJB类PSM的变换



MDA应用案例

- Web PSM到代码的变换
  - Web构件=〉JSP
    - 生成useBean，访问对应的EJB数据管理对象
    - 获得所有数据对象
    - 对每个数据对象生成HTML行，其中包含各property
  - 生成MainMenu JSP、AppError JSP

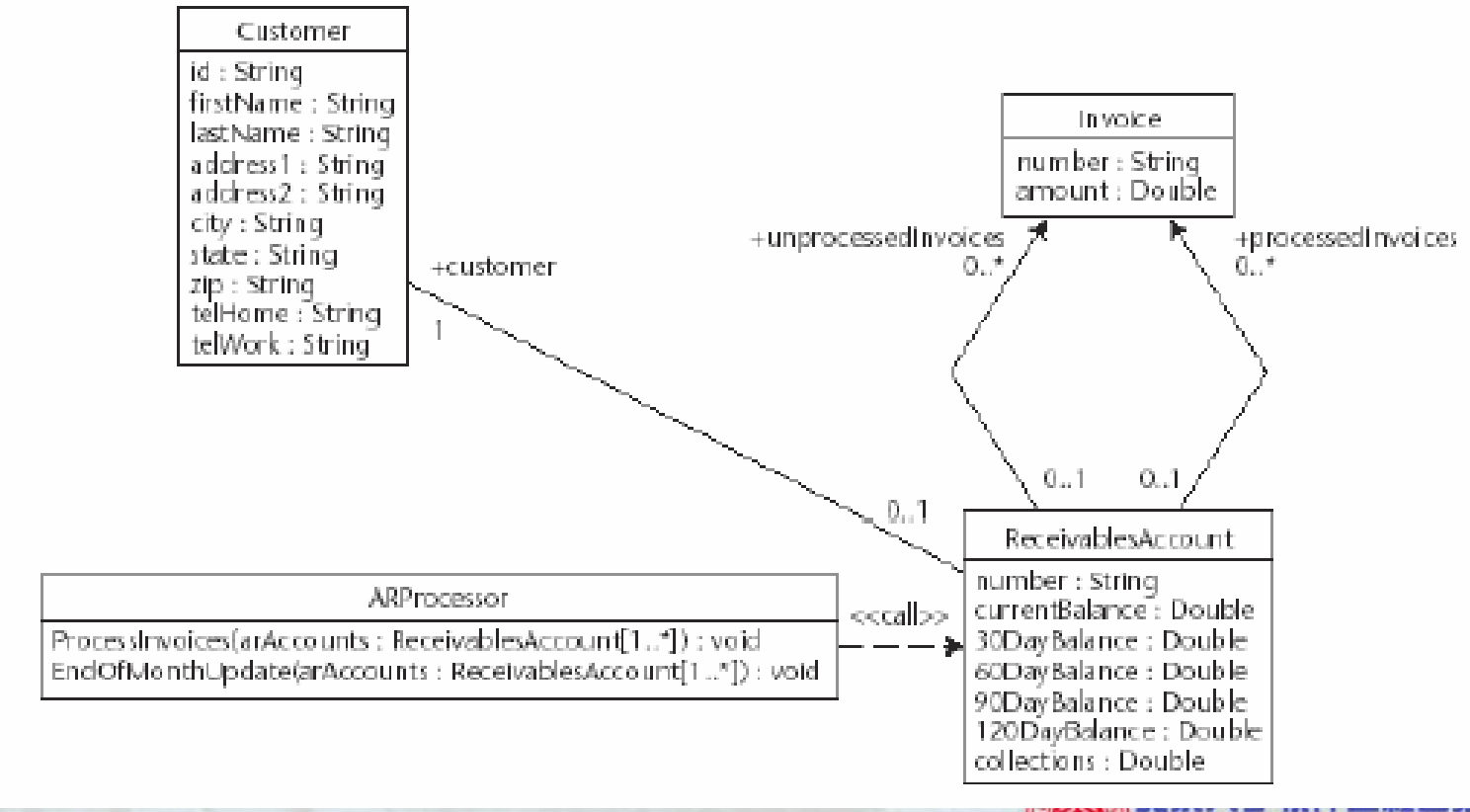
```
<tr>
   <td align=left><font size="3"><jsp:getProperty name="Customer"
property="id"/></font></td>
   <td align=left><font size="3"><jsp:getProperty name="Customer"
property="address"/></font></td>
   <td align=left><font size="3"><jsp:getProperty name="Customer"
property="accountNumber"/></font></td>
</tr>
```

内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

模型

- 采用契约式设计增强模型的精确性和完整性
  - 关于操作的断言——前置条件和后置条件
  - 关于系统状态的断言——不变式



模型

对操作EndofMonthUpdate的约束（采用OCL语法）

```
-----------------------------------------------
--ARProcessor::EndOfMonthUpdate pre-conditions
-----------------------------------------------
--There are no unprocessed invoices.

context ARProcessor::EndOfMonthUpdate (arAccounts : Set
(ReceivablesAccount)) pre:

   arAccounts->forAll (unprocessedInvoices->isEmpty () )
-----------------------------------------------
--ARProcessor::EndOfMonthUpdate post-conditions
-----------------------------------------------
context ARProcessor::EndOfMonthUpdate (arAccounts : Set
(ReceivablesAccount)) post:

   arAccounts->forAll
   (
      -- @pre modifies an identifier to refer to the value it had
      -- before the operation executed.
      currentBalance = 0 and
      30DayBalance =  currentBalance@pre and
      60DayBalance =  30DayBalance@pre and
      90DayBalance =  60DayBalance@pre and
      120DayBalance = 90DayBalance@pre and
      Collections = collections@pre + 120DayBalance@pre
   )
```

模型

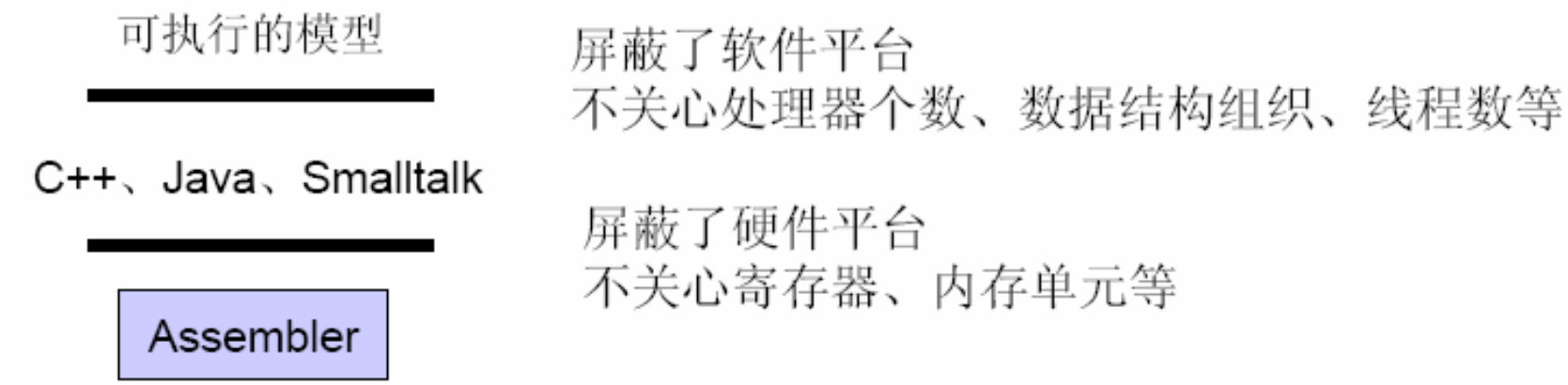
- 契约式设计的说明
  - 使模型同编程一样精确，精确性和细节不是一回事
  - 容易捕获异常
  - 便于生成测试框架，提供了质量保证
  - MDA促进了DBC的应用，因为这是提高整体效率的方法

模型

- 模型不仅描述静态结构，还要体现动态行为
  - 状态机（协议状态机和客户不可见状态转换）
  - 活动图：流程图，活动之间的控制
  - 交互图：描述执行任务时的交互和协议
  - 用例图：人与计算机的边界交互
  - 动作语义：定义了动作语言的元模型，直接使用动作语义表达式描述执行行为

模型

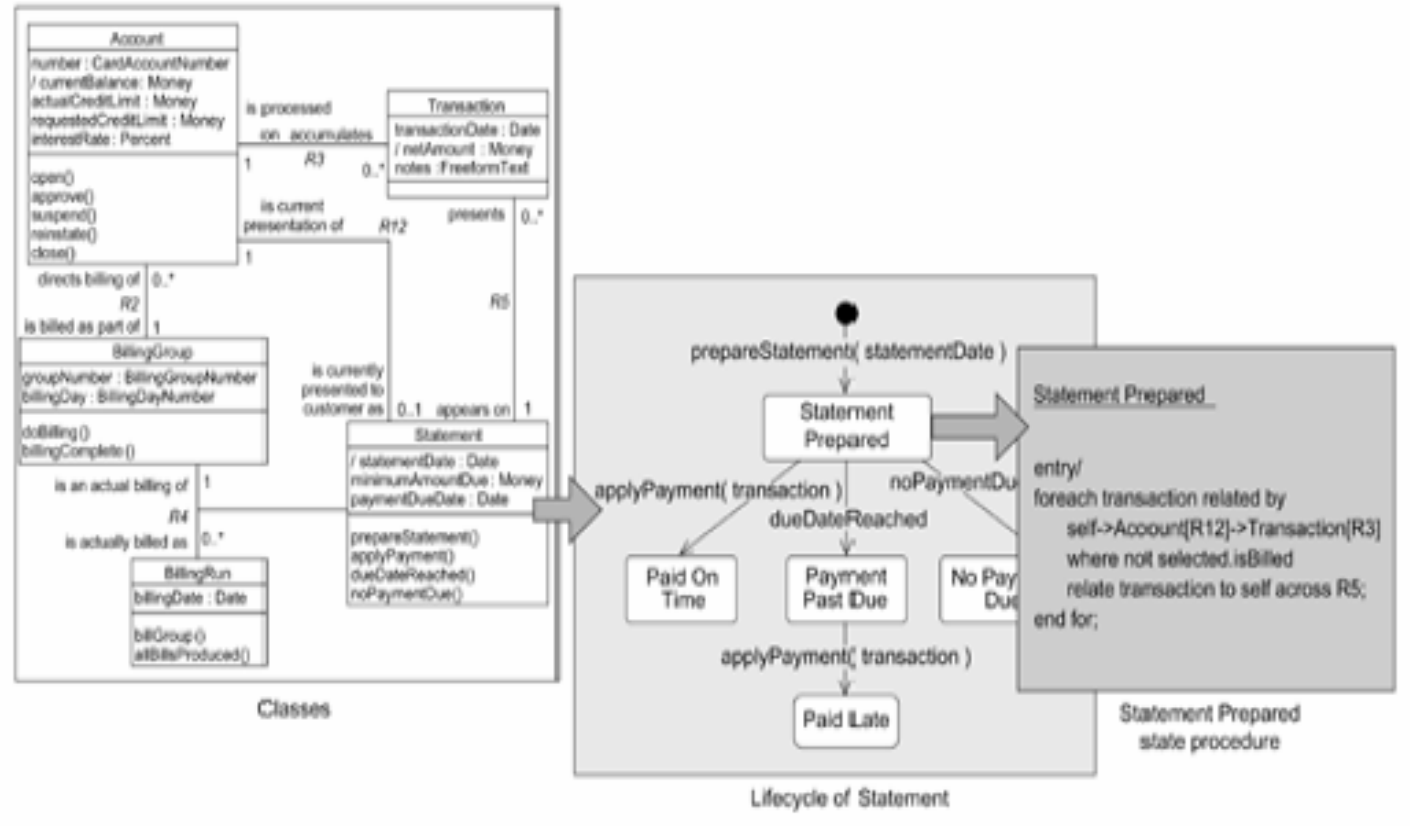
- 可执行的模型：类似于代码，但抽象级别更高



- 可执行的模型并不是最终的系统，可能与其他模型组合在一起共同生成系统(Weave)，其他系统包括用户界面、安全机制、数据存储机制等

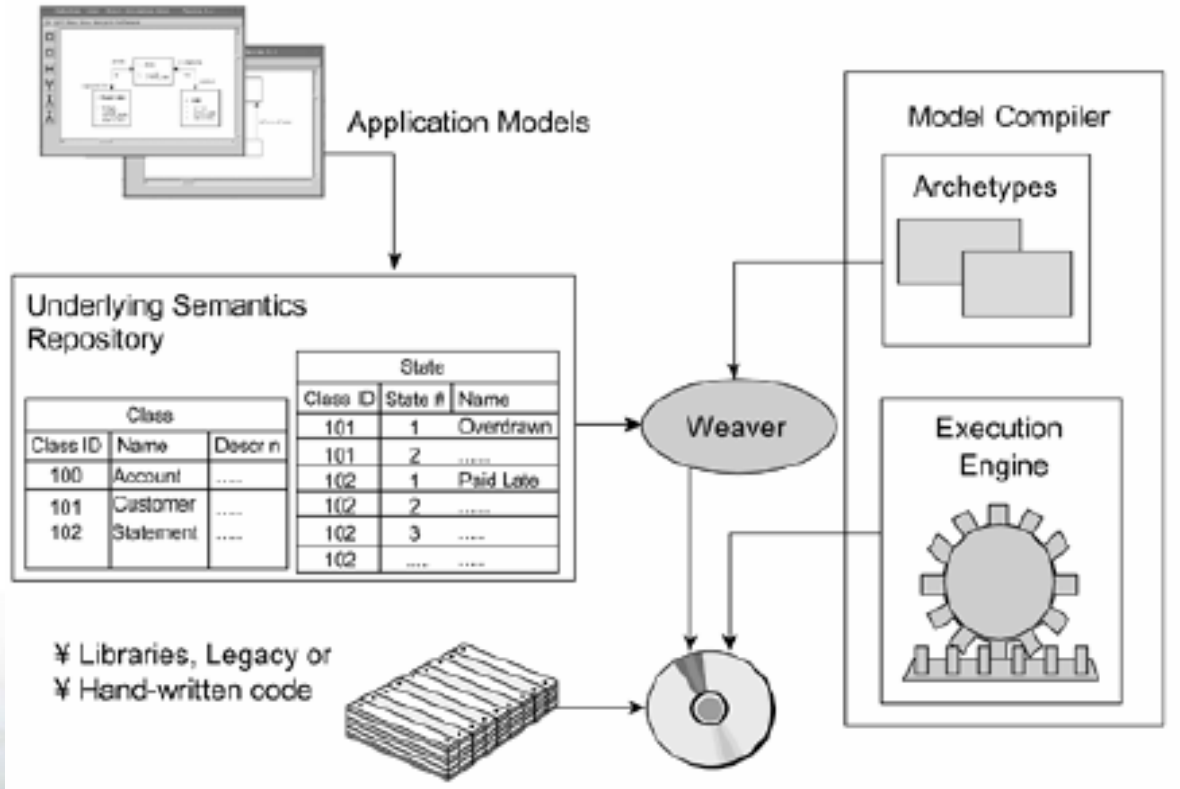
模型

- Executable UML – UML的扩展(profile)，为UML子集定义了执行语义



模型

- 模型编译：根据一系列规则，将带标记的模型映射到通用基础设施上，形成可执行代码的过程



模型

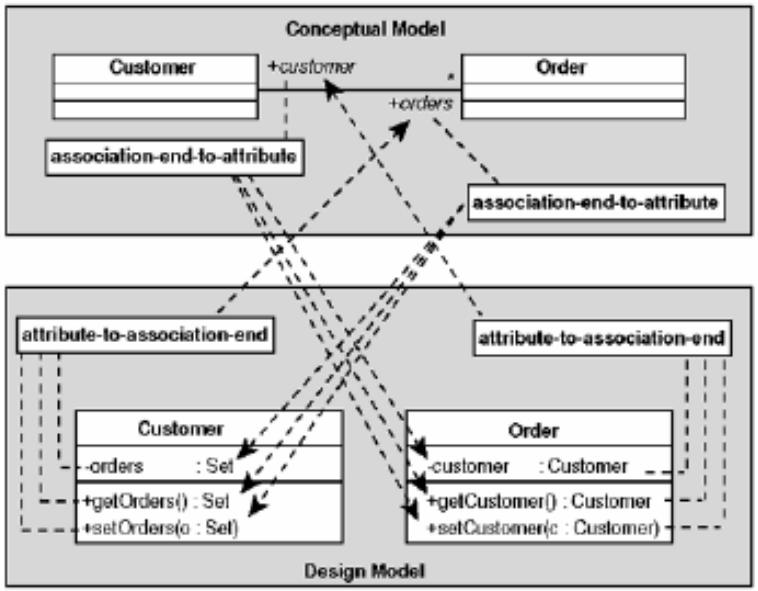
- 创建可编译的类模型建议
  - 不要为属性定义get和set操作
  - 明智的使用关联可溯性
  - 规定多值属性时要小心（顺序、单一性）
  - 正确使用聚合（组合和一般聚合）
  - 正确使用抽象类（生成create操作）
  - 区分感兴趣和不感兴趣的操作（如EJB中的create方法）
  - 语法完整性
    - 规定所有属性和操作参数的类型
    - 指定关联两端的多重性
    - 指定所有关联和关联端的名字
    - 不要留下任何悬挂的未用模型元素
    - 语义完整性：DBC契约设计

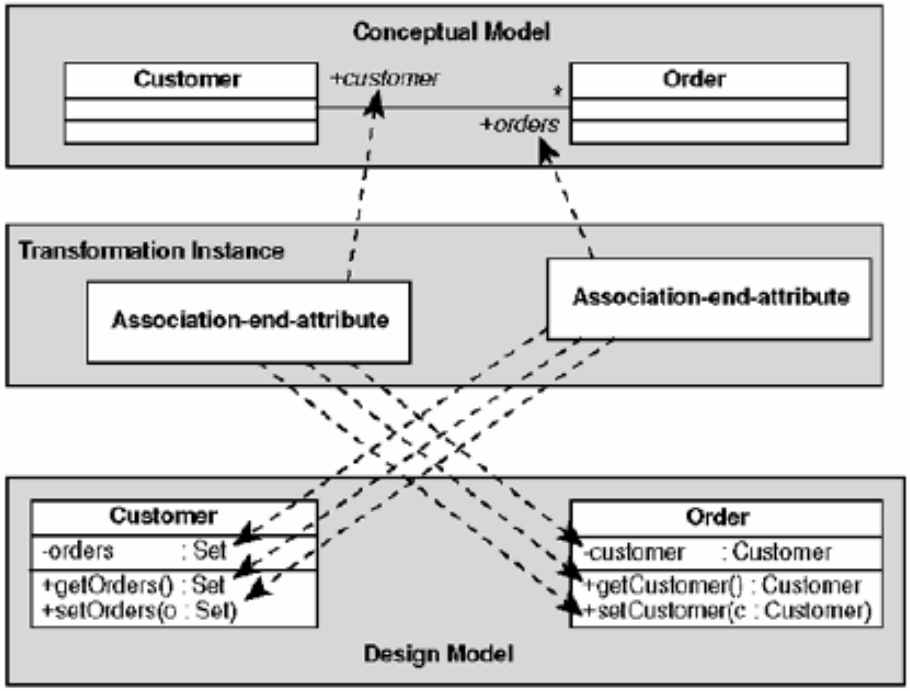
变换定义

- 变换的性质
  - 可调性(Tunability)：变换规则的应用是可调节的
    - 手工控制、变换条件、变换参数、额外信息
    - 变换参数的例子String =〉VARCHAR (parameter i: Integer [default=20])
  - 可追溯性(Traceability)：目标模型中的元素可以追溯到源模型中
  - 增量一致性(Incremental Consistency)：把目标相关信息加入目标模型后，重新生成的目标模型会保留这一信息
  - 双向性(Bidirectionality)：变换可以在源和目标之间双向进行

变换定义

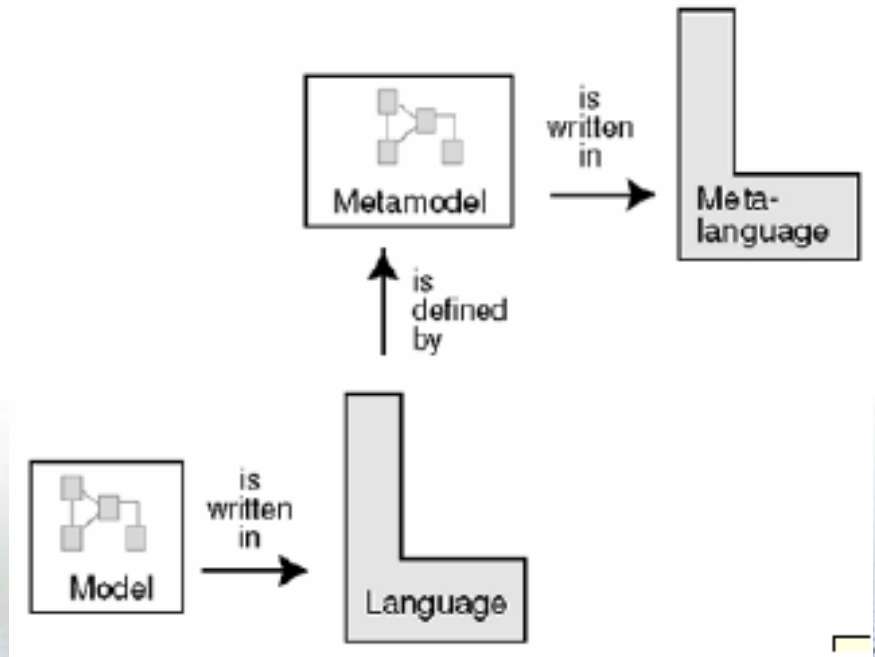
- 维持源-目标关系的一致性



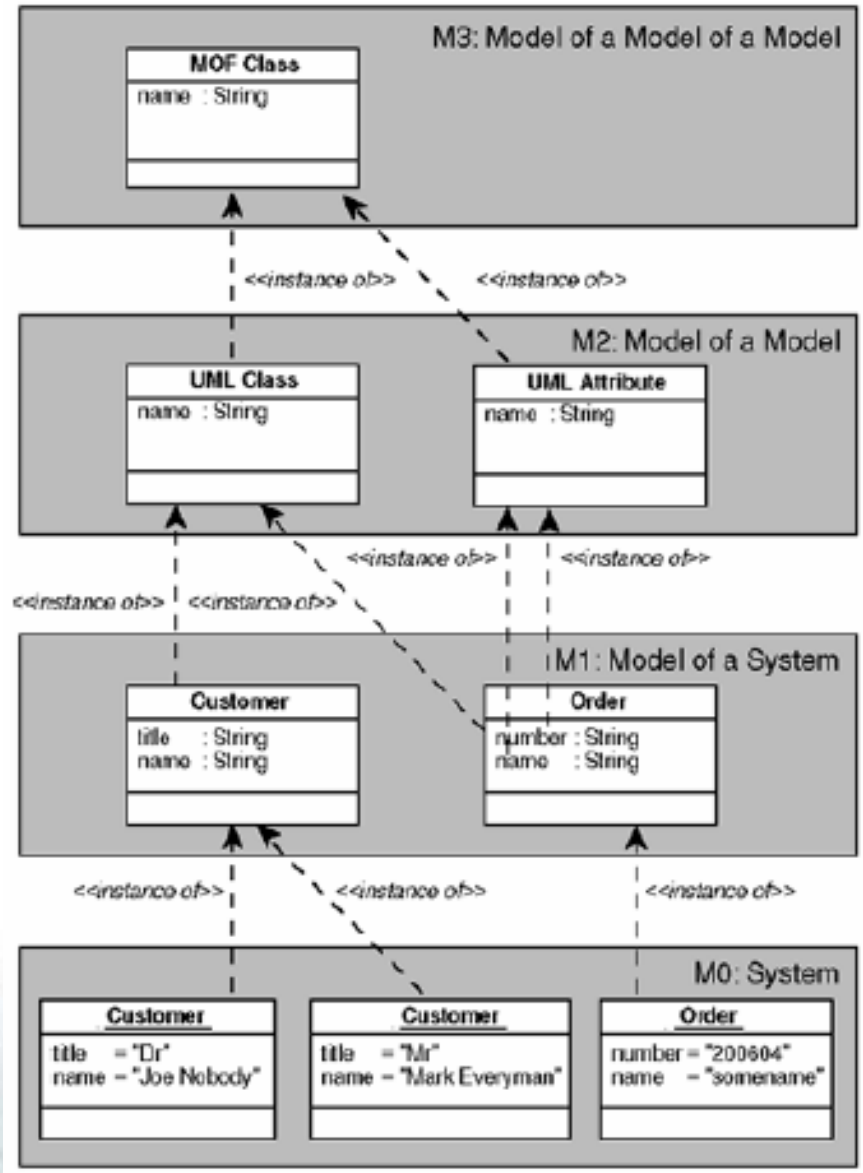


元建模

- 怎样精确定义语言？BNF范式（纯文本）
- 例：在UML模型中，定义了Customer类，类是一种建模元素，即元模型中的元素
- OMG中定义了4个建模层次
  - M0层：实例层
  - M1层：模型层
  - M2层：元模型层
  - M3层：元元模型层
  - 不再需要更多的层次

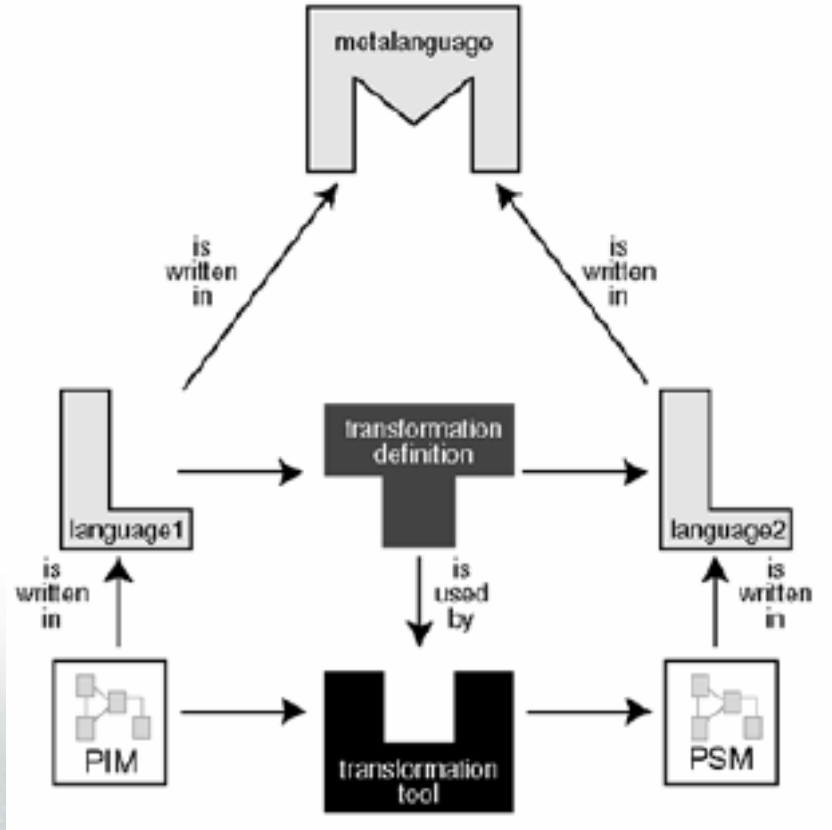


元建模



元建模

- 在MDA中使用元建模
  - 定义建模语言，变换工具得以理解模型
  - 变换规则使用源语言和目标语言的元模型来定义变换
- 扩展的MDA框架
  - 引入了定义语言的元模型



变换定义语言

- 变换定义语言应为形式化的
- 变换定义中包括
  - 源语言的引用
  - 目标语言的引用
  - 可选变换参数
  - 一组命名的源语言模型元素（称为S），来自源语言的元模型
  - 一组命名的目标语言模型元素（成为T），来自目标语言的元模型
  - 双向标记
  - 源语言条件：应用变换的前置条件
  - 目标语言条件：应用变换的后置条件
  - 变换规则集合：将S中的元素变换为T中的元素
- 变换定义可以是一组变换规则，也可以按顺序执行别的变换定义

变换定义语言

例1：

PIM中的共有属性

PSM中的私有属性

+set&get操作

```
Transformation PublicToPrivateAttributes (UML, UML) {
   params
        setterprefix: String = 'set';
        getterprefix: String = 'get';
   source
        sourceAttribute : UML::Attribute;
   target
        targetAttribute : UML::Attribute;
        getter          : UML::Operation;
        setter          : UML::Operation;
   source condition
        sourceAttribute.visibility = VisibilityKind::public;
   target condition
        targetAttribute.visibility = VisibilityKind::private      and
        setter.name = setterprefix.concat(targetAttribute.name)   and
        setter.parameters->exists( p |
                                   p.name = targetAttribute.name
                                   and
                                   p.type = targetAttribute.type) and
        setter.type = OclVoid                                     and
        getter.name = getterprefix.concat(targetAttribute.name)   and
        getter.parameters->isEmpty()                              and
        getter.type = targetAttribute.type                        and
        targetAttribute.class = setter.class                      and
        targetAttribute.class = getter.class;
   bidirectional;
   mapping
        sourceAttribute.name <-> targetAttribute.name;
        sourceAttribute.type <~> targetAttribute.type;
}
```

变换定义语言

例2

- PIM中的关联端

  PSM中的私有属性

  + set&get操作

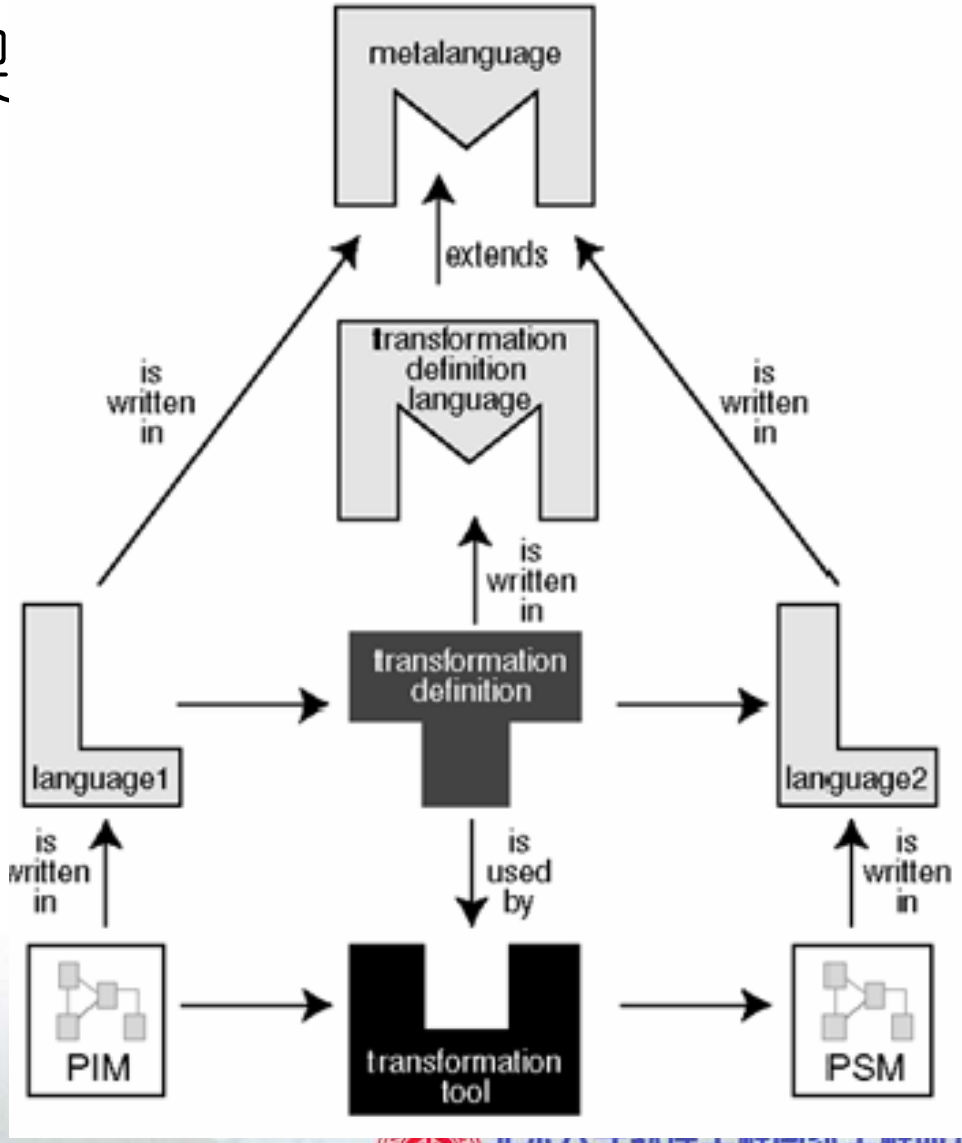
- 方法
  - 先将关联端变换为公有属性
  - 再运用上一条规则变换

```
Transformation ManyAssociationToAttribute (UML, UML) {
   params -- none
   source
        ae  : UML::AssociationEnd;
   target
        att : UML::Attribute;
   source condition
        ae.upper >= 1;
   target condition
        att.visibility = VisibilityKind::public and
        att.type.isTypeOf(Set);
   unidirectional;
   mapping
        ae.name <~> att.name;
        ae.type <~> att.type.elementType;
}

Transformation SimpleAssociationToAttribute (UML, UML) {
   params -- none
   source
        ae  : UML::AssociationEnd;
   target
        att : UML::Attribute;
   source condition
        ae.upper <= 1;
   target condition
        att.visibility = VisibilityKind::public and
        att.type.isTypeOf(Class);
   unidirectional;
   mapping
        ae.name <~> att.name;
        ae.type <-> att.type;
}
```

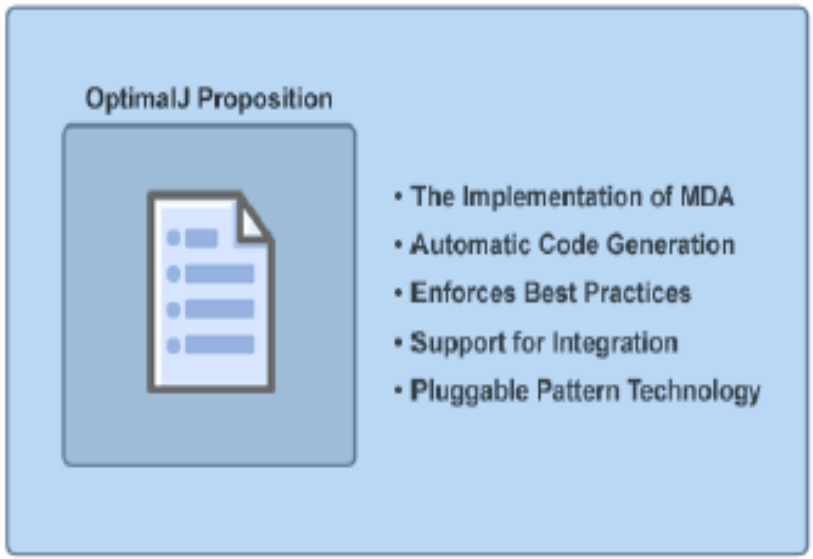
完整的MDA框架

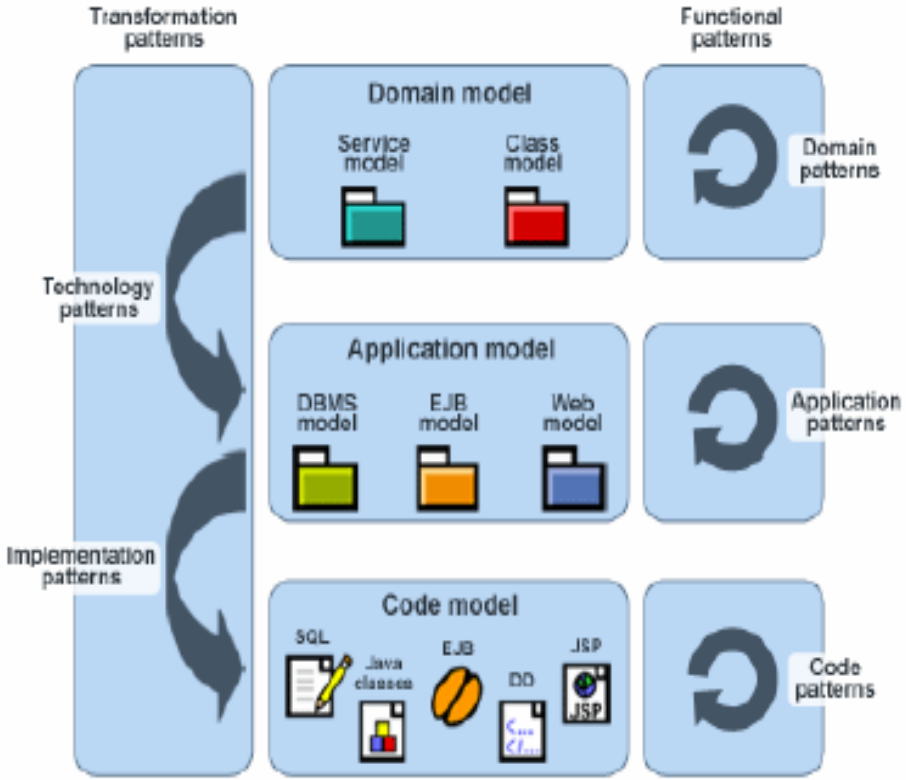
- 完整的MDA框架



MDA工具

- OptimalJ



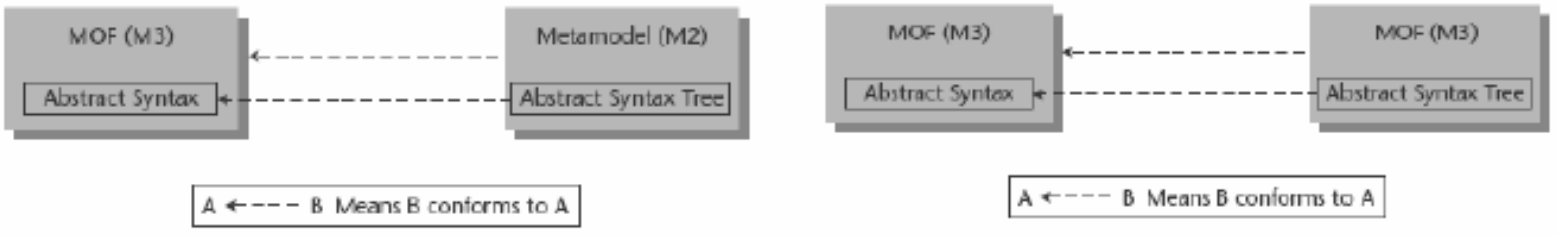


内容

- MDA简介
- MDA开发过程
- 简单的MDA框架
- MDA应用案例
- 完整的MDA框架
- OMG相关标准

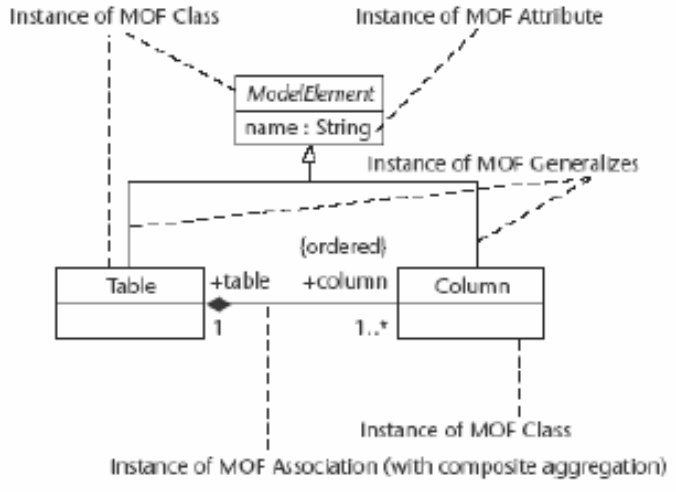
OMG标准

- MOF(Meta Object Facility)是定义建模语言的语言，位于M3层
- 基本思路
  - 建模语言有多种，有各自的建模结构集
  - 需要有一种一致的方法来描述语言结构
  - 将这些不同的建模结构合并成一个集合是不合理的
  - 在这些建模结构之上建立一层对它们进行一致的描述
    => MOF



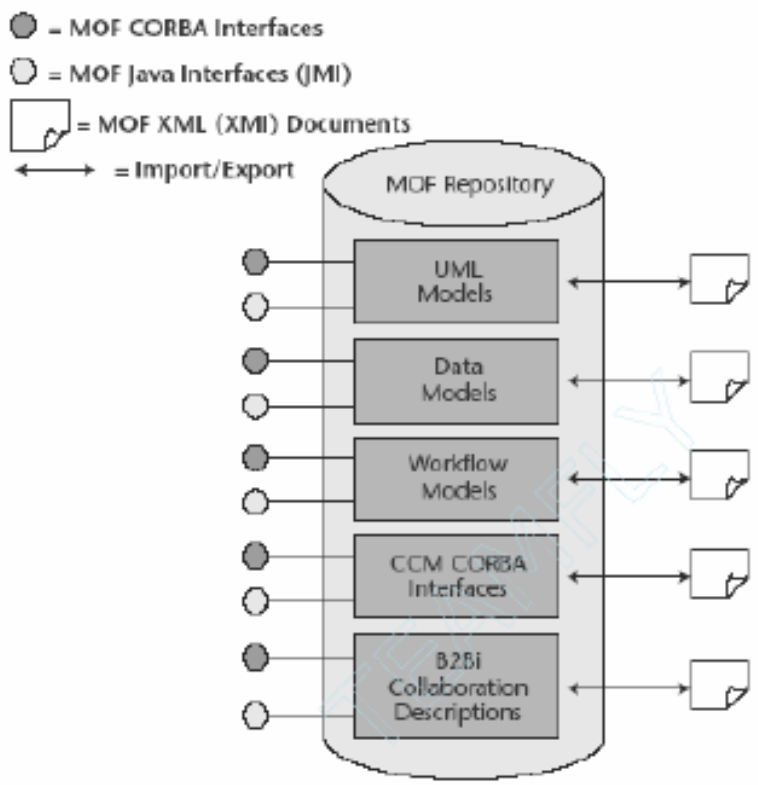
OMG标准

- MOF模型
  - 借用UML类建模结构，将其作为描述建模结构的抽象语法
  - 由MOF生成的元模型具有共性
    - UML类图元模型使用组合classifier- feature
    - CWM关系数据元模型使用组合table-column
    - UML状态图元模型使用组合status-transition



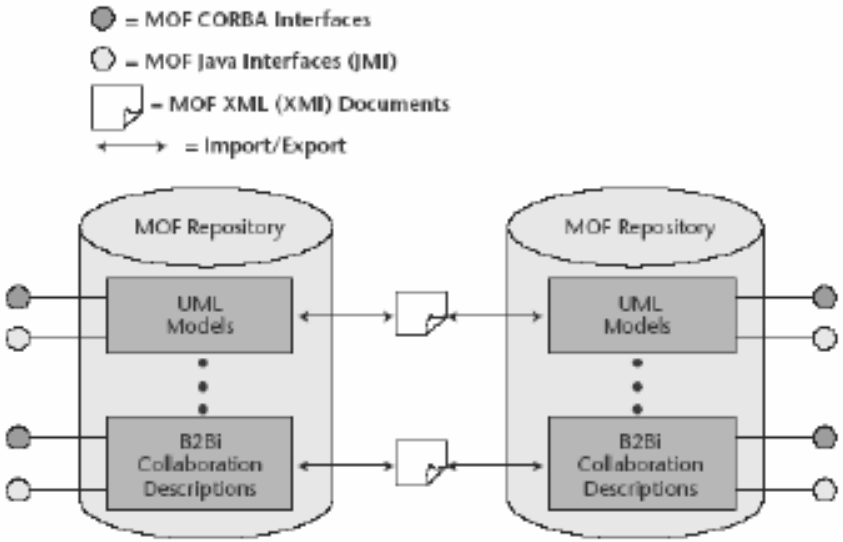
OMG标准

- 元数据管理
  - MOF是将模型序列化为XML文档的一种机制
  - 模型交换：用于为M1层模型定义基于流或基于文件的交换格式，基于XML，被称为XML Metadata Interchange
  - MOF仓库接口：从基于MOF的仓库获取M1层模型

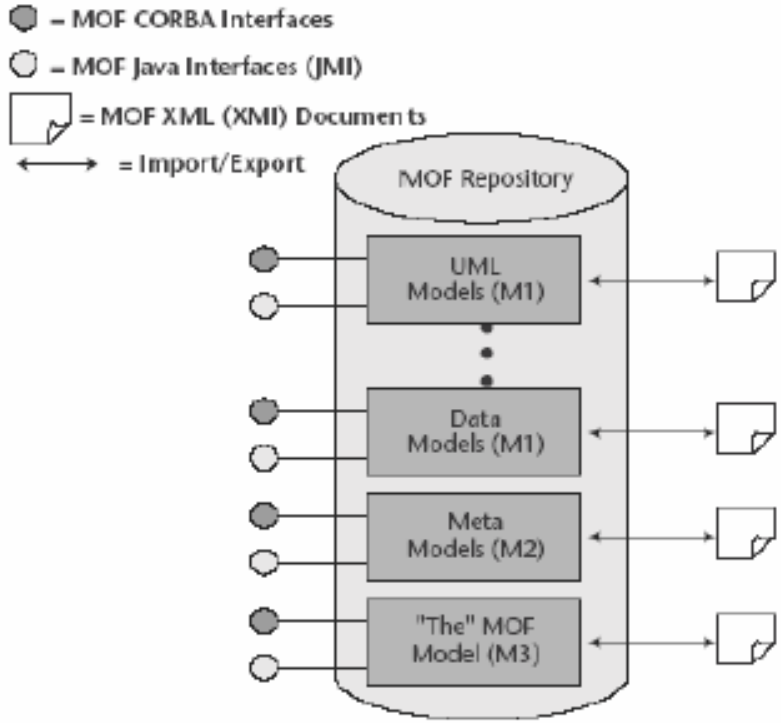


OMG标准

- MOF集成仓库管理元数据



通过XMI在仓库之间传递模型



MOF仓库以类似于M1层的方式管理M2层和M3层

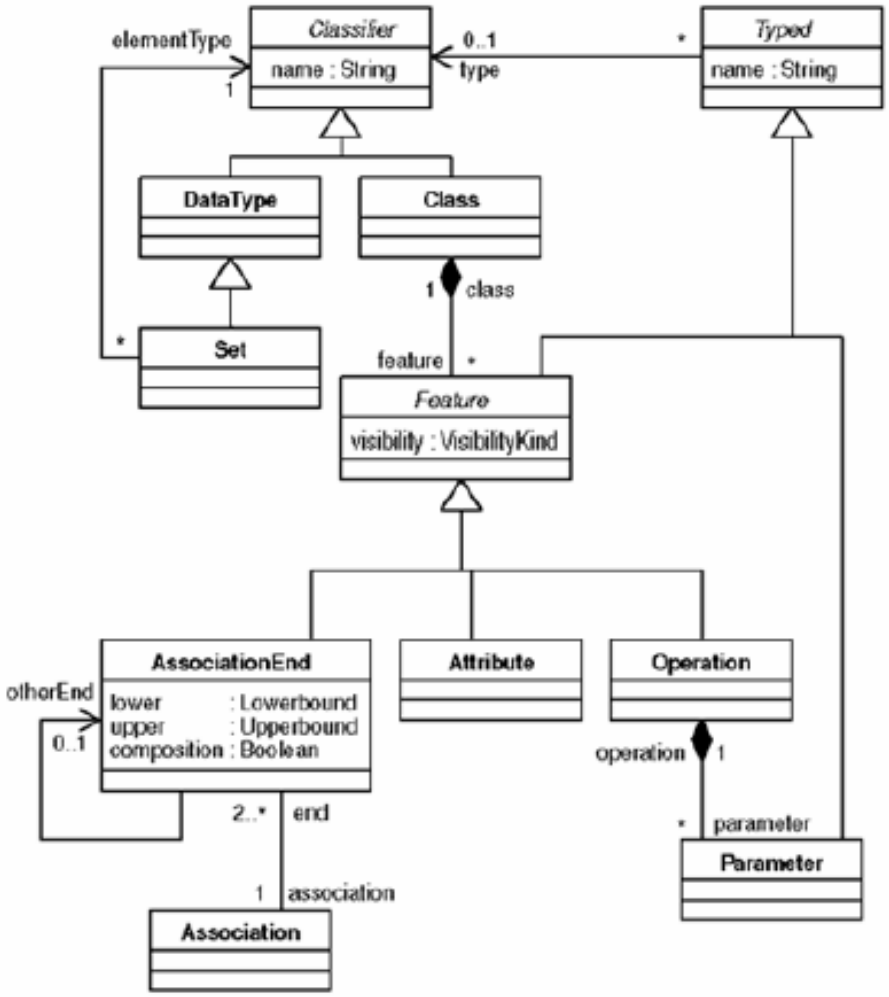
OMG标准

- MOF在MDA中的作用
  - 提供了思考建模语言的概念和工具
  - 使用建模语言的元模型来定义建模语言之间的变换
  - 可以定义全新的建模语言
  - MOF是让MDA成为现实的核心技术

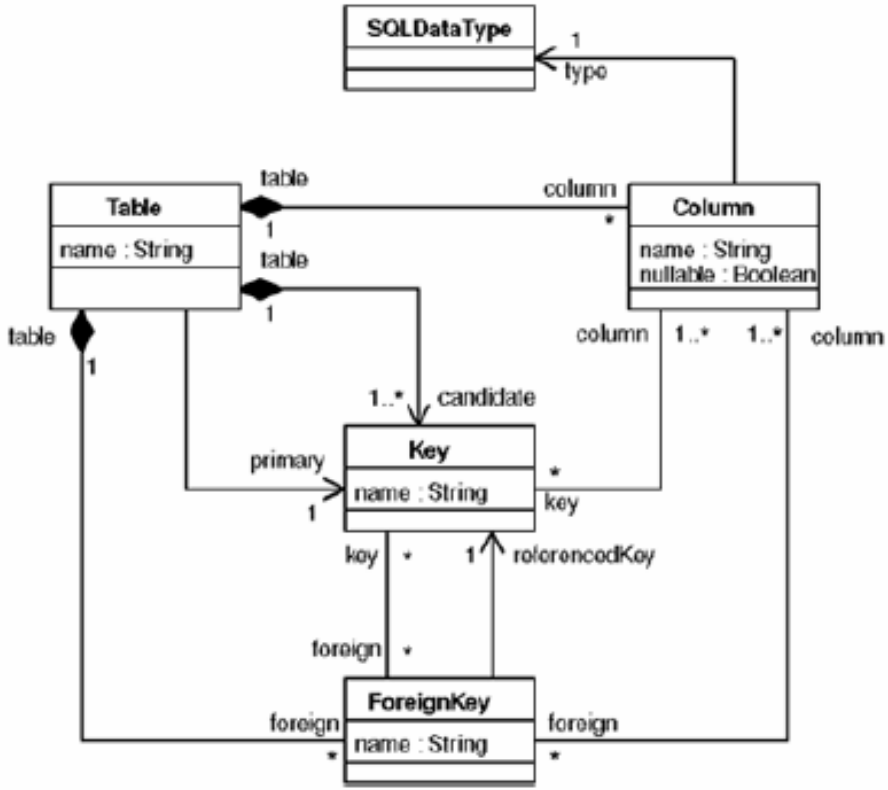
OMG标准

- UML (Unified Modeling Language)
  - 位于M2层的标准建模语言
  - UML元模型是MOF模型的实例
- UML在MDA中的作用
  - 为系统建模
  - 针对元模型，定义模型之间的变换

元建模



简化的UML元模型



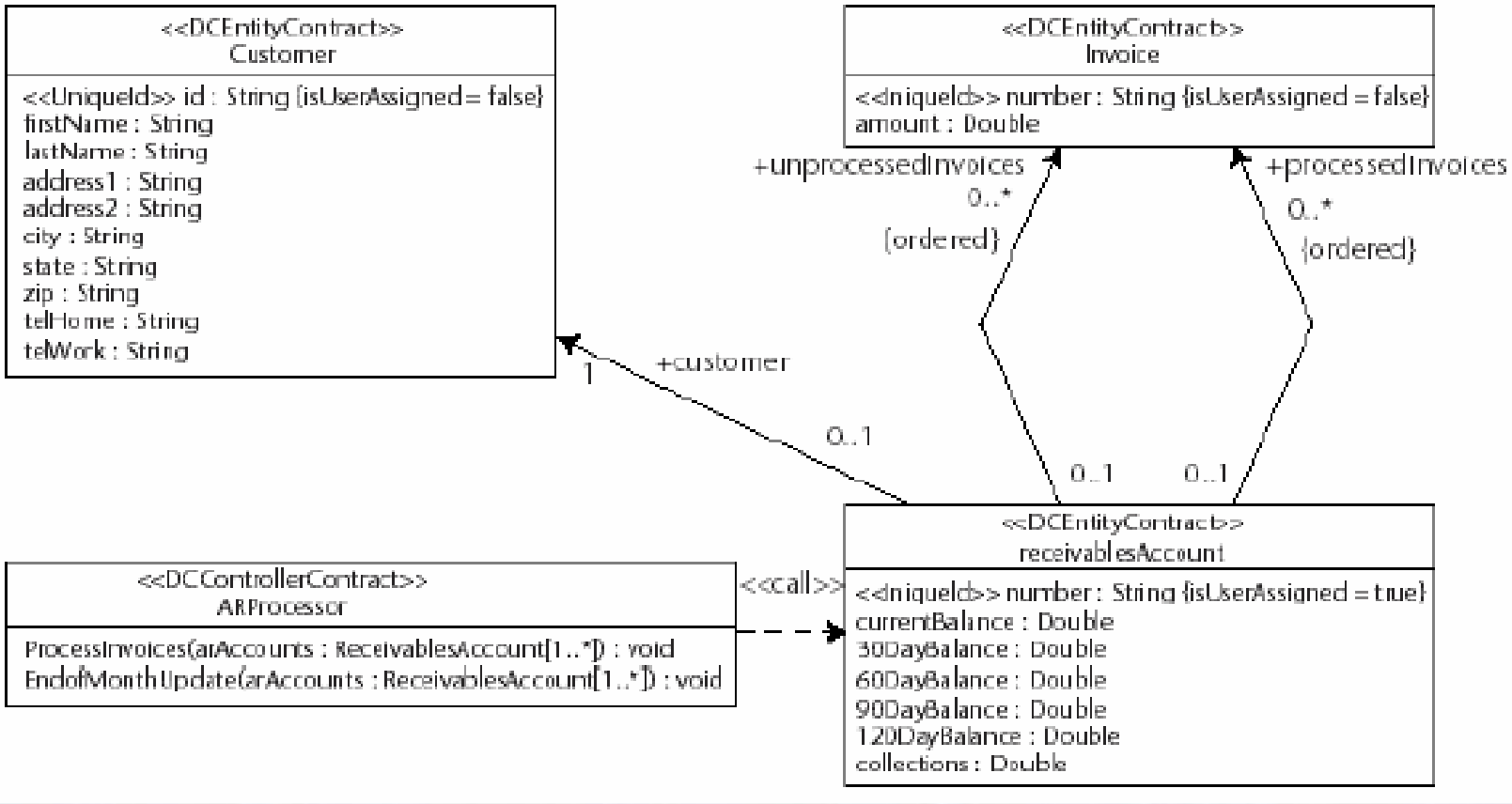
简化的SQL元模型

OMG标准

- UML Profile：UML的特化机制，定义一种使用UML的特定方式
- 实现方法
  - 构造型(stereotype)：具有一个名字并关联到UML元模型中的元素，如<<JavaClass>>是为UML元类定义的
  - 约束(constraint)：关联到构造型定义，描述对要应用该构造型的模型元素的实例的限制，如Java Class必须至少有一个超类
  - 标记值(tagged value)：额外的元属性，具有名字和类型
- UML Profile在MDA中的作用
  - profile通过复用UML元模型定义了一种新的语言
  - 增强模型的特定表达能力
  - 一般针对特定平台，对应PSM

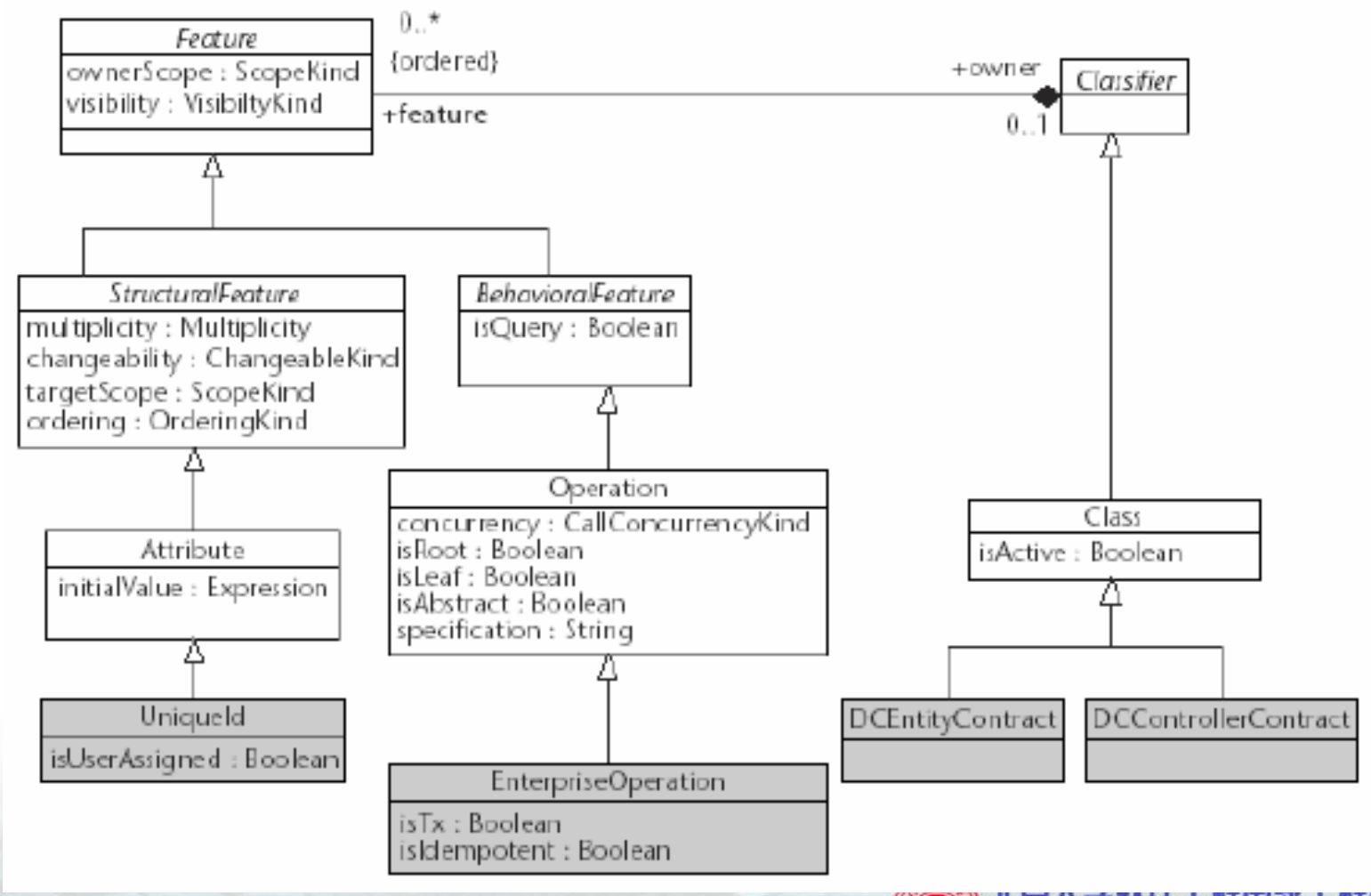
OMG标准

- 构造型和标记值示例



OMG标准

- 通过MOF扩展UML的重型扩展



OMG标准

- UML动作语义(Action Semantics)
  - 对UML的扩展，提供了一个动作语言
  - 用来编写可直接执行的UML模型
  - 将系统的所有状态都关联到状态机
  - 目前还没有标准化
- UML AS在MDA中的作用
  - 增强模型的动态表达能力
  - 尽可能使得PIM精确

OMG标准

- OCL(Object Constraint Language)是一种表达式语言
  - 不变式、前置条件、后置条件
  - 规约语言，描述值是什么，不描述如何计算
  - 可用于UML和MOF模型，扩展了模型的表达能力
  - 使用UML+OCL，模型更精确更完整
- OCL在MDA中的作用
  - 增强模型的精确性
  - 应用在变换定义中

OMG标准

- CWM(Common Warehouse Metamodel)
  - 位于M2层的数据仓库建模语言
  - CWM元模型是MOF模型的实例
  - 包括：关系数据库、记录或结构、OLAP、数据挖掘、业务元数据等
- CWM在MDA中的作用
  - 为系统建模
  - 针对元模型，定义模型之间的变换
- QVT (Query, Views, and Transformations)
  - 创建模型视图的语言
  - 查询模型的语言
  - 编写模型定义的语言
  - 解决了模型间的变换如何实现的问题

展望MDA

- 类比高级语言代替汇编语言的过程
- MDA处于“初级阶段”，体现出了提高效率的潜力
- MDA引起的变革——关注焦点从代码转向模型
- MDA对软件开发过程的影响
- MDA对软件开发工具的影响
- MDA对建模语言提出了更高的要求
  - 足够强的表现力：静态和动态方面
  - 一种通用的、不特定于某种应用的语言
  - 适合分布式应用系统
  - 模型和实现之间没有缝隙
  - 支持管理大型模型，如Aspect-Oriented建模方式

参考文献

- MDA Explained: The Practice and Promise of The Model Driven Architecture , Anneke Kleppe等著, 鲍志云译
- Model Driven Architecture: Applying MDA to Enterprise Computing , David S. Frankel 著, 鲍志云译
- Model Driven Architecture, whitepaper of OMG , Richard Soley and OMG Staff Strategy Group
- MDA Guide Version 1.0.1 , Joaquin Miller and Jishnu Mukerji
- MDA Distilled: Principles of Model-Driven Architecture, Stephen J. Mellor , Kendall Scott
- Domain-Specific Modeling and Model Driven Architecture , Steve Cook

谢谢大家！

Questions or comments?