计算机程序设计语言

领域特定语言与元语言

张 天

软件工程组

计算机科学与技术系

南京大学

示例：格兰特小姐的密室

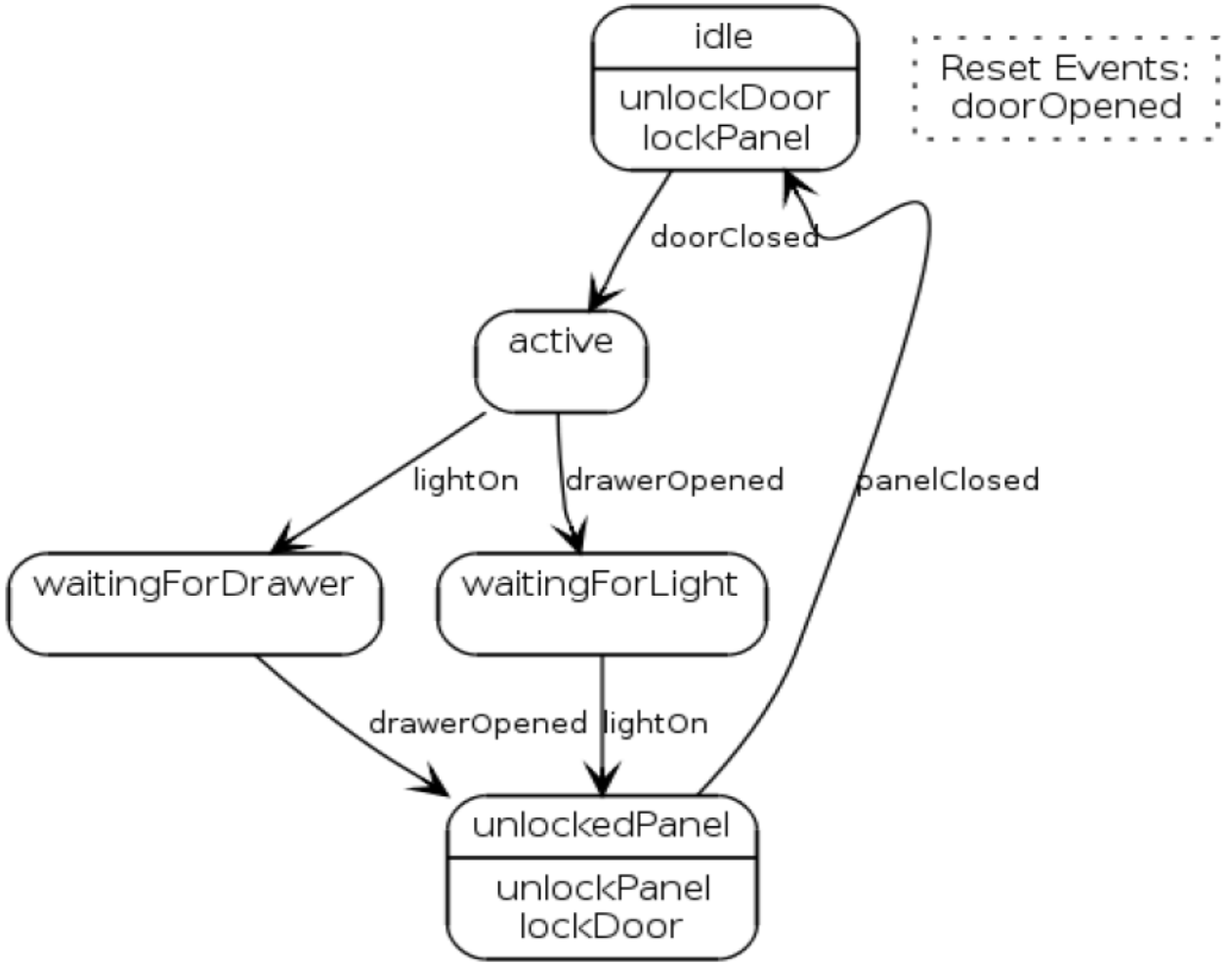
- 通过一个简单的示例，说明：
  - 什么是领域特定语言（DSL，Domain Specific Language）
  - 领域特定语言的构造过程
  - 元语言在其中的重要位置

Miss Grant's Controller

- 故事背景：
  - 格兰特小姐的卧室里有个密室，通常情况下，这个密室都会紧锁着，隐蔽在那里。要打开这个密室，她就要关上门，然后打开柜子里的第二个抽屉，打开床边的灯，三者顺序任意。做完这些，秘密面板就会解锁，她就可以打开密室了。

- 需求：
  - 开发格兰特小姐密室的控制器

使用状态机模型描述这个过程：



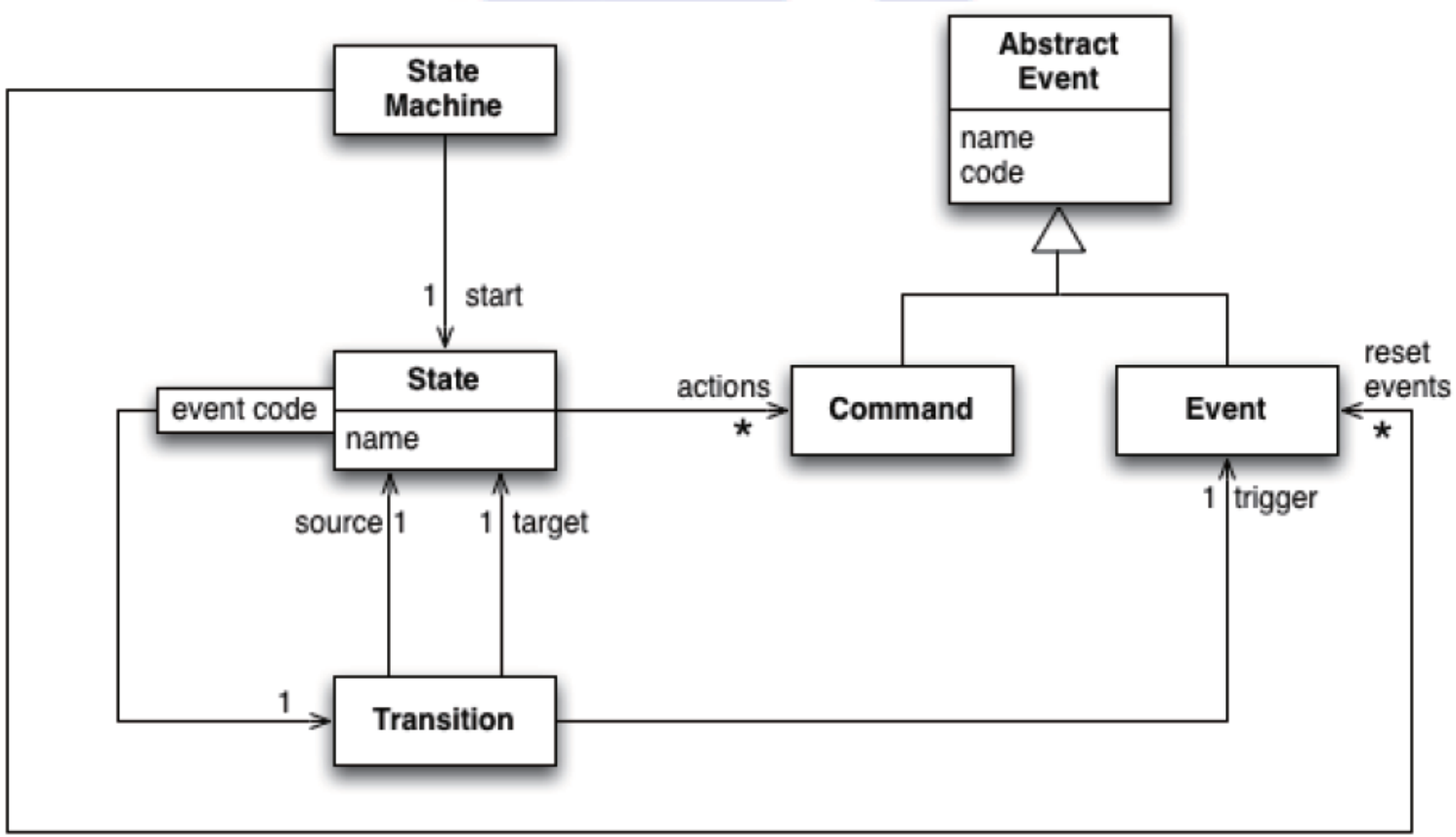
The State Machine Model

- 状态机是一种通过状态和状态之间的变化（即跃迁）来刻画系统行为的形式模型
  - 事件，event
  - 状态，state
  - 转换（跃迁），transition

注：也有称此类模型为状态迁移系统

状态机的类图

思考：将状态机视为一种语言，其语素有哪些？



在代码中体现状态机

- 我们已经基于状态机模型构造了格兰特小姐的控制器，但该状态机模型还不是最终可以执行的系统，因此我们还有基于高级程序设计语言构造真正可以被编译执行的系统！

Event classes

```
class AbstractEvent...
  private String name, code;
  public AbstractEvent(String name, String code) {
    this.name = name;
    this.code = code;
}
  public String getCode() { return code;}
  public String getName() { return name;}

public class Command extends AbstractEvent

public class Event extends AbstractEvent
```

The state classes

```
class State...
  private String name;
  private List<Command> actions =
    new ArrayList<Command>();
  private Map<String, Transition> transitions =
    new HashMap<String, Transition>();

class State...
  public void addTransition(Event event, State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(), new Transition(
      this, event, targetState));
}
```

The transition classes

```
class Transition...
  private final State source, target;
  private final Event trigger;

  public Transition(State source, Event trigger, State target) {
    this.source = source;
    this.target = target;
    this.trigger = trigger;
  }

  public State getSource() {return source;}
  public State getTarget() {return target;}
  public Event getTrigger() {return trigger;}
  public String getEventCode() {return trigger.getCode();}
```

start state

- 开始状态是一个状态机运行的起始点

```
class StateMachine...
  private State start;

  public StateMachine(State start) {
    this.start = start;
  }
```

- 状态机中的其他状态都是从起始状态可以到达的状态

```
class StateMachine...
  public Collection<State> getStates() {
    List<State> result = new ArrayList<State>();
    collectStates(result, start);
    return result;
  }
  private void collectStates(
      Collection<State> result, State s) {
    if (result.contains(s)) return;
    result.add(s);
    for (State next : s.getAllTargets())
      collectStates(result, next);
  }
class State...
  Collection<State> getAllTargets() {
    List<State> result = new ArrayList<State>();
    for (Transition t : transitions.values())
      result.add(t.getTarget());
    return result;
}
```

Programming Miss Grant's Controller

格兰特小姐密室控制器的构造过程如下：

1、首先把消息、命令、状态描述出来

```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");
```

2、基于前面构造的消息、命令、状态进一步构造控制过程（通过迁移将命令、状态等连接起来）

```
StateMachine machine = new StateMachine(idle);

idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened, waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);

waitingForLightState.addTransition(lightOn, unlockedPanelState);
waitingForDrawerState.addTransition(drawerOpened, unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);

machine.addResetEvents(doorOpened);
```

前后两部分代码之间的区别？

- On the one hand is the library, framework, or component implementation code;
- on the other is configuration or component assembly code.

或者，你也可以这样可以理解：

- 前面部分的代码SMM描述了怎么样建立一个状态机模型
  - 注意，这里的状态机模型

- 而后面部分的代码是定义了一个**具体的**控制器

你还可以这样可以理解：

- The State Machine Model
  - Domain Specific Language for State Machine
- Miss Grant's Controller
  - A kind of program of State Macine DSL
- 前者是一个语言，状态机建模语言，后置是用这个语言描述的”故事“，也就是格兰特小姐密室的具体控制规则。

特定领域语言

- 特定领域语言（Domain Specific Language）
  - 为了更好地描述特定领域的问题而专门设计的语言，其中主要针对该领域相关概念提供了在语言层面上的直接支持，即可以直接使用语言成分来描述领域概念

- 上例中的状态机模型其实就可以看做一种DSL

元语言和元编程

- 思考：
  - DSL在构造过程中是基于什么描述的？
  - 换句话说，描述DSL的语言是什么？

元编程

- 元编程（metaprogramming）
  - Metaprogramming is the writing of computer programs with the ability to treat programs as their data. [wikipedia]
  - 元编程是编写代码的代码[Rmeta]
    - （元编程是编写在**运行时**操纵语言构件的代码）
  - *目前，元编程并没有标准定义，以上是**两种**常见的对元编程的解释*

从编程角度来看

- *元编程是编写代码的代码*

- 《Ruby元编程》中的解释：
  - 元编程是编写在**运行时**操纵语言构件的代码
  - 这说明该书是聚焦在**运行时**这个维度上考虑

- *从这个角度来看，元编程主要是强调对语言概念层的实例进行动态运行时的创建和修改*

代码生成器

- 代码生成器和编译器
  - 从广义上讲，根据特定的规则来产生某种用途的完整代码或代码片段，甚至是配置性的文件，都可以认为是代码生成
  - 编译器也属于一种专门的代码生成器，但编译器主要聚焦在为特定编程语言生成特定体系结构上的可执行指令

代码生成与元编程

- 从元编程的角度来看待代码生成
  - 代码生成可以使用一种编程语言创建另一种语言的代码（源码），但这都是在静态期间
  - 此时，元语言只是将目标语言简单地视为普通数据（文本或字符串）
  - 元语言和目标语言都不需要专门针对元编程的特殊机制支撑
  - 元编程最重要的是看待语言的方式，尤其是理解这个过程中，语言和描述该语言的元语言

元语言的相对性

- 元（meta）概念的作用
  - 在计算机世界中，“元”的概念处处存在
  - 元主要提供了对被描述事物在概念层上的定义或解释
- PL中的元语言
  - 从元编程的角度来看，元语言主要用于对目标语言的操纵
  - 从语言构造的角度来看，元语言是被构造语言的解释语言，主要用于提供对其概念模型以及语义的刻画

元语言两个层面的理解

- 从概念层面来看
  - 元语言提供一种**描述能力**，可以形式化或半形式化地刻画目标语言的**概念模型**以及定义其**语义**
  - *这个层面主要关注概念模型*
- 从实现层面来看
  - 元语言提供一种**处理能力**，可以静态生成目标语言的代码，或者可以动态修改目标语言程序的运行时元信息
  - *这个层面主要关注运行时的动态修改*

元语言两个层面的理解：

从概念层面来看待元语言

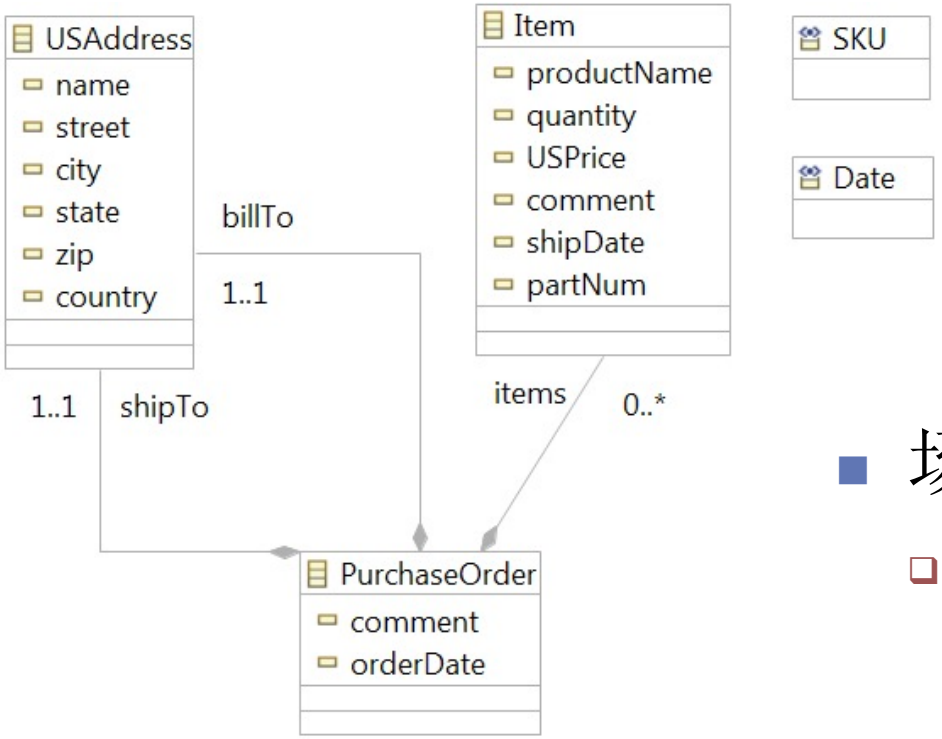
概念模型

- 概念模型（concept model）
  - 概念模型和元模型在很多情况下是混用的，所表示的内容也非常相似
  - 但概念模型主要强调刻画领域相关的一组概念，以及这些概念之间的关系
  - 元模型还可以用来描述抽象语法，以及相关约束等

示例

- 针对一个常见应用场景订单系统介绍如下概念：
  - 领域建模
  - 概念模型（元模型）
  - 用户模型
  - 元模型的相对性

对货物订单系统进行建模



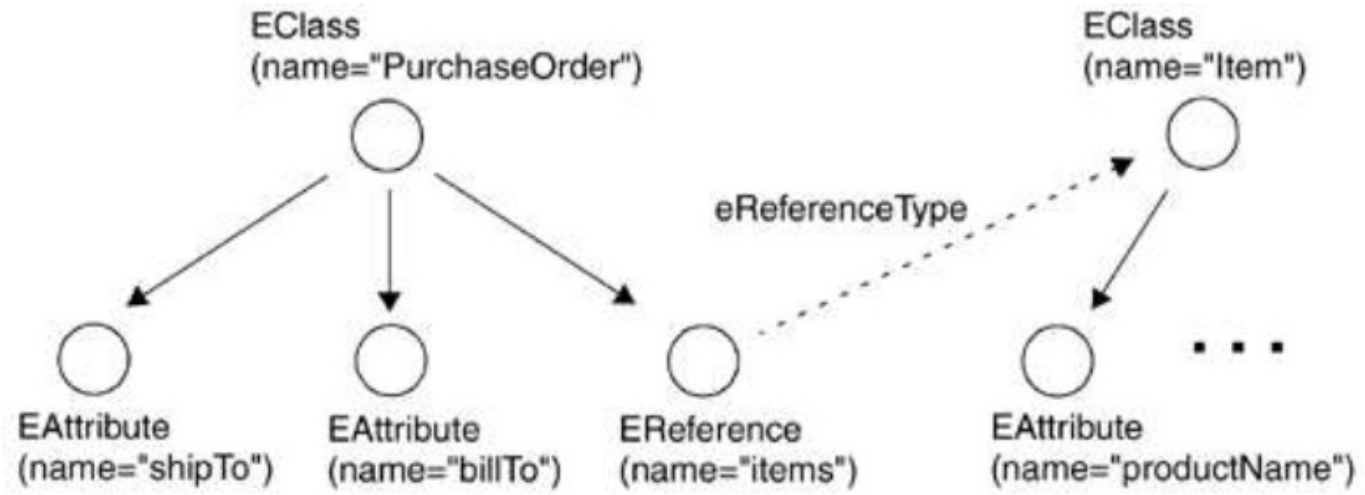
货物订单系统模型

- 场景描述
  - 订单包括如下信息：
    - 发货地址
    - 账单地址
    - 产品信息

订单实例

- 该订单系统的一个具体的实例：
  - AnAddress: theName, theStreet, theCity, theState, 1234567, theContry
  - APurchaseOrder: theComment, 2016.5.1
  - Item1: …
  - Item2: …
  - Item3: …

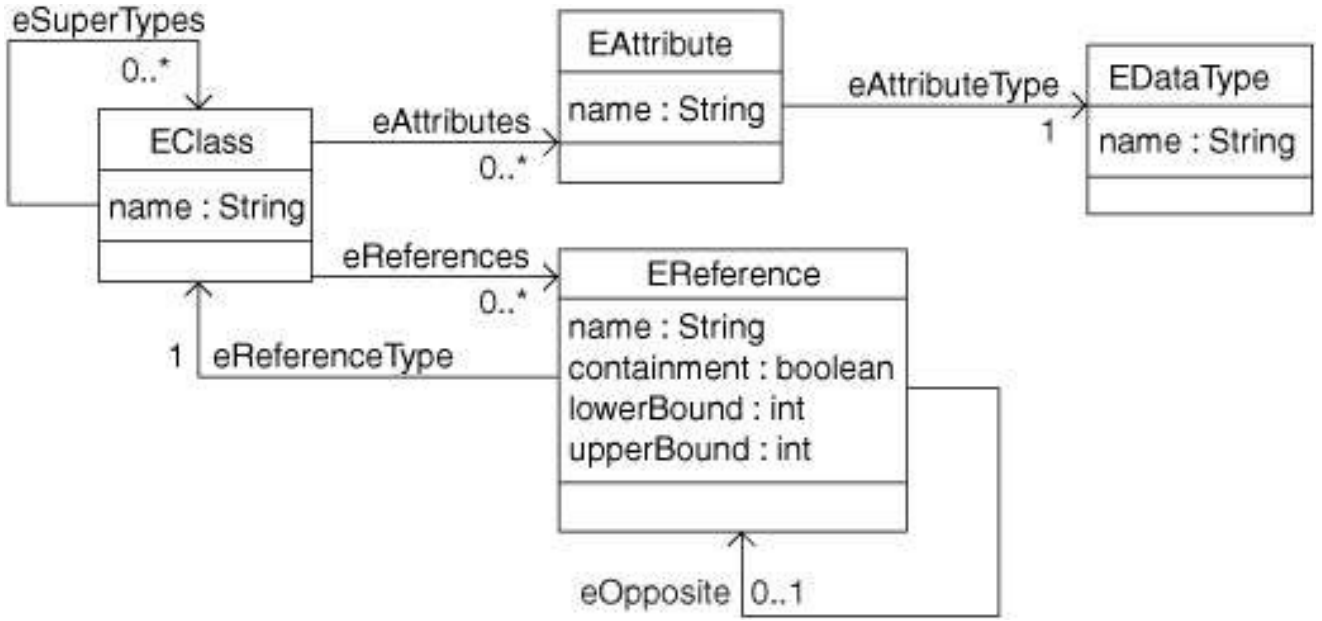
订单系统的元信息



- 这里所给出的元信息就是前面模型的语义相关信息，它解释了前面所出现模型元素的概念

所使用的元语言

- EMF Ecore Kernel



- 注：Ecore只定义了抽象语法和语义，并没有给出具体语法的定义，因此图中给出的图形表示形式均适用类似于UML的表现形式。

Ecore元模型

- Ecore元模型是Eclipse Modeling Framework (EMF)元模型体系的根
  - EMF是Eclipse平台官方支持的顶层项目，用于提供在Eclipse平台上进行建模的支撑
  - EMF是目前几乎所有Eclipse建模工具的基础
    - 如IBM Rational Rose、Topcased、Papyrus、Xtext等

领域模型

- 这里的领域是什么？领域模型是什么？
  - 可以从这样的视角来理解：即这里所讨论的问题有没有什么特定的领域概念？
  - 显然，这是一个和“订单”相关的场景，所使用的概念也是和“订单”场景相关的特定概念
  - 该订单模型就可以理解为是这样一类领域问题中的特定模型，也称为领域相关模型或领域模型

思考：

- 上例中的货物订单系统模型其实就可以看做一种DSL
  - 思考一下为什么？
  - 如果将其视为DSL的话，体会用该语言可以描述什么事情？

元语言

- 对于订单系统而言，Ecore建模语言就是其元语言
- 对于订单系统产生的实例（具体某张订单）而言，订单系统就是其元语言
- 思考：那么对于Ecore而言，什么语言会是定义它的语言，即它的元语言

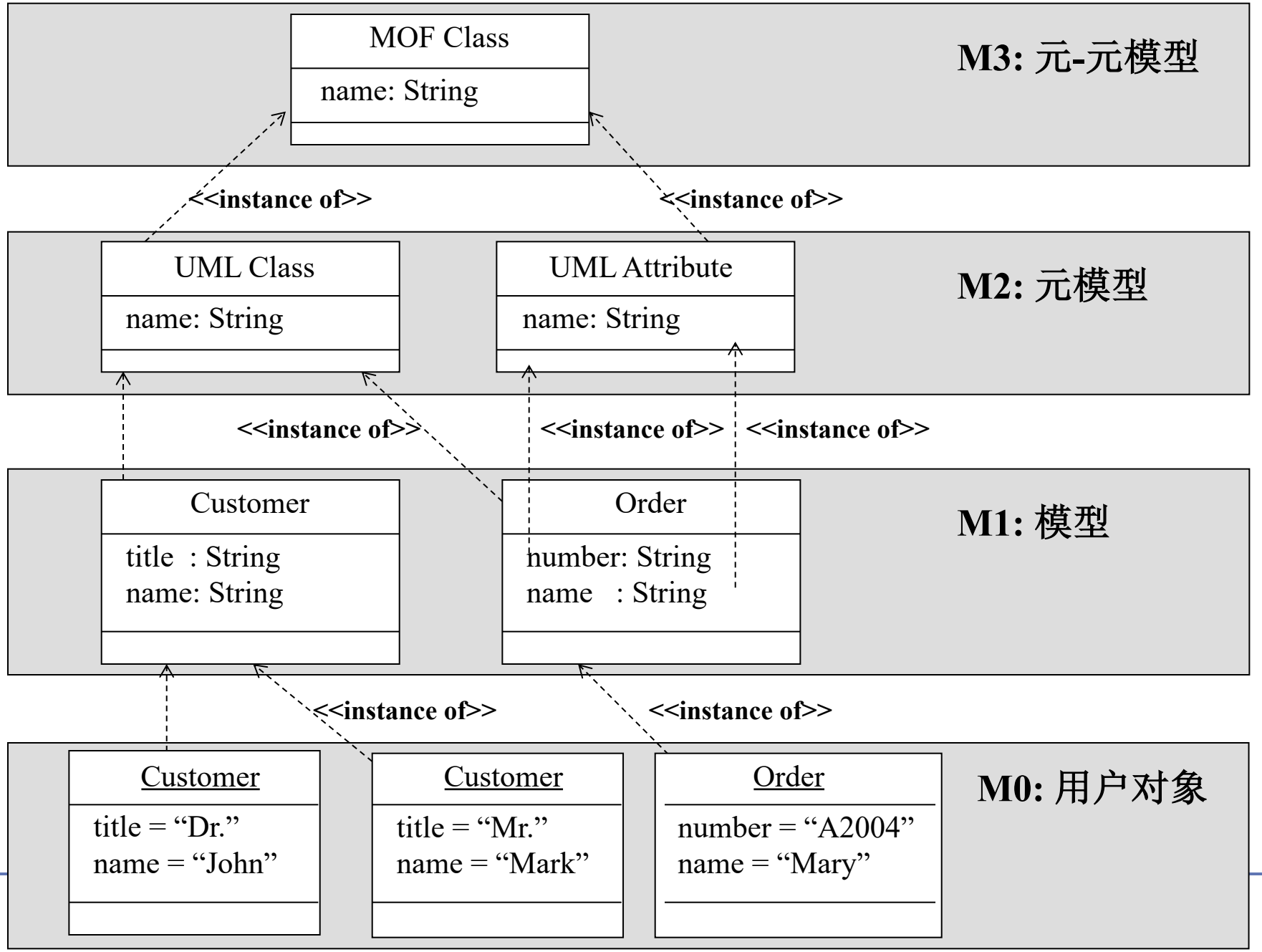
用户模型

- 用户模型是使用某种（建模）语言针对特定的业务逻辑所开发的系统模型
  - 因为是语言的使用者或用户所开发的，针对其业务所开发的模型，即所要开发的系统的模型，因此也称为用户模型
- 从Ecore 建模语言的角度来看，示例中的订单系统本身就是一个用户模型

示例：元模型体系的概念

- 4层元模型体系
  - MDA下的元模型体系分为4层，由MOF给出最终解释。

层 次	描 述	元 素
M3	MOF，即定义元素模型的构造集合	MOF类、 MOF属性、 MOF关联等
M2	元模型，由MOF构造的实例组成	UML类、 UML状态、 UML属性、 UML活动等
M1	模型，由M2元模型构造的实例组成	“Customer”类、 “Employee”表
M0	对象和数据，也即M1模型构造的实例	客户“张三”员工“李四”员工号“A2004”


MOF Class
name: String
M3: 元-元模型
<<instance of>>
<<instance of>>
UML Class
name: String
UML Attribute
name: String
M2: 元模型
<<instance of>>
<<instance of>>
<<instance of>>
Customer
title : String
name: String
Order
number: String
name : String
M1: 模型
<<instance of>>
<<instance of>>
Customer
title = “Dr.”
name = “John”
Customer
title = “Mr.”
name = “Mark”
Order
number = “A2004”
name = “Mary”
M0: 用户对象

元语言两个层面的理解：

从实现层面来看待元语言
重点关注动态元编程

动态元编程

- 动态元编程
  - 运行时的动态元编程需要语言层的支持
  - 运行时动态元编程开放给用户更多的灵活性，允许用户对**类型系统**做更多操纵（增、删、改）
  - 动态元编程一般都是语言针对自身提供的，所以元语言和目标语言都是自身

运行时的世界

- 语言是由各种称之为语言构件的元素构成的
  - 变量、数组、类、方法等
  - 对于语言来讲，这些就叫做语言的“元”（meta）
- 在很多编程语言中，语言的构件可以从源码中找到，但会消失在内存中，而另一些在不然
  - C++：一旦编译器完成了编译工作，像变量和方法这样的东西就看不见了，它们只是内存位置而已。你无法向类询问它的实例方法，因为此时它已不在了
  - Ruby：运行时绝大多数语言构件依然存在，你可以直接通过内省（introspection）机制获取其元信息

语言的支持

- 编程语言对动态元编程的支持可以概括为两个层次：
  - 只能在动态期间获取元信息，而不能加以修改（只读模式）
  - 可以动态获取元信息，而且可以动态的修改（读写模式）

  - *注：目前动态元编程最主要的应用是在面向对象编程语言中对类的操纵（思考一下为什么）*

Java中的元编程

- Java是一种静态语言，对元编程的支持主要是只读模式
  - Java提供了反射机制（或自省机制），但该机制主要以动态运行时获取类的属性、方法等为目的
  - 思考：Java如果要对其反射机制提供更大的权限，能否直接在reflect包中直接增加相关API即可？

  - *注：以Java为代表的同一类静态语言都具有相同的元编程特点*

重点以Ruby语言为具体的实例

在20世纪90年代由日本人松本行弘(Yukihiro Matsumoto)开发的一种动态面向对象语言。

Ruby元模型

- Ruby的对象模型与元模型
  - Ruby的对象模型与元模型有什么不同？
  - Ruby对象模型强调在运行时各种语言构件（language constructs）及其元信息解释，这里的元信息解释主要是指元模型
  - 元模型往往会隐含的包括**元元模型体系**的概念
- Ruby的元模型
  - 从Programming的角度来看，Ruby程序的元模型是什么样子的？
  - 从Language的角度来看，Ruby语言的元模型是什么（提示，Ruby的元元模型）？

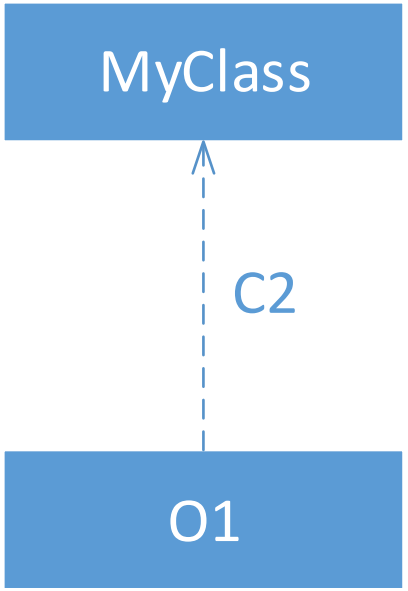
Ruby元模型

- 从程序的角度理解Ruby元模型
  - 动态语言Ruby的一大优点就是交互式过程中对元信息的反射机制开放度很高
  - 可以通过交互式环境动态查看Ruby程序级元模型

程序级元模型

- 自己定义类并创建对象
  - 打开irb来跟Ruby交互吧：$ irb
  - irb(main):001:0> **class MyClass; end**　　# => **nil**
  - irb(main):003:0> **o1 = MyClass.new**
    # => **#<MyClass:0x007fb3fb8a5938>**
  - irb(main):005:0> **o1.class**　　# => **MyClass**
  - MyClass是用户定义的一个“类”，o1是用户创建的一个MyClass的“对象”

MyClass

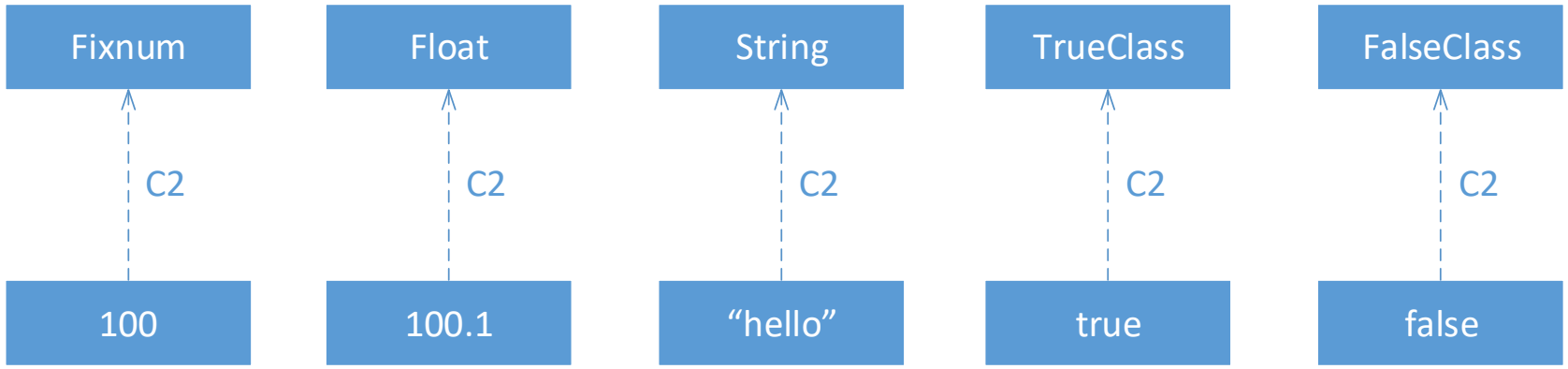


- 用户模型
  - MyClass是用在自己创建的类型
  - o1是MyClass类型的实例或对象
- *思考：在Ruby中的标量类型是什么样子的，做个试验试试看*

Ruby的标量类型

- 在Ruby中一切都是“对象”
  - 打开irb：$ irb
  - irb(main):001:0> **100.class** # => **Fixnum**
  - irb(main):003:0> **100.1.class** # => **Float**
  - irb(main):005:0> **12345678901234567890.class** # => **Bignum**
  - irb(main):014:0> **"hello".class** # => **String**
  - irb(main):015:0> **true.class** # => **TrueClass**
  - irb(main):016:0> **false.class** # => **FalseClass**

标量类型的“类型”



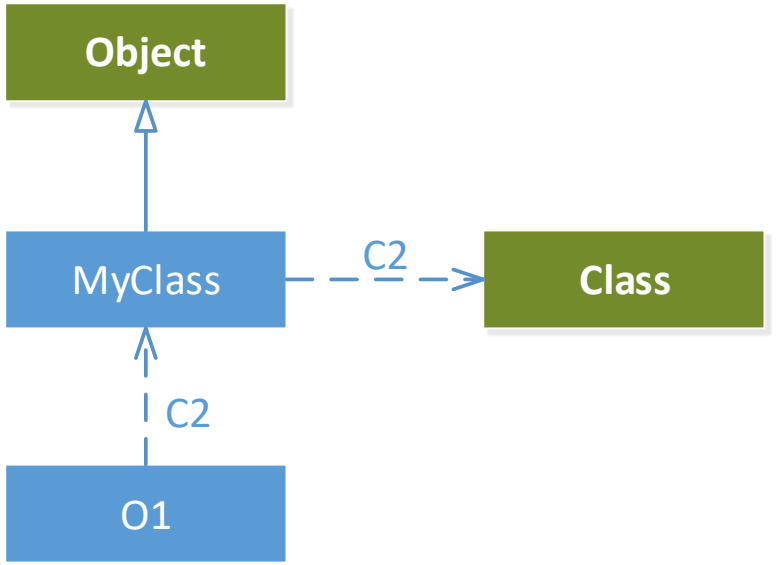
- 图中的下半部分是具体标量类型（实例）
- 上半部分是对应的Ruby标量类型模型（类型）

Ruby语言层

- 以上的两个示例MyClass和标量类型在元模型层次上是相同的，但在语言层次上是不同的
- **元模型层次上**
  - MyClass和标量类型都是用户模型层的类型
  - o1和100、100.1、”hello”、true、false等都是对应的实例
- **语言层次上**
  - MyClass并不是语言直接提供的构成部分（construct）
  - 但标量类型都是语言内置的构造部分（在语言元模型中直接提供）

MyClass的元模型

- 在irb中查看MyClass的元模型信息
  - irb(main):003:0> **MyClass.class** # => **Class**
  - irb(main):002:0> **MyClass.superclass** # => **Object**



标量类型的元模型

- 在irb中查看标量类型的元模型信息
  - irb(main):007:0> **String.class # => Class**
  - irb(main):008:0> **String.superclass # => Object**
  - irb(main):009:0> **Fixnum.class # => Class**
  - irb(main):010:0> **Fixnum.superclass # => Object**
  - irb(main):011:0> **Float.class# => Class**
  - irb(main):012:0> **Float.superclass # => Object**
  - irb(main):013:0> **TrueClass.class # => Class**
  - irb(main):014:0> **TrueClass.superclass # => Object**

练习和课后作业：

- 尝试给出元模型的图形化描述

Ruby元模型

- 对Object和Class进一步查看元信息
  - irb(main):006:0> **Class.class** # => **Class**
  - irb(main):007:0> Class.superclass # => Module
  - irb(main):008:0> Module.class # => Class
  - irb(main):009:0> Module.superclass # => Object
  - irb(main):010:0> Object.class # => Class
  - irb(main):011:0> Object.superclass # => BasicObject
  - irb(main):013:0> BasicObject.class # => Class
  - irb(main):014:0> BasicObject.superclass # => nil

Ruby元操作

- 列出类中的方法和属性
  - 按public和private分别列出
  - 区分类变量和实例变量
- 打开一个已有的类进行修改
  - 对现有方法进行增删改
- 修改一个已有的对象
  - 任意修改实例变量，允许同一个类的对象拥有不同实例变量
- 动态执行代码字符串

附录：交互式开发环境irb

- Ruby提供了用于交互式开发的shell方式环境
  - 以Mac为例，在Terminal中键入命令：irb
  - 提示符：**irb(main):001:0>**
  - 装载一个已经写好的ruby程序：load ‘xxx.rb’
    - 注意：应该从xxx.rb程序所在的目录启动irb交互shell

irb元命令

- 查看“类型”：
  - “hello”.class# String
  - String.class
  - ……

阅读资料

1. Python学习手册（第4版），Mark Lutz，机械工业出版社，2011
2. Ruby元编程（第2版），Paolo Perrotta，华中科技大学出版社，2015
3. 松本行弘的程序世界， 松本行弘，人民邮电出版社，2011
4. 编程的修炼，Edsger W. Dijkstra，电子工业出版社，2013
5. Programming Ruby (4th), Dave Thomas et al, The Pragmatic Programmers, 2013