第三章　软件架构模型

王璐璐 wanglulu@seu.edu.cn
廖力　lliao@seu.edu.cn

第3章 软件架构模型

3.1 引言

3.2 软件架构的可视化建模方法

3.3 软件架构的形式化建模方法

3.4 其他建模方法

3.5 软件架构建模方法的发展趋势分析

3.1 引言

- 软件架构模型
  - 软件架构建模是对架构设计决策的具象化和文档化。
  - 软件架构建模的意义在于它能够将软件架构某些关键或关注的方面剥离出来，使用统一的图形、文档和数据进行描述，达到直观、便捷地理解、分析和交流。

3.1 引言

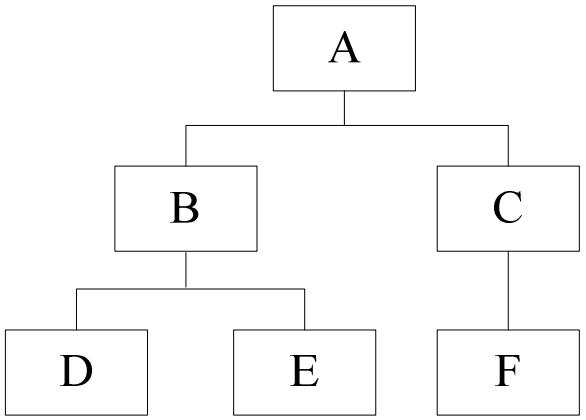
- 软件架构建模先后出现了五类方法：
  - 基于非规范的图形表示的建模方法
  - 基于UML的建模方法
  - 基于形式化的建模方法
  - 基于UML形式化的方法
  - 其他建模方法

3.2 软件架构的可视化建模方法

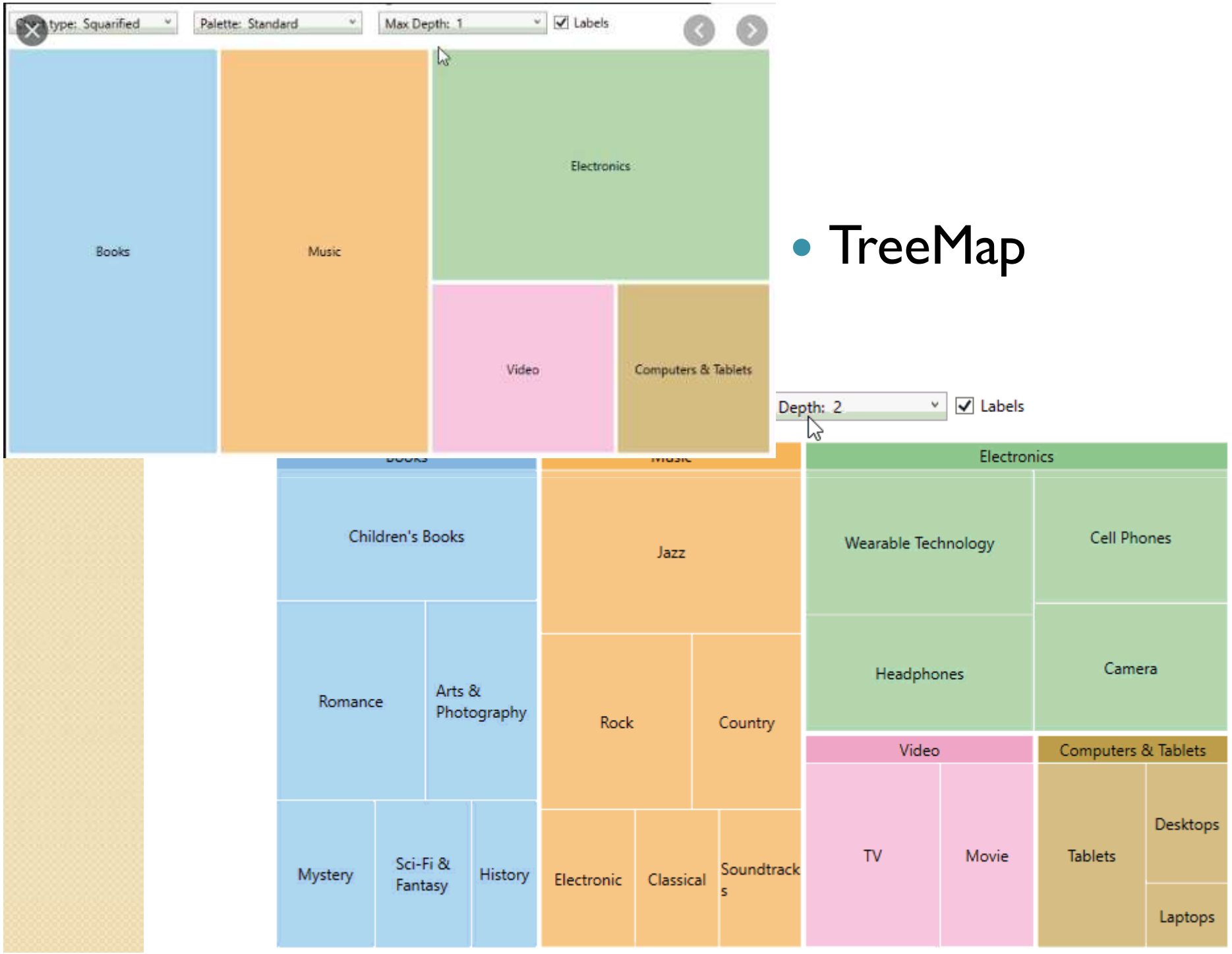
- 3.2.1 基于图形可视化建模方法
  - 图形可视化是将软件架构按照图形的方式进行表达，需要便于涉众阅读、理解和交流，使之不会因图形过于复杂而难以把握软件架构的概况。
  - 该方法分为两类
    - 正式图形表示：有严格定义的结构。
    - 非正式图形表示：不具有严格的标准，较为随意，具有一定方便交流的作用。如：盒线图（Box-Line Diagram）、PowerPoint风格图形等。

3.2 软件架构的可视化建模方法

- 3.2.1基于图形可视化建模方法
  - 正式图形表示：
    - 树形结构
    - 树地图（TreeMap）
    - 改进的树地图
    - 冰块图（Icicle Plot）
    - 旭日图（Sunburst）
    - 双曲树（Hyperbolic Tree）

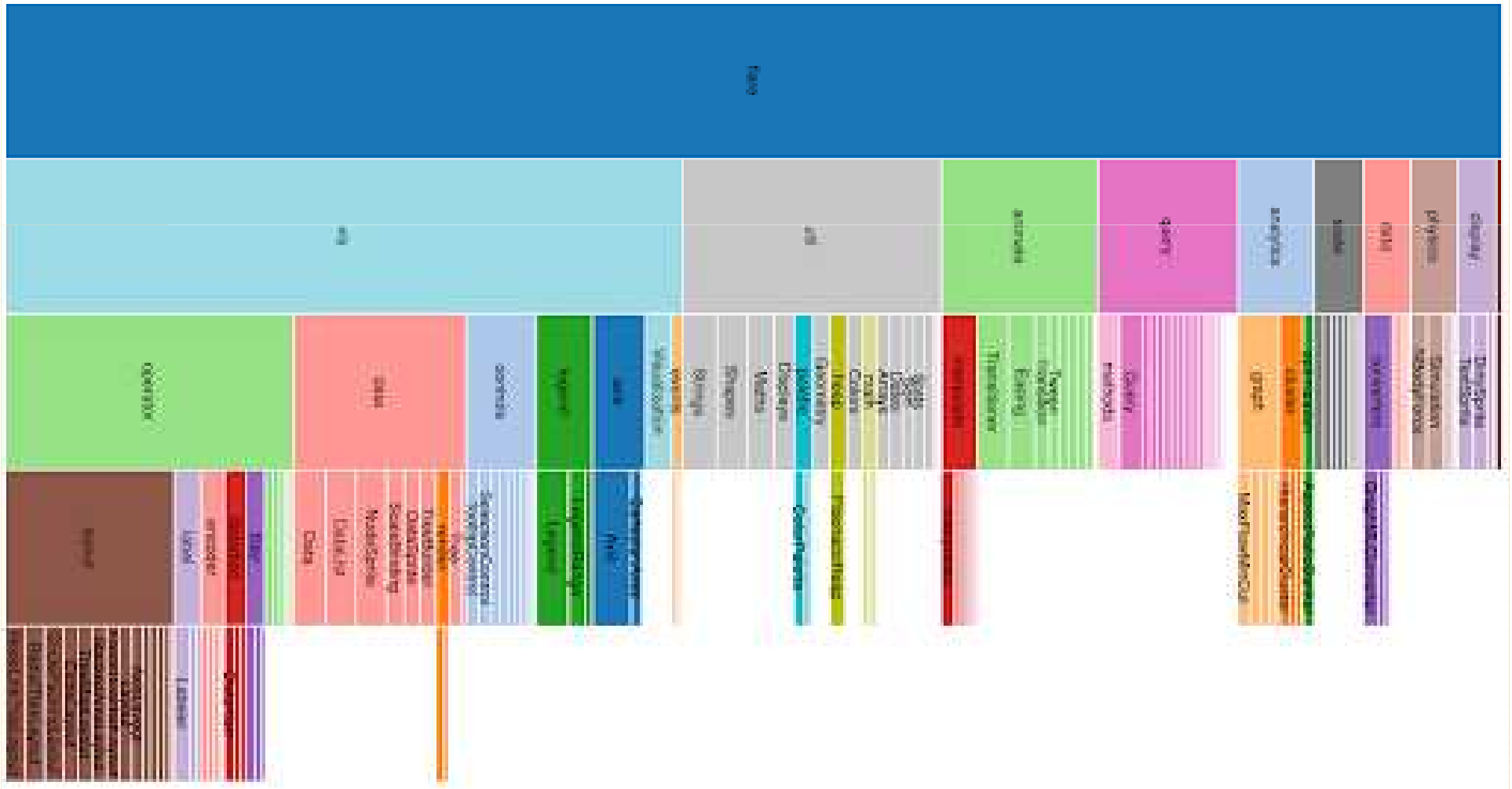


树形结构


type: Squarified
Palette: Standard
Max Depth: 1
Labels
Books
Music
Electronics
Video
Computers & Tablets
Depth: 2
Labels
Children's Books
Romance
Arts & Photography
Mystery
Sci-Fi & Fantasy
History
Jazz
Rock
Country
Electronic
Classical
Soundtracks
Electronics
Wearable Technology
Cell Phones
Headphones
Camera
Video
TV
Movie
Computers & Tablets
Tablets
Desktops
Laptops

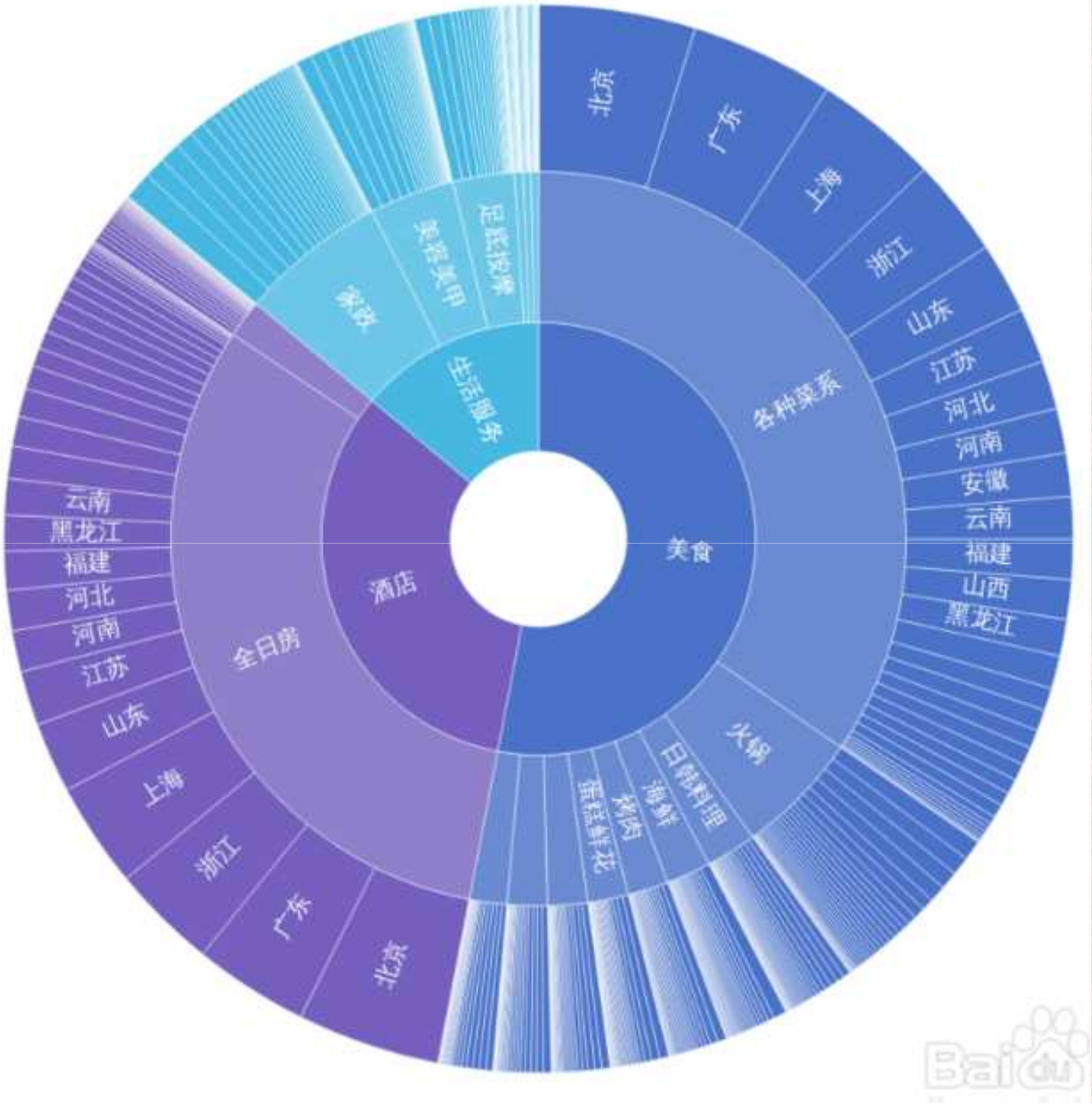
- TreeMap

- 冰块图（Icicle Plot）

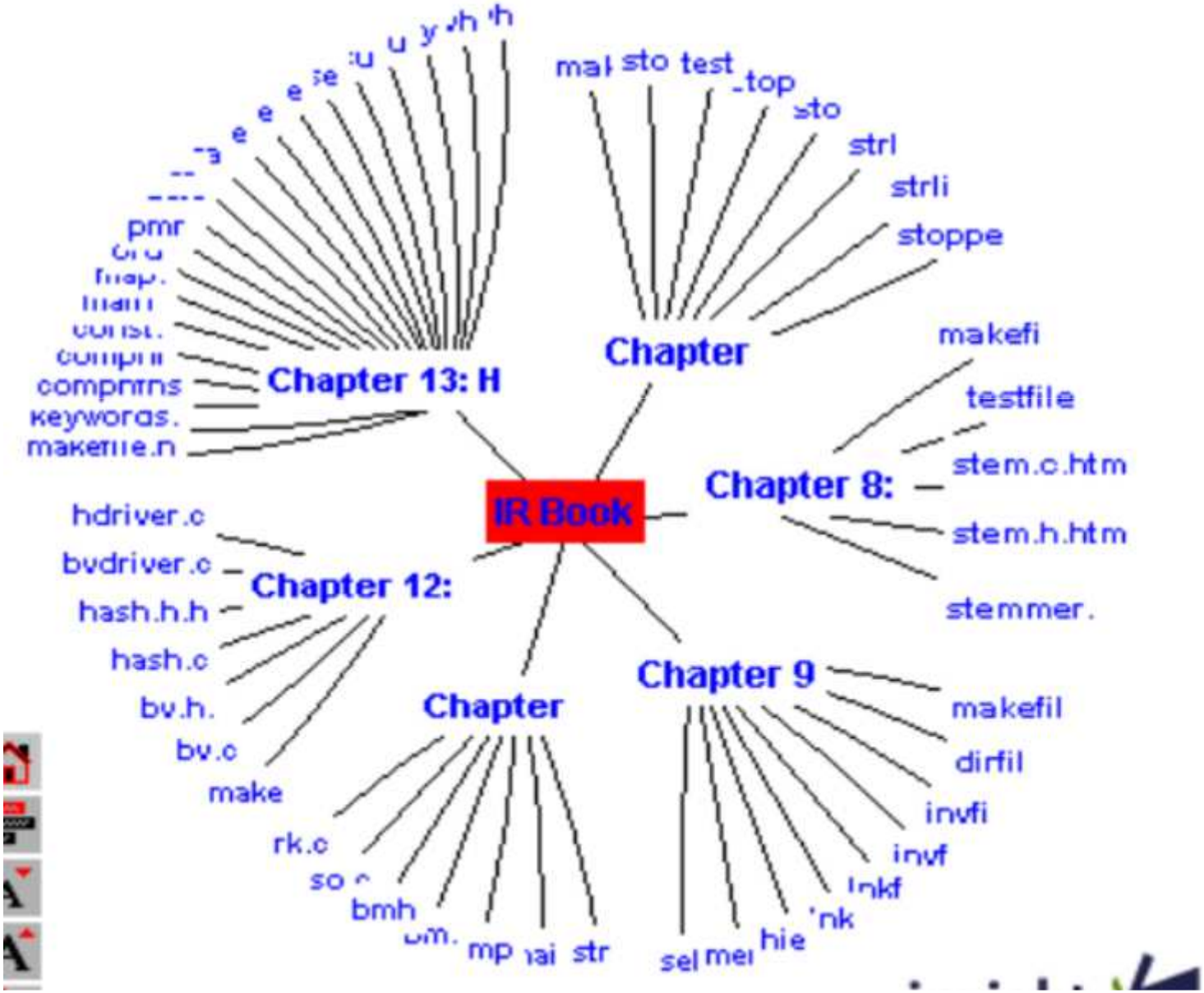


- 旭日图

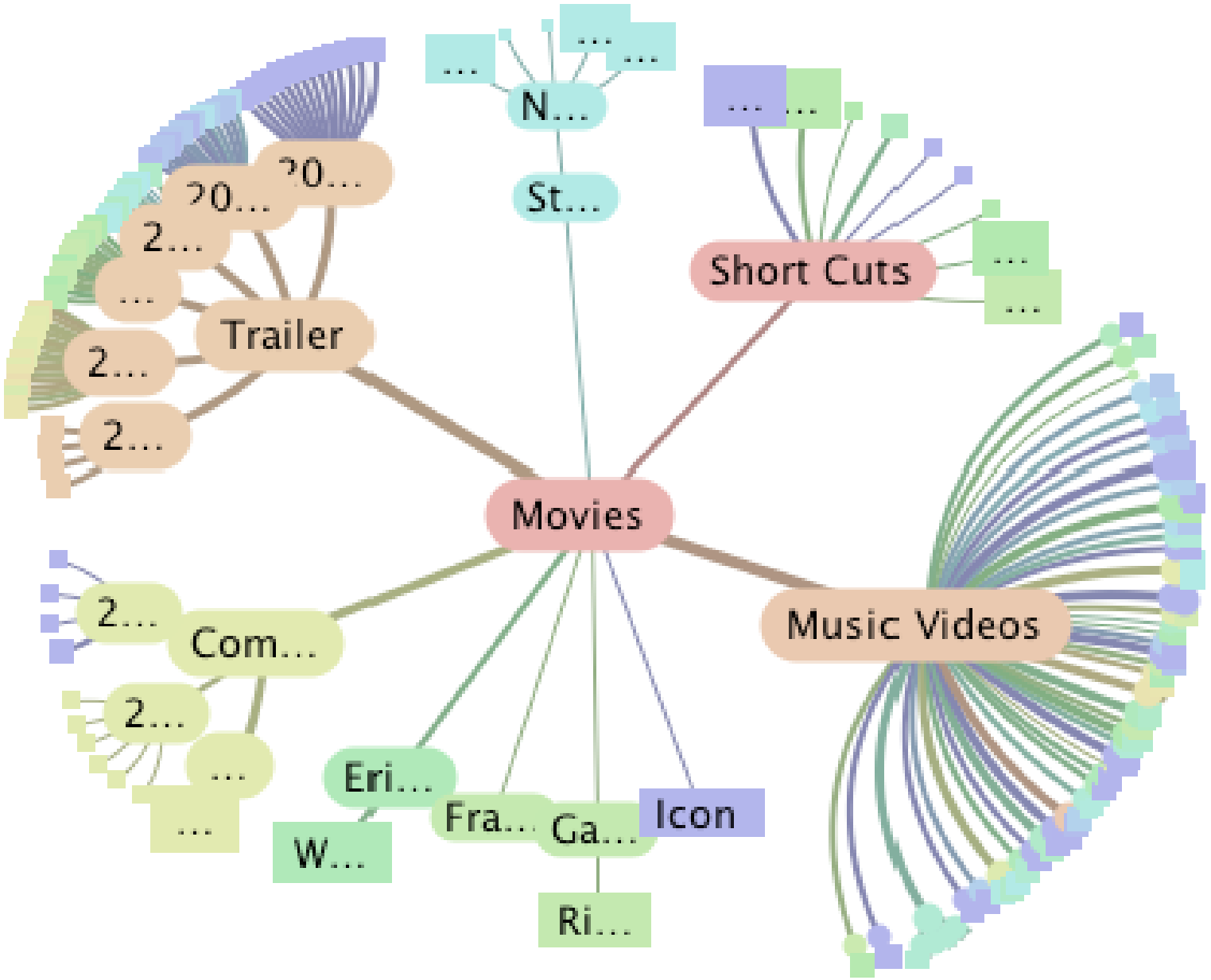
SunBurst



- 双曲树


IR Book
Chapter 13: H
pmr
Chapter
strl
strli
stoppe
Chapter 8:
makefi
testfile
stem.c.htm
stem.h.htm
stemmer.
Chapter 12:
hdriver.c
bvdriver.c
hash.h.h
hash.c
bv.h.
bv.c
make
Chapter
bmh
Chapter 9
makefil
dirfil
invfi
invf

- 双曲树


Movies
Trailer
Short Cuts
Music Videos
Com...
N...
St...
Eri...
Fra...
Ga...
Icon
W...
Ri...

3.2 软件架构的可视化建模方法

- 3.2.2 基于UML的建模方法
  - UML定义了一组丰富的模型元素以建模组件、接口、关系和约束。
  - UML 作为一个工业化标准的可视化建模语言，支持多角度、多层次、多方面的建模需求，支持扩展，并有强大的工具支持。
  - UML的最初意图是软件的细节设计，并不是专门为了描述软件架构而提出的。
  - 对于大部分架构构造，在UML中都可以找到相应的元素与之相对应。因此可以把UML看作一种架构建模语言。

3.2 软件架构的可视化建模方法

- 3.2.2 基于UML的建模方法
  - UML提供了对架构中组成要素建模的支持

架构元素	UML模型元素
组件	分类器（如类、组件、节点、用例等）
接口	接口
关系（连接器）	关系（如泛化、关联、依赖等）
约束（规则）	规则

3.2.2 基于UML的建模方法

- 为了降低架构建模的复杂度，软件设计人员可以利用UML从多个不同的视角来描述软件架构
  - 逻辑视图 关注功能，可以采用UML用例图来实现
  - 开发视图 关注软件的静态组织结构，使用UML的类图、对象图和组件图来表示模块，用包来表示子系统，利用连接表示模块或子系统之间的关联
  - 过程视图 关注软件的流程特征，可以采用UML的状态图、顺序图和活动图来实现
  - 物理视图 关注功能单元的分布状况，可以使用UML的配置图来实现
  - 场景视图 关注用户需求，可以使用用例 或场景来描述

3.2.2 基于UML的建模方法

- 使用UML 表示软件架构的具体实现如下：
  - 软件架构仅由其组件元素构成；
  - 组件具有两个标记值（Tagged value）： kindofComponent 表示它是原子组件或组合组件，而sub-Components表示它是子组件；
  - 组件只能通过Port与其他连接件相关联，而不能直接与其他组件相关联；每一个组件都有一个或多个Port；一个Port 至多只能与一个连接件关联；
  - 每一个连接件至少与两个组件相连，且组件和连接件不参加其语义范围以外的任何关联；
  - 组合组件的子组件只能是由连接件连接起来的组合组件或原子组件；
  - 原子组件不能再包含其他组件（原子组件或子组件）；

3.2.2 基于UML的建模方法

- UML对软件架构建模的主要优点有
  - 通用的模型表示法、统一的标准，便于理解和交流；
  - 支持多视图结构，能够从不同角度来刻画软件架构，可以有效地用于分析、设计和实现过程；
  - 有效利用模型操作工具(支持UML 的工具集)，可缩短开发周期， 提高开发效率;
  - 统一的交叉引用(cross-referencing) 模型信息的方法， 有利于维护开发元素的可处理性， 避免错误的产生。

3.2.2 基于UML的建模方法

- UML 虽然可以对软件架构进行较好的描述，但它只是针对特定的面向对象的架构，比如对架构缺少形式化的支持，使用UML建模存在着一些问题：
  - 对架构的构造性建模能力不强，还缺乏对架构风格和显式连接件的直接支持；
  - 虽然UML使用交互图、状态图和活动图描述系统行为，但语义的精确性不足；
  - 使用UML多视图建模产生信息冗余和不一致；
  - 对架构的建模只能到达非形式化的层次，不能保证软件开发过程的可靠性，不能充分表现软件架构的本质。

3.2.2 基于UML的建模方法

- 用UML对架构进行建模的三种途径
  - 将UML看作是一种软件架构描述语言直接对架构建模。这种方法简单易行，易理解，但有些UML结构无法和架构的概念直接对应起来，例如，连接件和软件架构风格在UML中无直接对应的元素，其对应关系必须由建模人员来维护。
  - 通过扩展机制约束UML的元模型以支持软件架构建模的需要。通过扩展机制增添新的结构而不改变现有的语法或语义，能显式地表示软件架构的约束，所建立的软件架构模型仍然可用标准的UML工具进行操纵。然而，对OCL约束进行检查的工具还不是很多。

3.2.2 基于UML的建模方法

- 用UML对架构进行建模的三种途径
  - 对UML的元模型进行扩充，增加架构建模元素，使其直接支持软件架构的概念。该方法使UML中包含各种 ADL 所具有的优良特性，并且具有直接支持软件架构建模的能力。然而，扩展的概念不符合UML标准，因而与UML工具不兼容。

3.2.2 基于UML的建模方法

- 将UML模型转换为架构模型步骤
  - 建立基于UML的应用领域模型。
  - 建立非形式化的架构图
    - 1）定义类接口，来表示架构中的主要组件元素——消息接口。
    - 2）定义连接件connectors类。每个连接件可以认为是一个简单类，将接收到的消息发送到合适的组件。
    - 3）建立表示架构风格的UML类图，主要描述类之间接口关系。这步主要是建立精化的类图。
    - 4）使用协作图描述类的实例，表达拓扑内容。

3.2.2 基于UML的建模方法

- 基于UML的性能模型
  - 性能模型又称作面向分析的模型，指为了进行性能或其它非功能性属性分析所建立的模型
  - 基于UML的性能模型的作用
    - 在软件开发早期对软件模型的性能进行研究
    - 对软件架构设计方案进行定量的预测和评价
    - 对各种设计决策进行比较，从而选择更优的软件架构设计方案

3.2.2 基于UML的建模方法

- 基于UML的性能模型
  - UML通常侧重于描述系统的功能性行为。
  - 为使UML模型能够描述系统性能需求，一种叫做SPT性能文档(UML profile for schedulability, performance and time)的扩展语言已经被OMG组织采纳并定为规范。
  - SPT性能文档通过子类型(stereotype)和标记值(tagged value)扩展了UML语言，以反映系统的性能需求。

3.2.2 基于UML的建模方法

- 基于UML的性能模型
  - 可以利用SPT性能文档在UML活动图中标记性能信息，比如：执行时间、访问频率或资源需求等；
  - 然后利用LQN模型求解工具对导出的性能模型求解，进而可以根据求解结果评价和指导系统设计。

3.3 软件架构的形式化建模方法

- 基于形式化规格说明语言的建模
  - 基本思想是利用一些已知特性的数学抽象来为目标软件系统的状态特征和行为特征构造模型。
- 基于UML形式化的建模：
  - 基本思想是利用形式化与UML结合的建模方法研究成果，对UML图形赋予形式化语义，然后就可以利用已有的形式化语言和工具对UML模型进行推理验证。

3.3.1基于形式化规格说明语言的建模

- Z语言
  - Z语言是迄今为止应用最为广泛的形式化语言之一。大型软件的开发中经常采用Z语言进行需求分析、软件架构建模。
  - Z语言是一种基于一阶谓词逻辑和集合论的形式规格说明语言，它采用了严格的数学理论，将函数、映射、关系等数学方法用于规格说明，具有精确、简洁、无二义性且可证明等优点。
  - 使用Z语言可以保证其书写的规格说明文档的正确性，同时还能保证有很好的可读性和可理解性。

过滤器模式

过滤器结构

端口定义

输入数据和输出不等

$$
\begin{aligned}
&\text{Filter}: \\
&[\text{filter\_id}: \text{FILER} \\
&\text{in}, \text{out}: \text{PORT} \\
&\text{in\_ports}, \text{out\_ports}: \mathbb{P}\ \text{PORT} \\
&\text{alphabet}: \text{PORT} \rightarrowtail \text{seq}\ \text{DATA} \\
&\text{transition}: \ \text{alphabet}(\text{in}) \rightarrowtail \text{alphabet}(\text{out}) \\
&\mid \text{in} \in \text{in\_ports} \land \text{out} \in \text{out\_ports} \lor \\
&\text{in\_ports} \cap \text{out\_ports} = \varnothing \ \lor \\
&\text{in\_ports} \cup \text{out\_ports} = \text{dom alphabet} \lor \\
&\forall \text{in\_port}: \ \text{PORT} \cdot \text{in\_port} \in \text{in\_ports} \land \\
&\text{dom transition}(\text{in\_port}) = \text{alphabet}(\text{in\_port}) \lor \\
&\forall \text{out\_port}: \ \text{PORT}; \ \exists \text{in\_port}: \text{PORT} \\
&\cdot \text{out\_port} \in \text{out\_ports} \land \text{in\_port} \in \text{in\_ports} \land \\
&\text{ran transition}(\text{in\_port}) \ = \text{alphabet}(\text{out\_port})]
\end{aligned}
$$

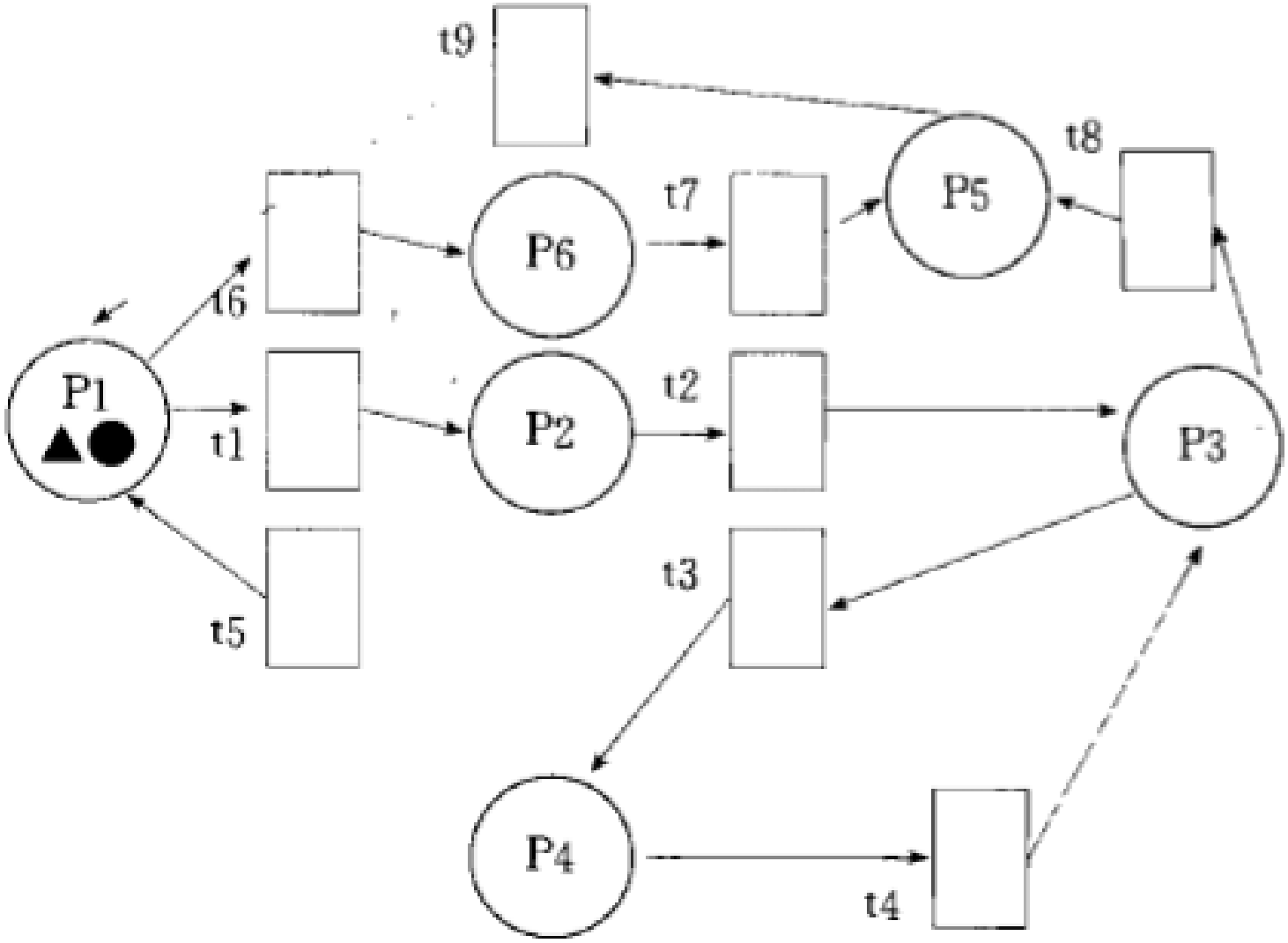
Transitions处理函数的定义域值域分别为输入数据和输出数据

端口映射函数alphabet

3.3.1基于形式化规格说明语言的建模

- Petri网
  - Petri网是一种系统的数学和图形的建模和分析工具，适用于对具有并发、同步、冲突等特点的系统进行模拟和分析，广泛应用于复杂系统的设计与分析。
  - 经典的软件架构的Petri网描述是一个四元组：（组件，连接件，角色，约束）。
  - Petri网形象地描述了软件架构的动态语义，通过变迁的发射Token从一个库所分配到另外一个库所，表明了资源或消息的传递，较好地说明了整个软件系统的流程。

- 简化物流系统



3.3.1基于形式化规格说明语言的建模

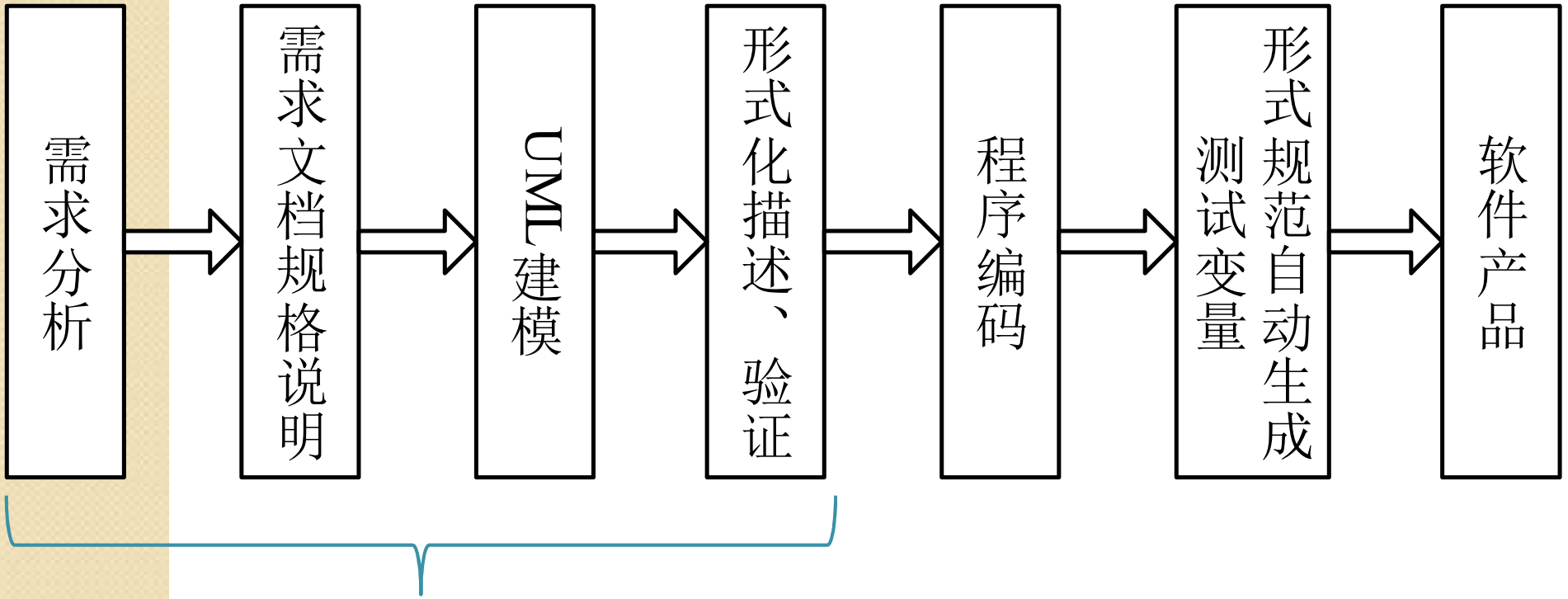
- B语言
  - 一种简单的伪码语言，描述需求模型、规格说明，并进行中间设计和实现。
- VDM
  - VDM是一种功能构造性规格说明技术，通过一阶谓词逻辑和已建立的抽象数据类型来描述每个运算或函数的功能。
- CSP
  - CSP，即基于进程代数的描述语言，它以进程和进程之间的关系描述为基础，用来描述一个复杂并发系统的动态交互行为特性。

3.3.2 基于UML形式化的建模方法

- UML不是一种形式化的语言
  - 尽管UML可以用于描述软件架构，对各种软件系统或离散型系统进行建模，并且通过相应支持工具的配合进行架构的文档化和部分目标语言代码的生成。然而，UML不是一种形式化的语言，不能精确的描述系统的运行语义。
  - 形式化方法和UML存在很大的互补性，二者的结合研究对提高软件架构的建模质量有着非常重要的意义。
  - 形式化与UML结合的建模过程和UML统一建模过程有明显的不同，它的目标是希望能够直接构造出尽可能正确的系统。

3.3.2 基于UML形式化的建模方法

- 形式化与UML结合的建模过程



3.3.2 基于UML形式化的建模方法

- UML的形式化
  - 类图的形式化
    - 类的形式化、关联形式化、泛化形式化
  - 用例图的形式化
    - 角色形式化、用例形式化、系统形式化
  - 状态图的形式化
    - 迁移形式化、状态形式化
  - 顺序图的形式化
    - 实例形式化、消息形式化

3.3.2.1 UML类图的形式化

- 类的形式化
  - 名称、属性和操作
  - 属性：
    - 名称、可见性、类型和多重性
  - 操作：
    - 名称、可见性
    - 参数操作的每个参数有名称和类型

3.3.2.1 UML类图的形式化

Attribute

name: Name
type: Type
visibility: VisibilityKind
Multiplicy: PN
initialValue: Expression

Parameter

name: Name
type: Type
defaultValue: Expression

类的属性和参数的形式化表示

Operation

name: Name
type: Type
Parameter: seq Parameter

$\forall p_1, p_2$: ran parameter
$p_1.name = p_2.name \Rightarrow p_1 = p_2$

类的操作的形式化表示

3.3.2.1 UML类图的形式化

Class

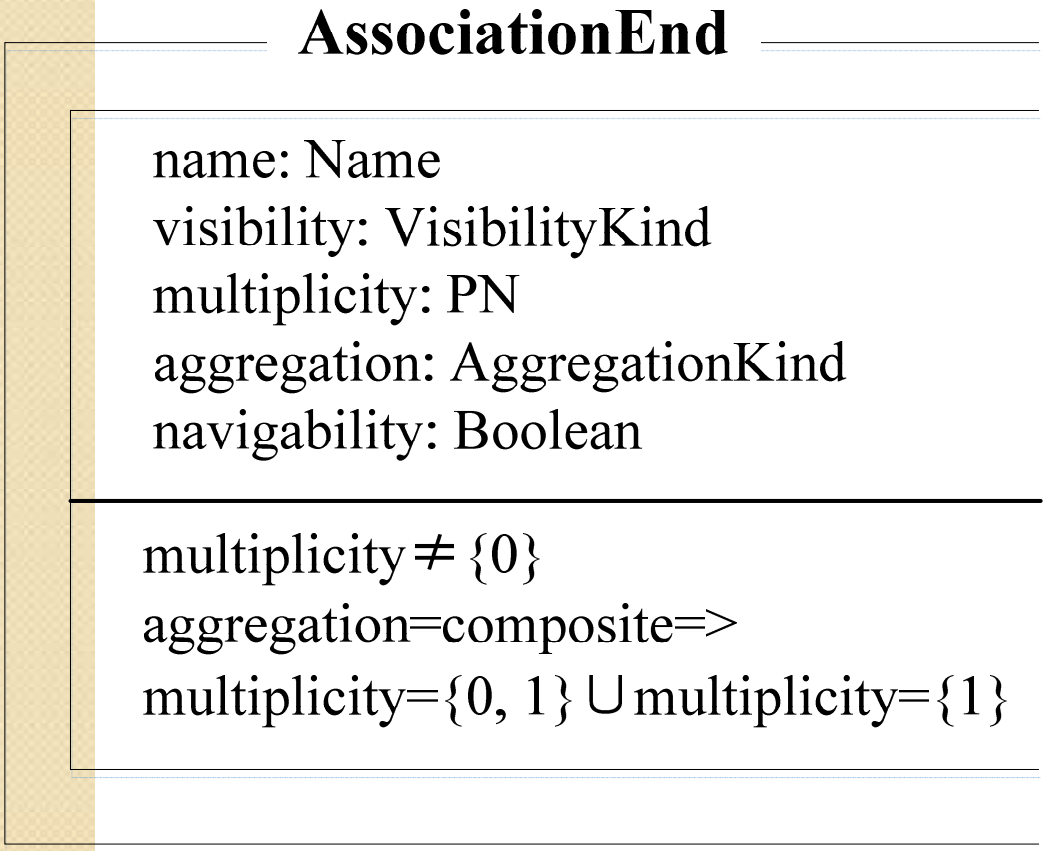
name: Name
attribute: F Attribute
operation: F Operation
isRoot: Boolean
isLeaf: Boolean
isAbstract: Boolean
Generalization, specialization: Name ->F Name

$\forall a_1, a_2$: attribute●
$a_1.name=a_2.name=>a_1=a_2$
$\forall op_1, op_2$: operation●
$(op_1.name=op_2.name \cap \#op_1.parameter=\#op_2.parameter \cap$
$\forall i$: 1...$\#op_1.parameter$●
$op_1.parameter(i).name=op_2.parameter(i).name \cap$
$op_1.parameter(i).type=op_2.parameter(i).type=>op_1=op_2$

类的形式化表示

3.3.2.1 UML类图的形式化

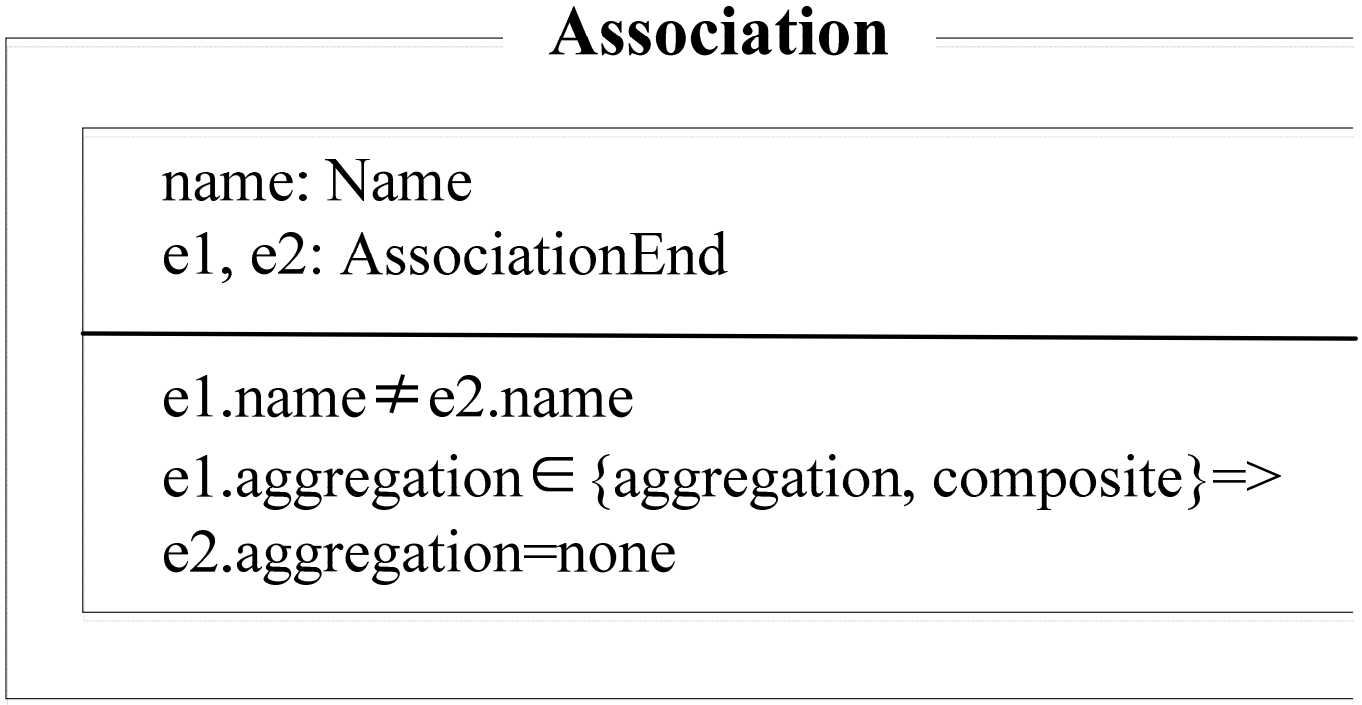
- 关联的形式化
  - 类之间的关系用关联来表示，通常是二元关联，有一个关联名和两个关联端



关联端的形式化表示

3.3.2.1 UML类图的形式化

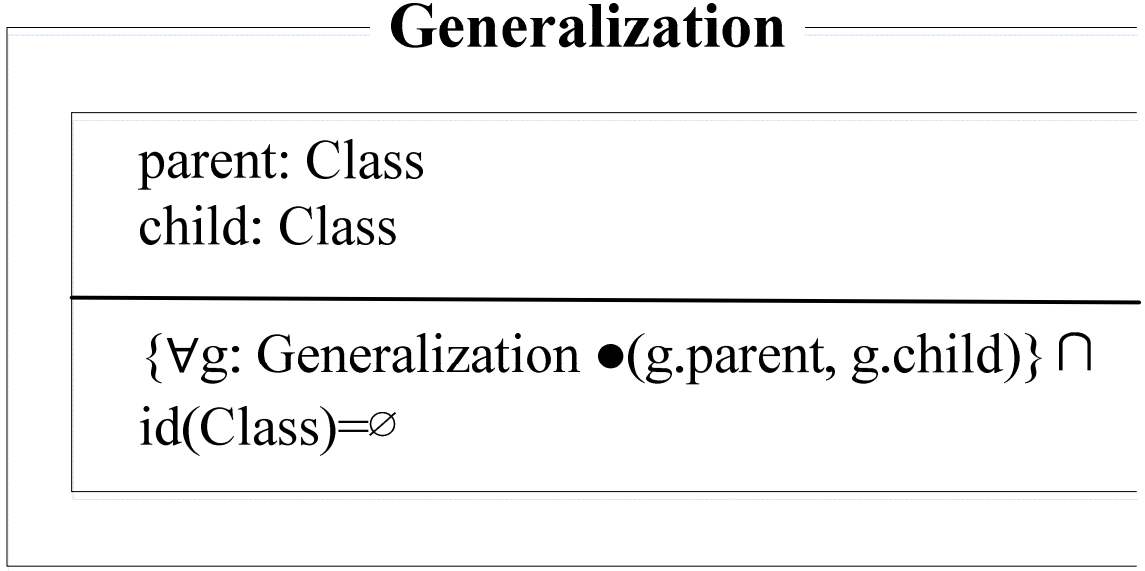
- 关联的形式化
  - 二元关联有一个名字并且恰好有两个关联端。



联系的形式化表示

3.3.2.1 UML类图的形式化

- 泛化的形式化
  - 泛化描述了对象之间的分类关系
  - 超类对象描述了通用的信息，子类对象描述了特定的信息。
  - 该约束表示不允许循环继承。



泛化的形式化表示

3.3.2.1 UML类图的形式化

- 类图的形式化
  - UML类图是由类、联系和泛化组成的。
  - 在类图中，类的名称是唯一的并且联系和泛化涉及到的类应该在同一个类图中。

ClassDiagram

class: F class
association: F Association
generalization: F Generalization

∀c1, c2:class ●c1.name=c2.name=>c1=c2
∪{g.generalization●{g.parent, g.child}}⊆Class
associationClass⊆class

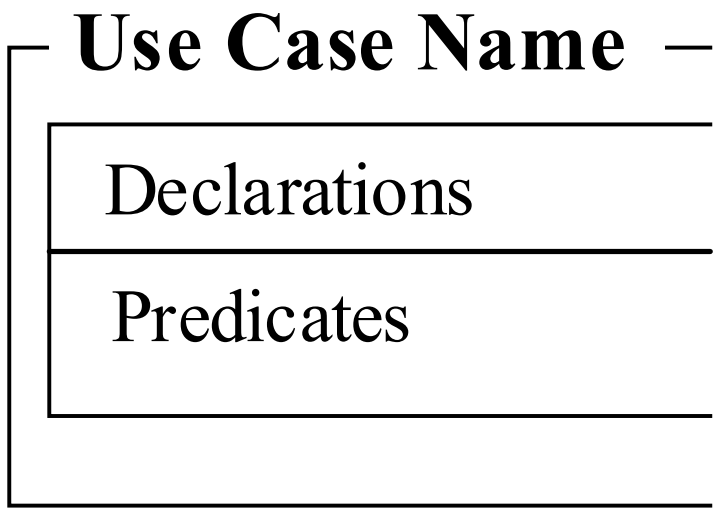
类图的形式化表示

3.3.2.2 UML用例图的形式化

- 角色的形式化描述
  - 角色是与系统交互的外部实体（人或事），代表的是一类能使用某个功能的人或事
  - 假设类A和类B使用系统的功能时具有角色C，则角色C可以表示成Actor C==A∪B。

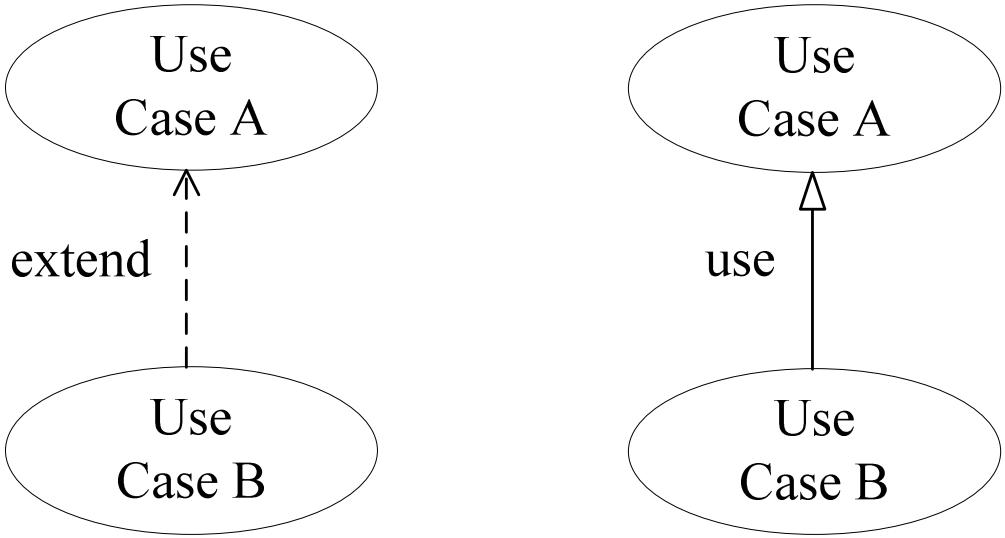
3.3.2.2 UML用例图的形式化

- 用例图的形式化描述
  - 用例图是由角色、用例和系统组成的
  - Declarations是声明部分，Predicates是谓词不变式部分。



用例图的形式化表示

3.3.2.2 UML用例图的形式化



具有扩展关系和使用关系的用例图

有关系的用例图的形式化表示

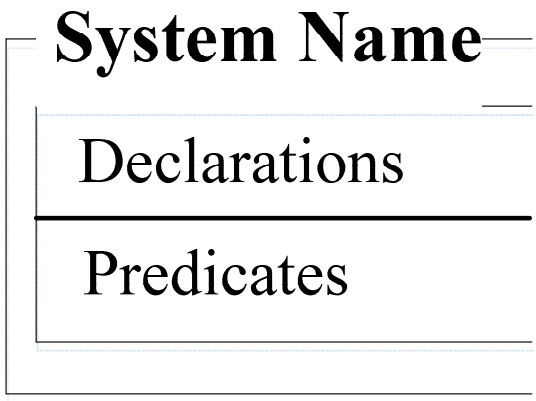
User Case Name(子)

User Case Name(父)
Declarations(子 Use Case)

Pre-Pred(子 Use Case)
Post-Pred(子 Use Case)

3.3.2.2 UML用例图的形式化

- 系统的形式化描述
  - System Name是系统的名字
  - Declarations是系统所提供的功能（即用例）的声明
  - Predicates是对Use Case的约束



系统的形式化表示

3.3.2.3 UML状态图的形式化

- 迁移模式
  - 迁移模式表征一个迁移所引起的状态变化，关联迁移前的变量值与迁移后的变量值。
  - 迁移具有触发事件、迁移条件、动作、源状态和目标状态。

Transfer

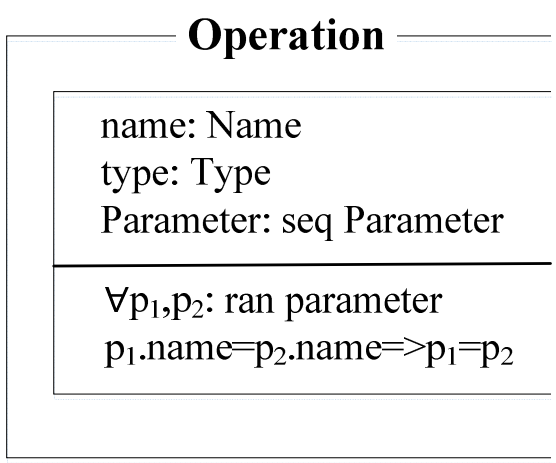
name: Name
E: EE
C: CE
A: AE
S, S: State
TriggerEE: P EE
Source_state, object_state: P_state

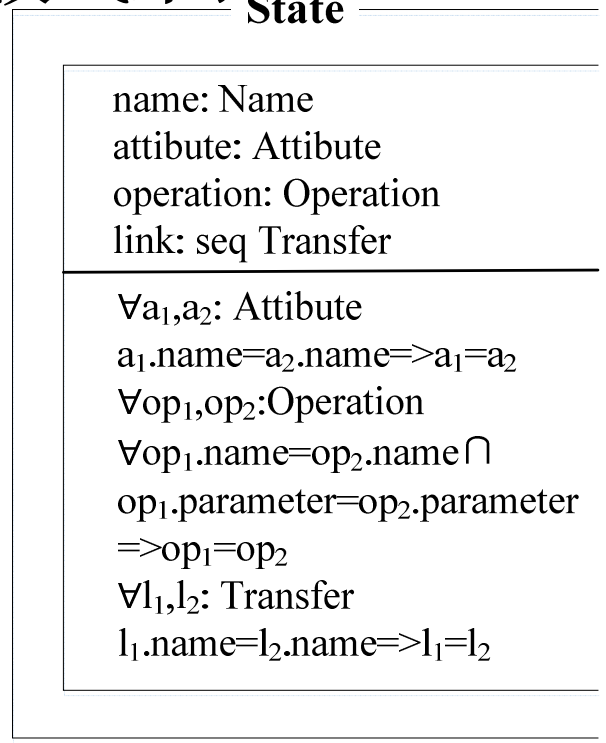
∀E:EE ∀C:CE∧∨S=Source_state
E∈TriggerEE∩C=True=>
Execute A∩S ∈object_state

状态迁移的形式化表示

3.3.2.3 UML状态图的形式化

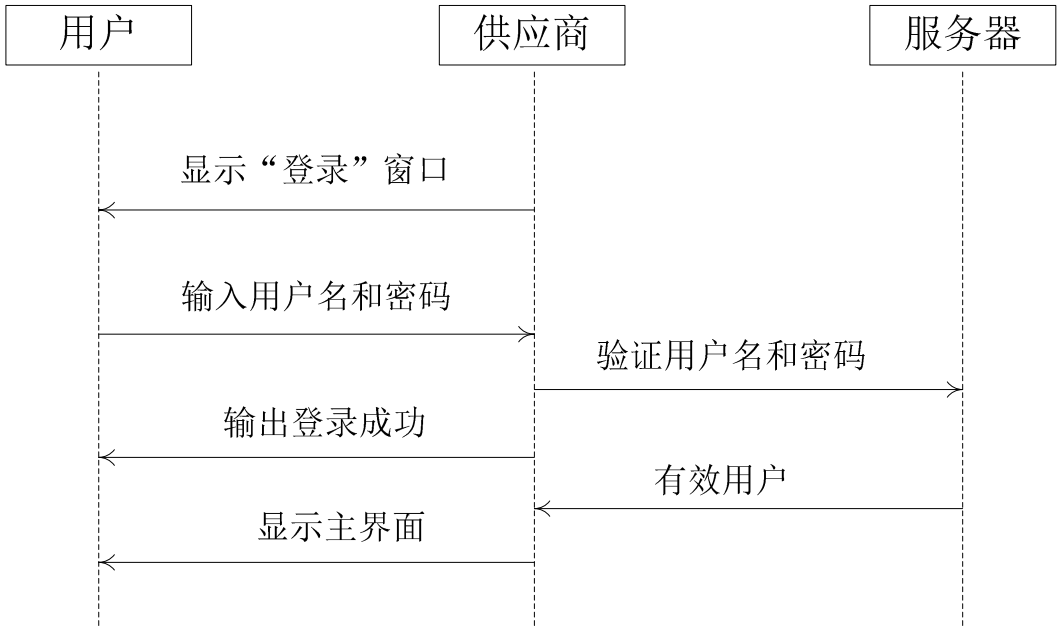
- 状态模式
  - 状态模式包含变量声明和使用声明变量的谓词集。
  - 动态属性由迁移模式描述，而静态属性则由属性模式和操作模式来描述





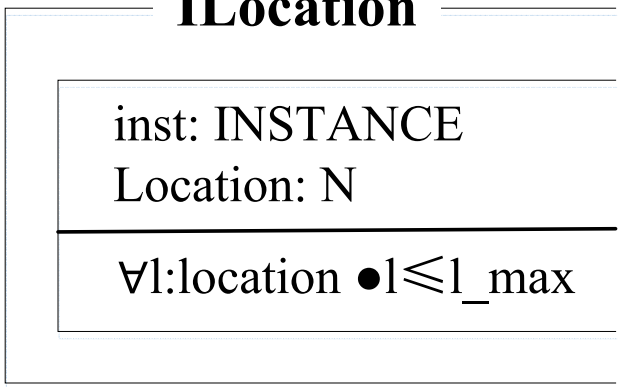
3.3.2.4 UML顺序图的形式化

- UML顺序图用于描述对象间动态的交互关系，着重体现对象间消息传递的时间顺序
- 水平轴表示不同的实例
- 垂直轴表示时间



3.3.2.4 UML顺序图的形式化

- 实例的形式化描述
  - 序列图具有一定的抽象句法，也必须满足一些合式规则
  - [INSTANCE, TYPE, NAME]
    - INSTANCE　所有实例的集合
    - TYPE　　所有类型的集合
    - NAME　　所有名称的集合
  - l _max，最大位置点。
  - 实例生命线上的位置点用模式ILocation 描述
  - 任意一个位置点均不大于l_max。



ILocation的形式化描述

3.3.2.4 UML顺序图的形式化

- 消息的形式化描述
  - 在每个位置点上可以发送消息给其他实例，也可以接受其他实例的消息
  - 消息标识用模式MsgId描述
  - 返回标识由模式ReturnId定义

MsgId

m_name: Name
paralist: seq(P TYPE*NAME)
returnv: TYPE
m_no: N

$\forall m_1, m_2 : m\_no \bullet m_1 \neq m_2$

MsgId的形式化描述

ReturnId

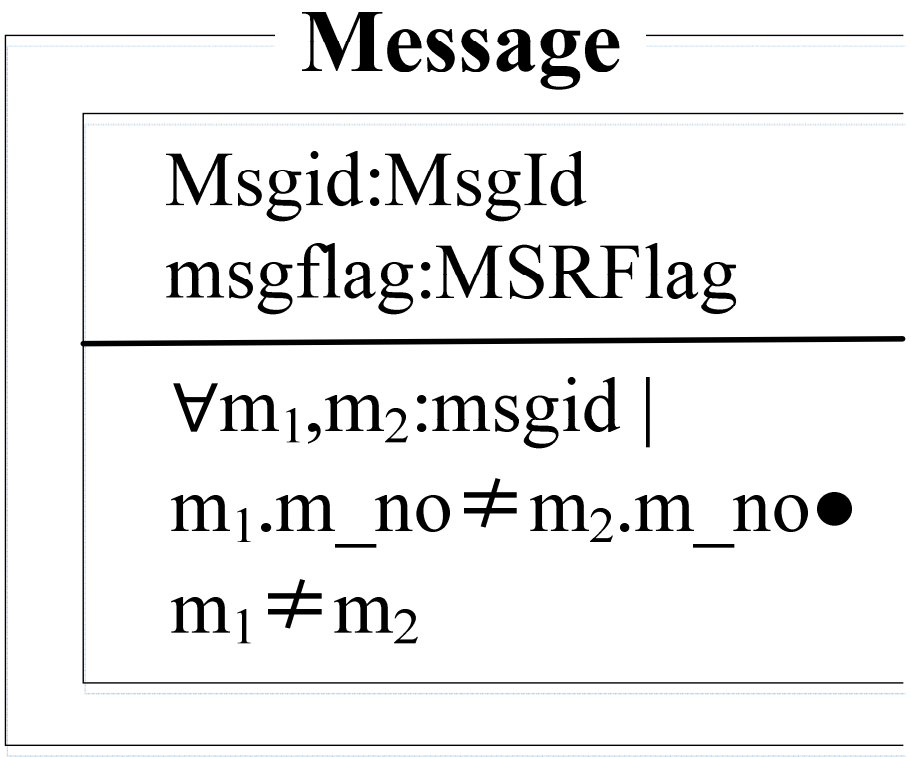
M_id:MsgId
returnv: TYPE

$\forall i_1, i_2 : m\_id \bullet i_1 \neq i_2$

ReturnId的形式化描述

3.3.2.4 UML顺序图的形式化

- 消息的形式化描述
  - 发送消息的位置点和接收消息的位置点用模式Message描述



Message的形式化描述

3.3.2.4 UML顺序图的形式化

- 消息的形式化描述
  - 合式的序列图用模式WFSequenceDiagram描述

WFSequenceDiagram

instance: F1 INSTANCE
iloc: F Ilocation
message: F message
msg: llocation <->F Message
Source, target: Message ->: F1 INSTANCE

#inst≥1
#message≥1
∀l:iloc; m_id:Msgld; u: message |
u=(m_id,S)∧u∈msg(l)●source(u)=l.inst
∀l:iloc;m_id:Msgld;u:message |
u=(m_id,R)∧u∈msg(l)●target(u)=l.inst
∀m:message●source(m)≠target(m)

3.3.2 基于UML形式化的建模方法

- UML与Z结合的建模过程是在目前软件规模和复杂性不断增大的情况下提出的，它是对现在工业界软件架构建模过程做了一些改进，并提出了一些新的思路和构想。
- 其他的形式化方法也可以将非形式化的UML图形转换为具有精确语义的形式化规范，在非形式化的图形表示与形式化定义之间建立映射关系。
- UML中的类图、用例图、状态图以及顺序图较适合形式化描述，其他的图用形式化的描述方法反而使得描述过程变得更加复杂，容易降低效率。

3.4 其他建模方法

- 文本语言建模方法
- 模型驱动的架构建模方法

3.4.1 文本语言建模方法

- 文本语言建模是通过文本文件描绘架构
  - 这些文本文件通常需要符合某些特殊的句法格式
- 可用方法：
  - 语法高亮显示：方便相关人员的阅读和编辑
  - 文本的静态检查：静态程序分析是指在不实际执行程序的情况下对计算机软件进行分析
  - 自动补全：有利于人机交互
  - 代码折叠：有利于开发人员管理源代码文件

XML文本建模方法

```
<instance: xArch  xsi: type = "instance: XArch">
   <types: archStructure  xsi: type = "types: ArchStructure"
                                types: id = "ClientArch">
      <types: description  xsi: type = "instance: Description">
         Client Architecture
      </types: description>
      <types: component  xsi: type = "types: Component"
                                types: id = "WebBrowser">
         <types: description  xsi: type = "instance: Description">
            Web Browser
         </types: description>
         <types: interface  xsi: type = "types: Interface"
                                 types: id = "WebBrowserInterface">
            <types: description  xsi: type = "instance: Description">
               Web Browser Interface
            </types: description>
            <types: direction  xsi: type = "instance: Direction">
               inout
            </types: direction>
         </types: interface>
      </types: component>
   </types: archStructure>
</instance: xArch>
```

- xADLite文本建模方法

```
xArch{
    archStructure{
      id = "ClientArch"
      description = "Client Architecture"
      component{
        id = "WebBrowser"
        description = "Web Browser"
        interface{
          id = "WebBrowserInterface"
          description = "Web Browser Interface"
          direction = "inout"
        }
      }
   }
}
```

3.4.1 文本语言建模方法

- 文本语言建模优势：
  - 单个文档中描述整体架构，并且存在众多文本编辑器方便用户与文本文档的交互；
  - 许多工具能够生成程序库来对使用该语言的文本文档进行句法分析和检查；
  - 许多编辑器附带额外的开发支持工具。
- 文本语言建模缺陷：
  - 用文本语言建模方法表示类图形结构就不易理解；
  - 文本编辑器通常限于显示连续满屏的文本，很难以另外的方式组织文本。

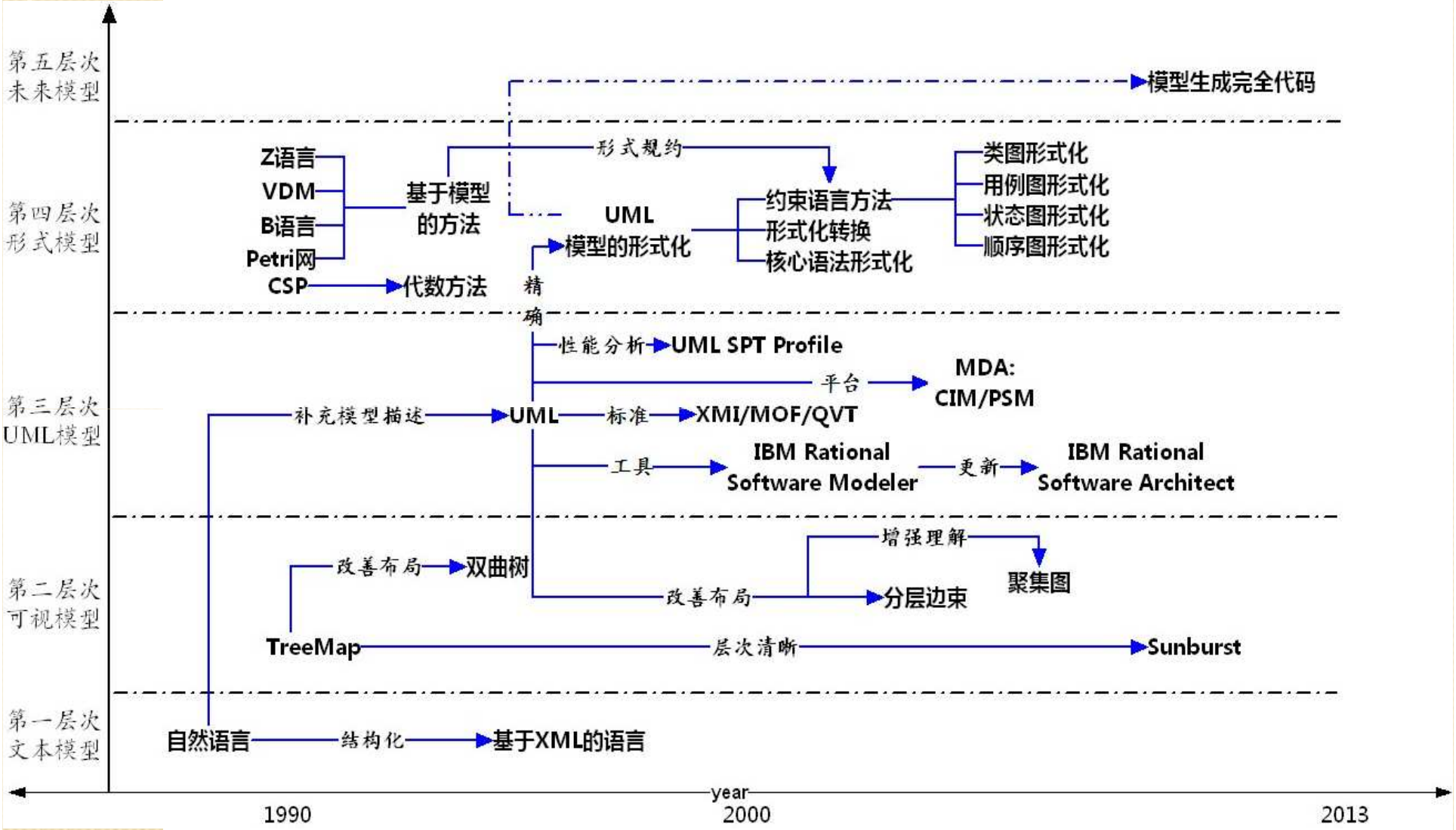
3.4.2 模型驱动的架构建模方法

- MDA（Model Driven Architecture, MDA）不是一个实现分布式系统的软件架构，而是一个模型技术进行软件开发的方法
- 模型驱动架构MDA，是OMG于2001年正式提出的一个框架规范。
- MDA致力于将软件开发从以代码为中心提高到以模型为中心，使模型不仅被作为设计文档和规格说明来使用，更成为一种能够自动转换为最终可运行系统的重要软件制品

3.4.2 模型驱动的架构建模方法

- 在MDA中，将模型区分为PIM和PSM
  - 平台无关模型（Platform Independent Model, PIM）
    - PIM是一个系统的形式化规范，它与具体的实现技术无关
  - 平台相关模型（Platform Specific Model, PSM）
    - PSM基于某一具体目标平台的形式化规范。模型不再仅仅是描绘系统、辅助沟通的工具，而是软件开发的核心和主干
- 它的核心思想是抽象出与实现技术无关、完整描述业务功能的平台独立模型。

3.5软件架构建模方法发展趋势分析



软件架构建模的技术发展路线

3.5软件架构建模方法发展趋势分析

- 第一层次：文本模型
  - 从原始的自然语言文档化到以XML为代表的有固定规范、结构的文档化
- 第二层次：图形可视化模型
  - 将软件架构按照图形的方式进行表达，需要便于涉众阅读、理解和交流，使之不致因图形过于复杂而难以把握软件架构的概况
- 第三层次：UML模型
  - 使用单一的集成的表示法来对系统的多个方面进行建模，模型范畴包括从系统的静态结构到动态行为特征、从系统的逻辑功能到物理部署

3.5软件架构建模方法发展趋势分析

- 第四层次：形式化模型
  - 两个形式化研究热点的比较

	兴起时间	理论基础	研究方法	缺点
形式化建模	90年代初期	形式化规格说明语言	基于模型的方法	规范难以确认
			代数方法	难以理解，扩展性差
UML模型的形式化	90年代后期	UML模型和形式化规格说明语言	核心语法形式化	难以和各种UML图形匹配
			约束语言方法	难以获得元模型数据
			形式化转换	受到形式语言技术制约

3.5软件架构建模方法发展趋势分析

- 第四层次：形式化模型
  - 形式化规格说明语言
    - 面向模型的方法，其基本思想是利用一些已知特性的数学抽象来为目标软件系统的状态特征和行为特征构造模型。
    - 代数方法，它为目标软件系统的规格说明提供了一些特殊的机制，包括描述抽象概念并进行进程间联接和推理的方法。

3.5软件架构建模方法发展趋势分析

- 第四层次：形式化模型
  - UML语义的形式化
    - 对UML核心语法进行形式化，使得UML成为精确的语言：其目标是运用浅显的数学知识将UML发展为一种精确的（形式化的）建模语言。
    - 约束语言方法，此方法的思想是通过设置一个好的约束来消除语义歧义性。
    - 形式化转换方法，利用形式化语言在不丢失或者少丢失信息的前提下，把对象模型转换为具有一致性管理过程和强有力的工具支持的形式注释。

3.5软件架构建模方法发展趋势分析

- 第五层次：未来模型
  - 第五层次是一种设想，需要前四层技术都发展到相当成熟的阶段；
  - 高层次的方法虽然在方法论上先进，但是在某些实际应用中并不见得比低层次的建模方法优秀；
  - 无论UML为主导的方法和自然语言位主导的方法都不能被证明是更为有效的对软件架构设计决策进行交互的方法，并且对于母语非英语的参与者，这种图形化方式并不能消除在文档中提取信息的困难；

3.5软件架构建模方法发展趋势分析

- 第五层次：未来模型
  - 即使用形式化符号表达，往往辅以非形式化和不完整的图表，来增强对形式化模型的理解。
  - 只有将各个层次的先进技术结合在一起，才能建立完备、精确的模型，才能利用模型直接生成全部的代码。

3.6 小结

- 图形可视化建模方法
- UML建模方法
- 形式化建模方法
- 文本建模方法

- 至今没有一种建模方法能满足软件架构建模的所有需求。