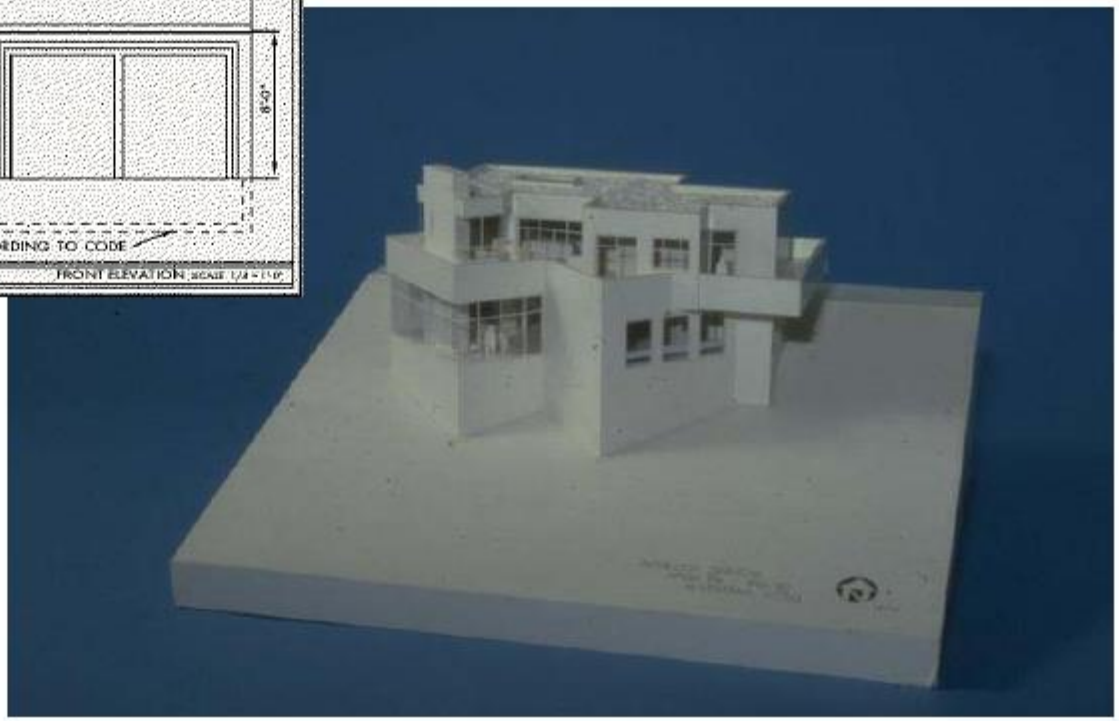
第4章 软件体系结构描述与建模

内容

- 4.1 为什么要进行SA建模？
- 4.2 常用SA描述方法
- 4.3 Kruchten 4+1视图模型
- 4.4 其他常用视图
- 4.5 接口建模（参考《Paul Clements-Documenting Software Architectures Views and Beyond (2nd Edition)》自学）

对房子进行建模

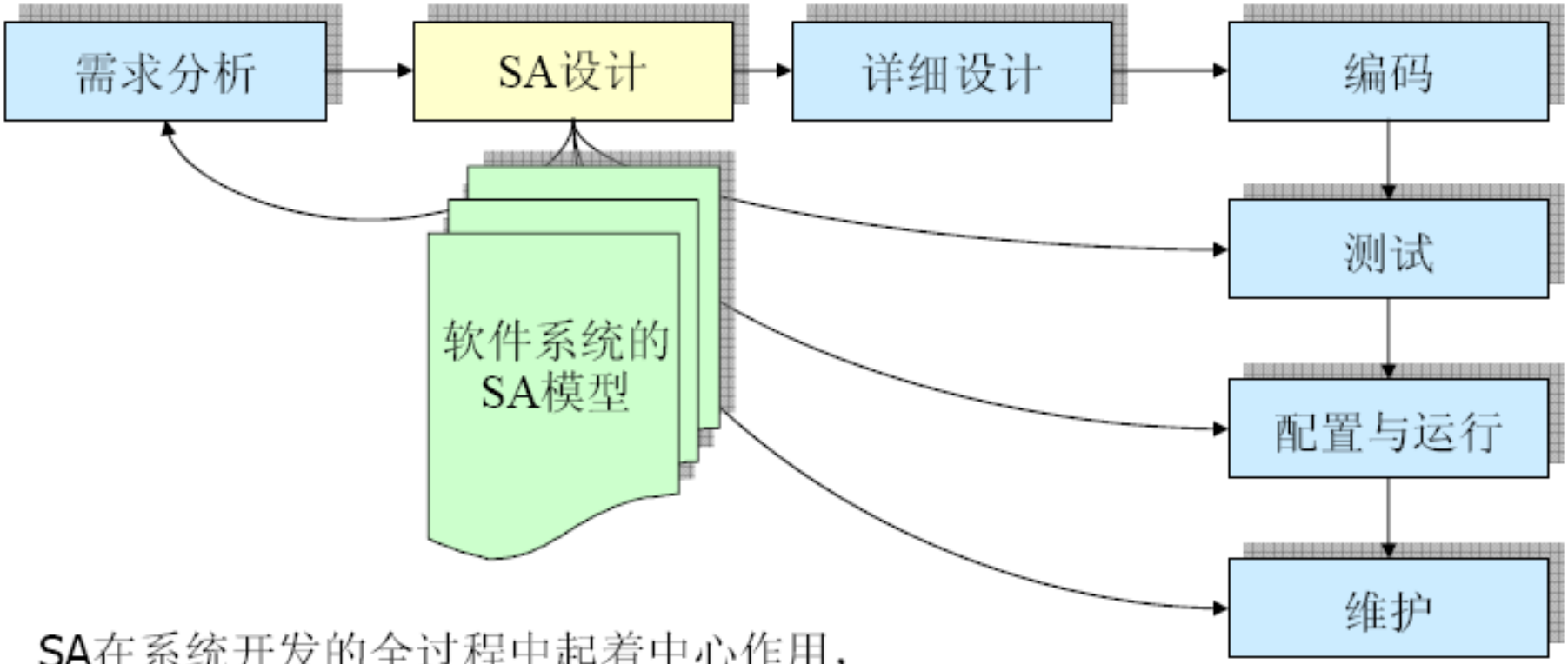

SADDLE
BRICK
FLASHING
ASPHALT SHINGLES
STONE VENEER
CONC. FTG. ACCORDING TO CODE
CONSTRUCTION BLUEPRINT
FRONT ELEVATION



回顾：视图模型

- 视点（View point）
  - ISO/IEC 42010:2007 (IEEE-Std-1471-2000)中规定：视点是一个有关单个视图的规格说明。视图是基于某一视点对整个系统的一种表达。一个视图可由一个或多个架构模型组成
- 架构模型
  - **架构意义上的图及其文字描述**（如软件架构结构图）
- 视图模型
  - 一个视图是关于整个系统某一方面的表达，一个视图模型则是指一组用来构建系统或软件架构的相关视图的集合，这样一组从不同视角表达系统的视图组合在一起构成对系统比较完整的表达

SA生命周期模型



SA在系统开发的全过程中起着中心作用，
是设计/开发的起点和依据，
同时也是配置、运行和维护的指南。

为什么要建立SA模型？

- 针对某一具体的软件系统研发项目，需要以某种可视化/形式化的形式将SA的设计结果加以显式的表达出来，进而支持：
  - 用户、软件架构师、开发者等各方人员之间的**交流**；
  - 分析、**验证SA**设计的优劣；
  - **指导**软件开发组进行系统**研发**；
  - 为日后的软件**维护**提供基本文档。

- 一个软件系统的基础组织可以通过以下几种元素表达：
  - 构成或组成系统的**结构元素**及其**接口**；
  - **行为元素**表现了结构元素之间的交互和协作；
  - 结构元素和行为元素的组合，从而形成更大的子系统。

内容

- 4.1 为什么要进行SA建模？
- 4.2 常用SA描述方法
- 4.3 Kruchten 4+1视图模型
- 4.4 其他常用视图
- 4.5 接口建模（参考《Paul Clements-Documenting Software Architectures Views and Beyond (2nd Edition)》自学）

线框描述法

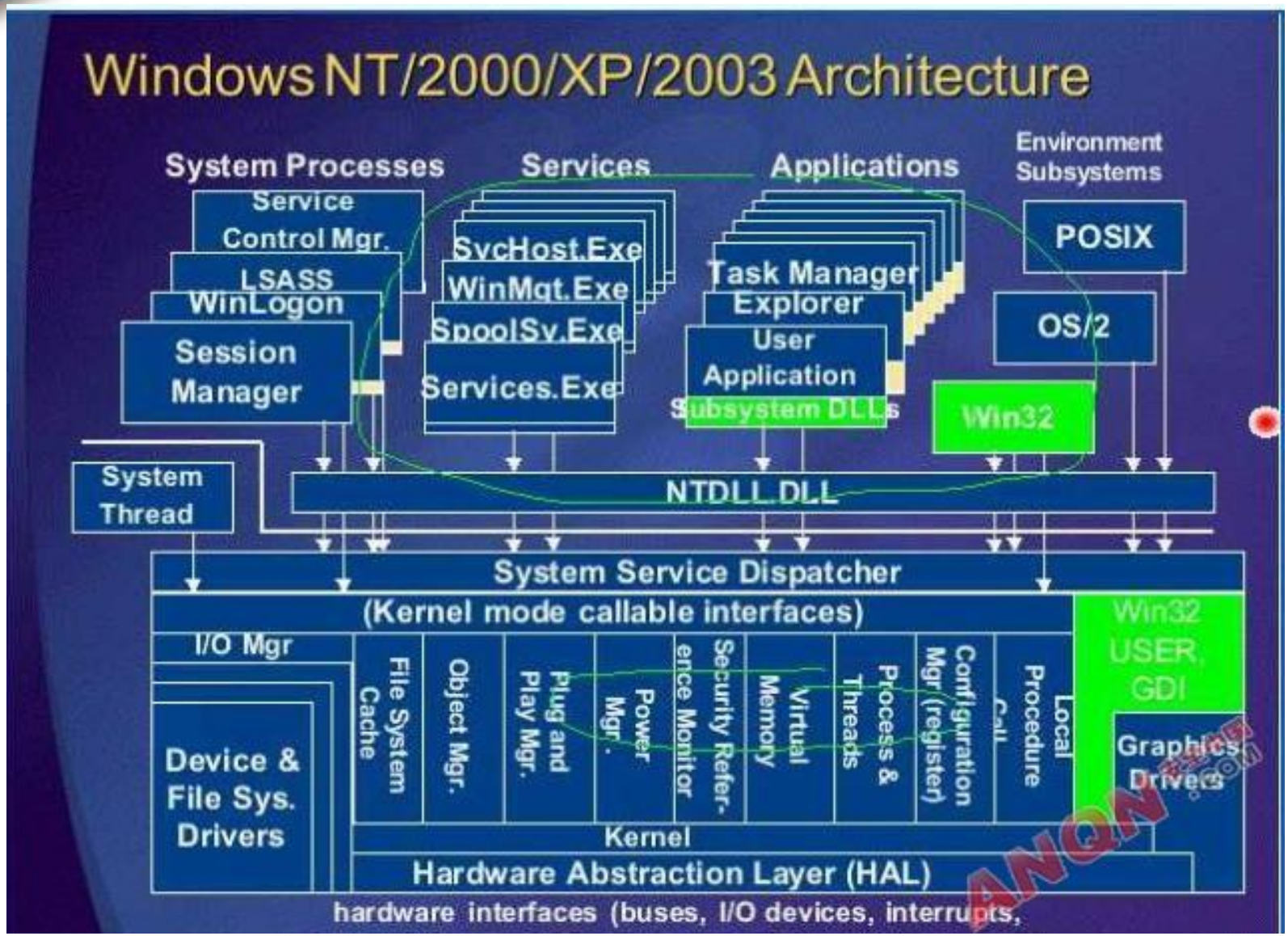
- 采用由矩形框和有向线段组合而成的图形表达工具。在这种方法中**矩形框代表抽象构件**，框内注明的文字为抽象构件的名称，**有向线段代表辅助各构件**进行通信、控制或关联的**连接件**。
- 优点：
  - 灵活
  - 能够直观反应系统架构，同时也易于理解

常用工具：Word/Powerpoint/Visio/Smart Draw

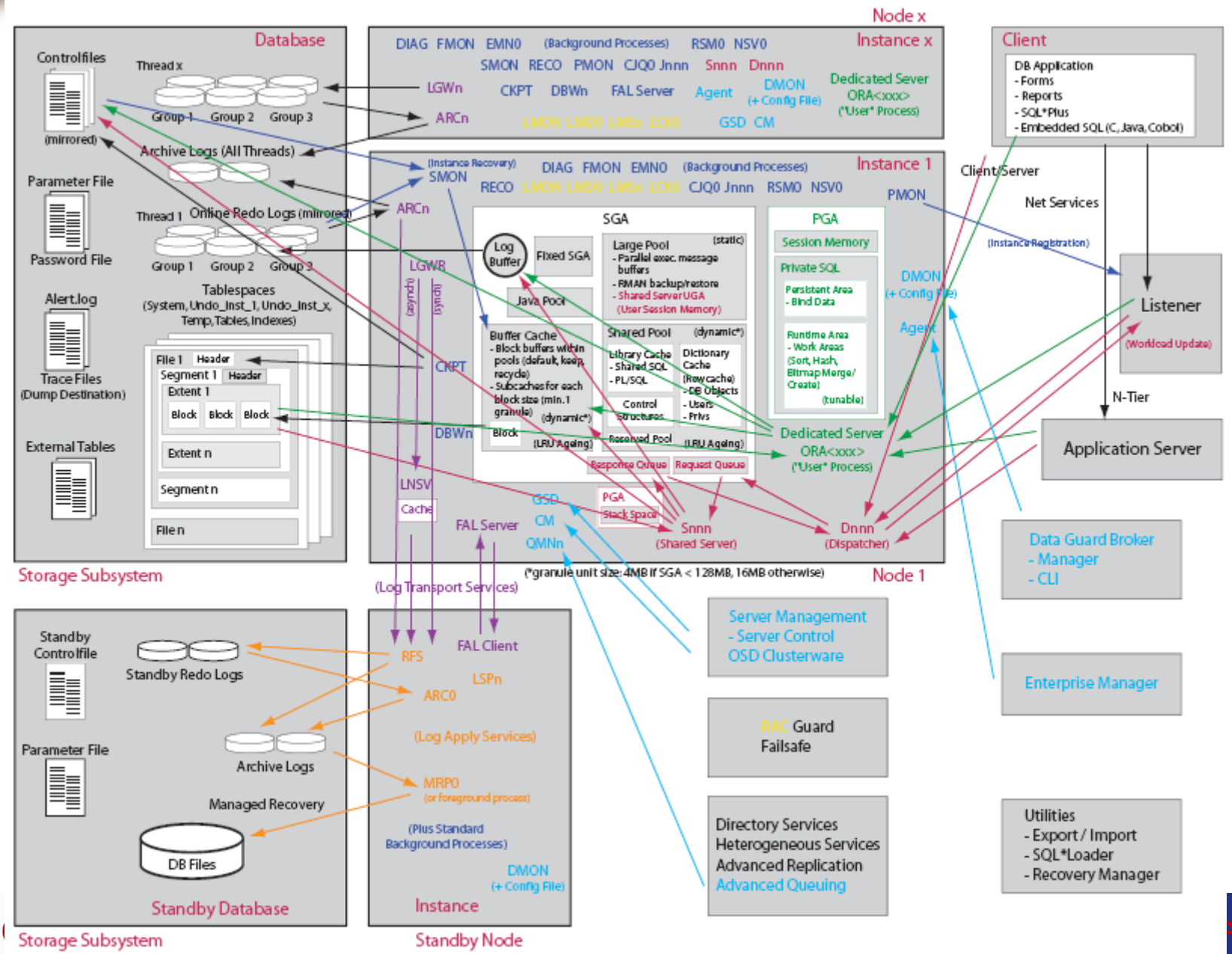
线框描述法（续）

- 缺点：
  - **二义性**：图形的本质所决定的模糊性，不同人有不同的理解；
  - **矛盾性**：模型中可能存在相互冲突的陈述；
  - **不完备**：无法描述所有的细节；
  - **异构性**：各个建模规范不同，模型也不同，难以支持模型在各个建模工具之间的交换；
  - **无法自动化**：只能由人理解，靠软件工具来立即比较困难，因此无法实现自动化的验证与推理。

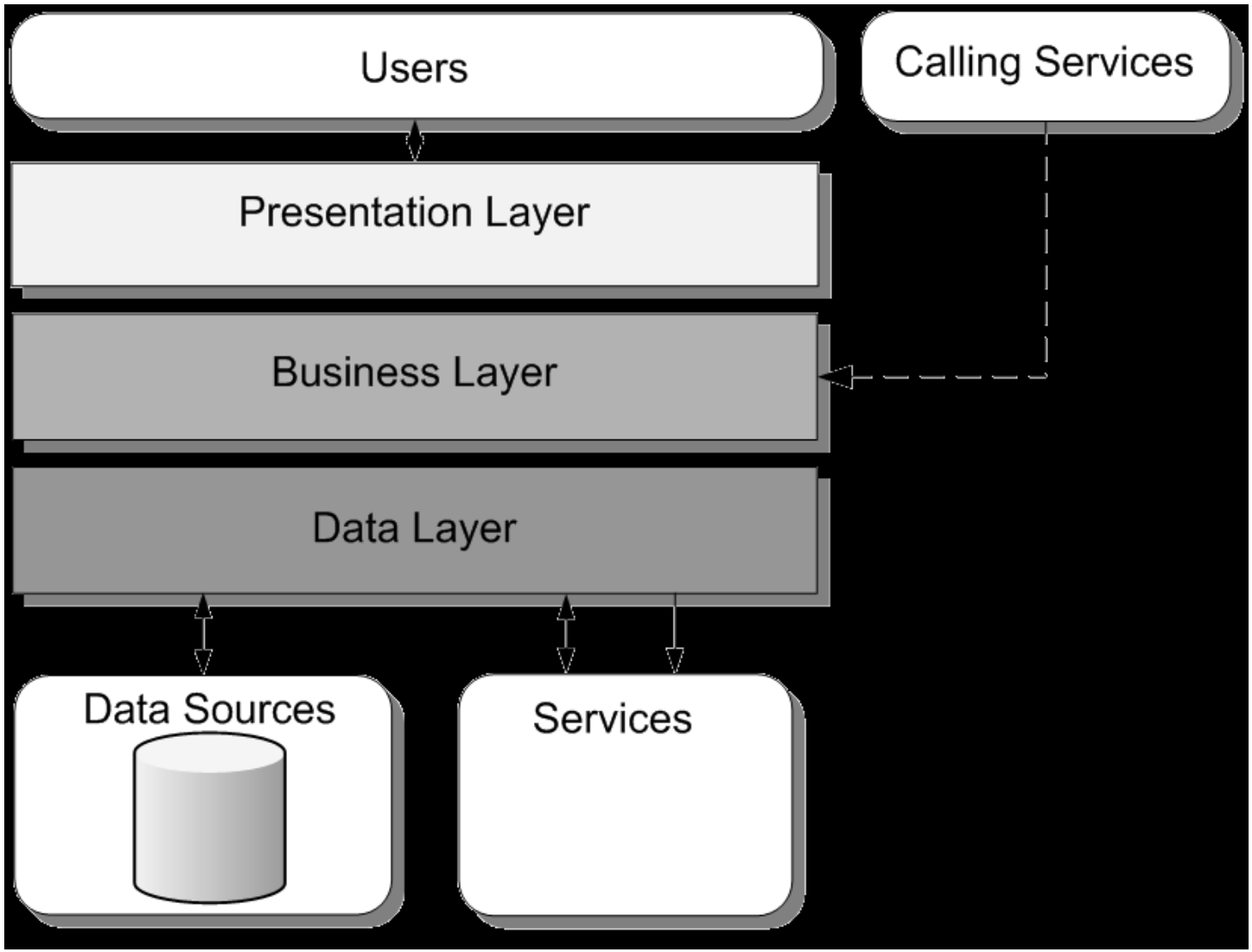
WindowsNT 2000～2003体系结构图



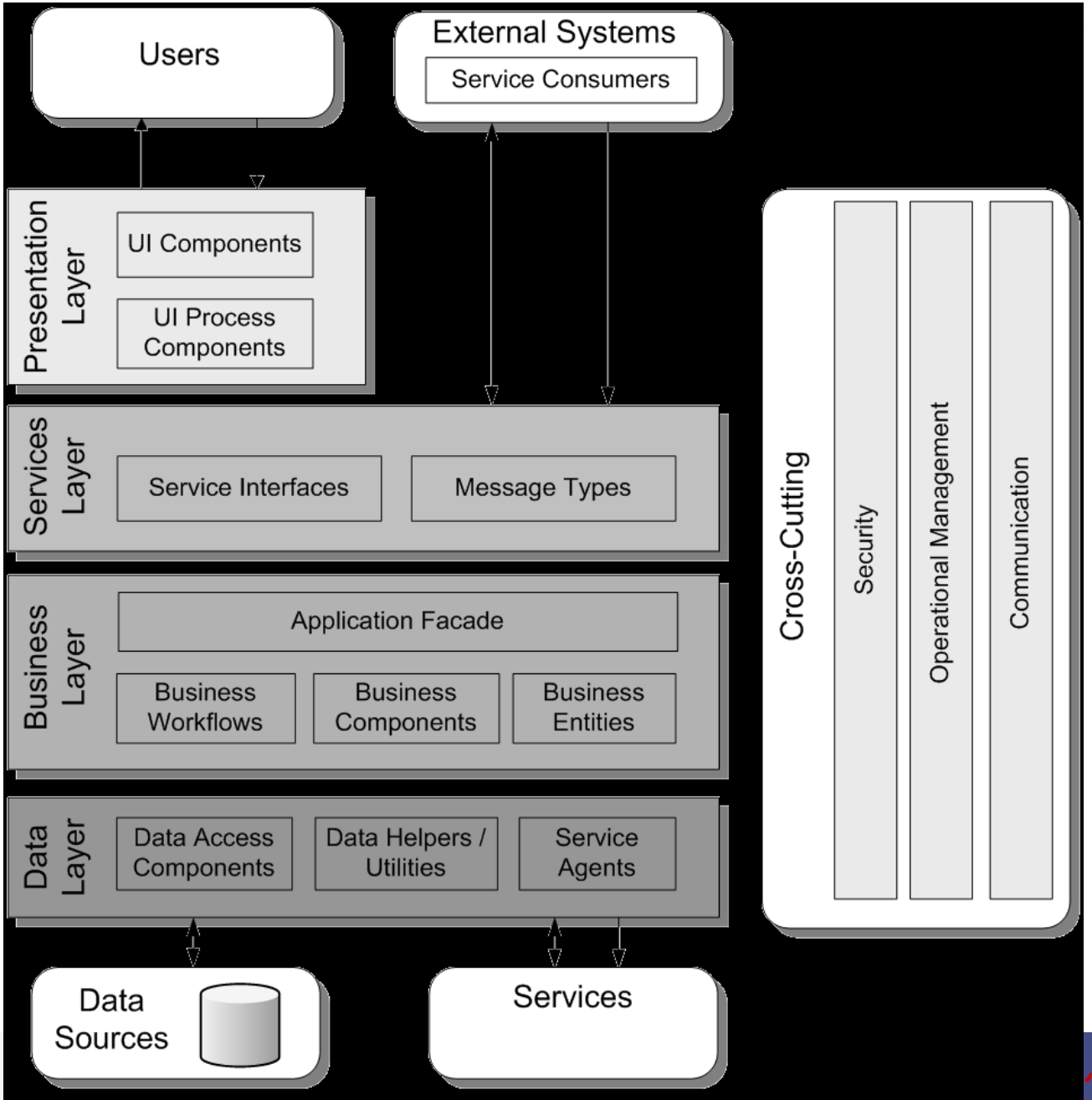
Oracle服务器软件体系结构图



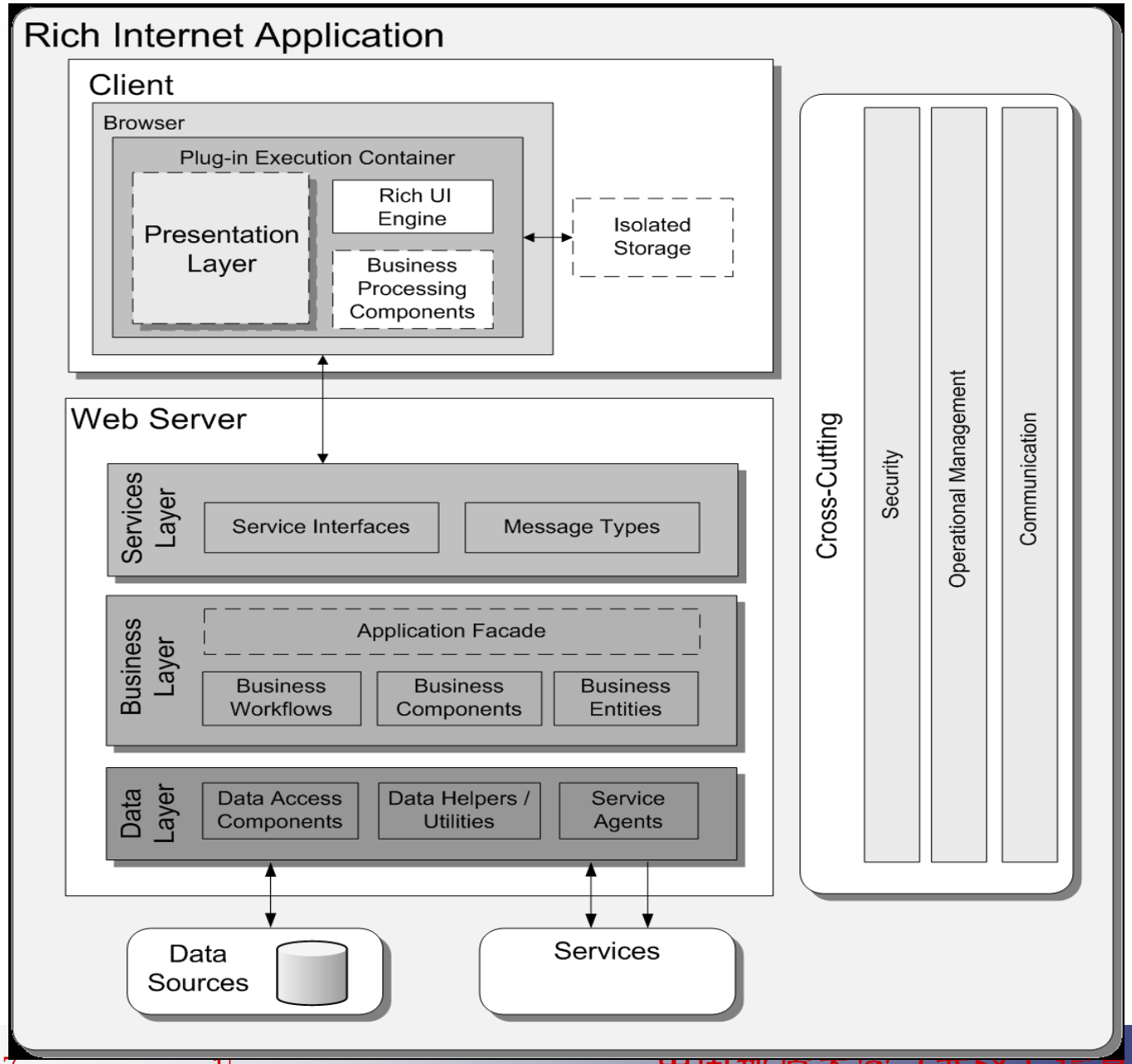
The logical architecture view of a layered system



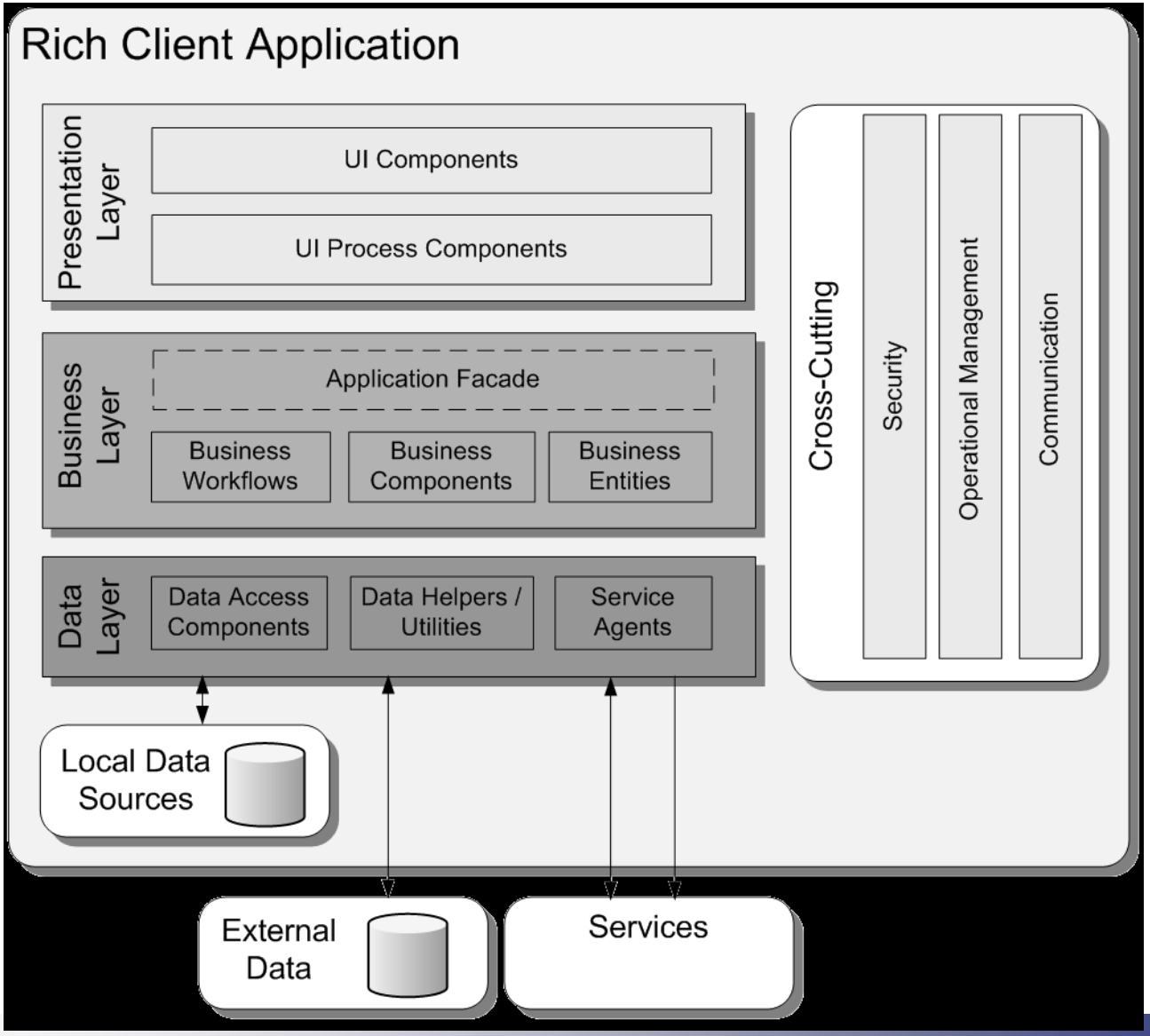
Common application architecture



Architecture of a typical RIA implementation



Overall view of typical Rich(Fat) Client Architecture



形式化描述方法

- 软件体系结构描述语言
- Architectural Description Language (**ADL**)
- 在底层统一的语义模型的支持下，精确、无歧义的描述SA，为SA的表示、分析、演化、细化、设计过程等 提供支持；
- **使用数学的符号标记把系统分解为构件和连接件**，并说明这些元素如何连接在一起构成一个系统

典型 ADL

- ACME (CMU/USC)
  - http://www.cs.cmu.edu/~acme
- Rapide (Stanford)
  - http://pavg.stanford.edu/rapide/
- Wright (CMU)
  - http://www.cs.cmu.edu/afs/cs/project/able/www/wright/index.html
- Unicon (CMU)
- Aesop (CMU)
- MetaH (Honeywell)
- C2 SADL (UCI)
- SADL (SRI)
- etc

形式化的软件体系结构描述

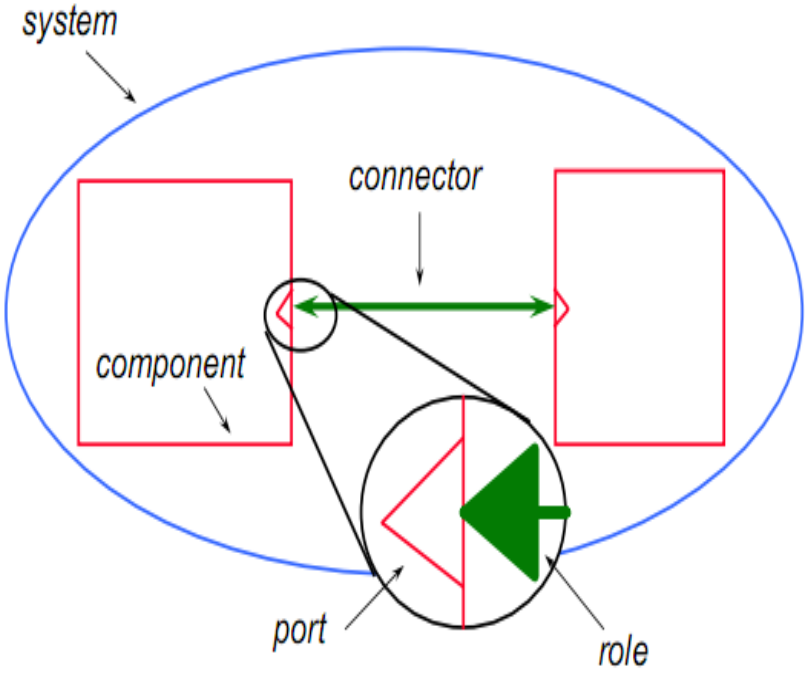
ADL示例：ACME

ACME

- ACME是由CMU的David Garlan等人创建的一种ADL；
- 相比于其他ADL，ACME非常简单、容易理解；
- ACME支持从四个方面对SA进行描述：
  - 结构
  - 属性
  - 设计约束
  - 类型与风格
- ACME可描述某个具体系统的SA，也可描述某一种抽象的SA风格(管道、C/S、…)

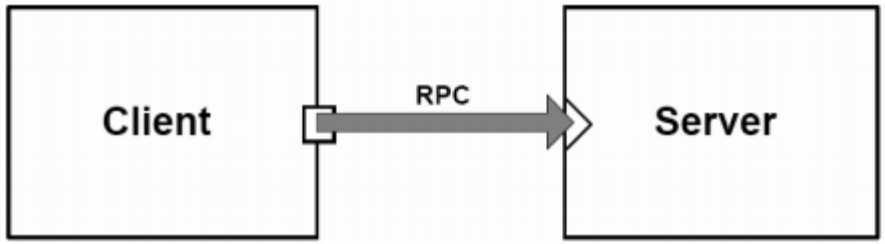
结 构

- ACME定义了八种实体，分别是：
  - 系统(system)
  - 构件(component)
  - 接口(port or interface)
  - 连接件(connector)
  - 角色(role)
  - 构件与连接件之间的关系(attachments)



  - 表述(representations)
  - 表述映射(map)

C/S系统的ACME



定义系统

```
System simple_cs = {
        Component client = { Port send-request; };
        Component server = { Port receive-request; };
        Connector rpc = { Roles { caller, callee}};
        Attachments {
                client.send-request to rpc.caller;
                server.receive-request to rpc.callee;
        }
}
```

定义两个构件：client/server

定义连接件

定义连接件与构件之间的联系

C/S系统的ACME

```
System simple_CS = {
    Component client = {…}
    Component server = {
        Port receiveRequest;
        Representation serverDetails = {
            System serverDetailsSys = {
                Component connectionManager ={
                    Ports {externalSocket; securityCheckIntf; dbQueryIntf}}
                Component securitymanager = {
                    Ports {securityAuthorization; credentialQuery}}
                Component database = {
                    Ports {securityManagementIntf; queryIntf; }}
                Connector SQLQuery = {Role {caller; callee }}
                Connector clearanceRequest = {Roles {requestor; grantor}}
                Connector securityQuery = {Roles {securityManager; requestor}}
                Attachments {
                    connectionManager.securityCheckIntf to clearanceRequest.requestor;
                    securityManager.securityAuthorization to clearanceRequest.grantor;
                    connectionManager.dbQueryIntf to SQLQuery.caller;
                    database.queryIntf to SQLQuery.callee;
                    securityManager.credentialQuery to securityQuery.securityManager;
                    database.securityManagementIntf to securityQuery.requestor;
                    }
                Bindings { connectionManager.externalSocket to server.receiveRequest }
            }}
    Connector rpc = {…}
    Attachments {client.sendRequest to rpc.caller; server.receiveRequest to rpc.callee}
}
```

定义两个构件：client/server

为构件server定义了一个表述，用来描述其内部细节

表述中的构件

表述中的连接件

带有表述的C/S结构的ACME

表述中构件与连接件之间的联系

定义连接件

定义连接件与构件之间的联系

示例：ACME属性

```
System simple-cs = {
…
    Component server = {
            port rpc-request = {
            Property sync-requests : boolean = true;同步性=true
            };
            Property max-transactions-per-sec : int = 5;每秒最大事务处理数目=5
            Property max-clients-supported : int = 100;最多可支持的客户端数目=100
    };
    Connector rpc = { …
            Property protocol : string = “aix-rpc”;连接协议=aix-rpc
    };
…
};
```

设计约束

- 设计约束(Design Constraints)是SA描述的关键成分，描述了构件/连接件需要遵循何种约束条件。
- ACME使用一阶谓词逻辑(First Order Predicate Logic)约束语言来描述设计约束，约束被当作谓词。
- 约束可以与ACME中任何设计元素相关联。
- 例如：

(1) connected(client, server)
(2) forall conn : connector in systemInstance.Connectors @ size(conn.roles)=2
(3) self.throughputRate >= 3095
(4) comp.totalLatency = (comp.readLatency + comp.processingLatency + comp.writeLatency)

设计约束

• 约束可以通过两种方式附加到设计元素，分别是invariant和heuristic。

• Invariant：不可违反的规则

• heuristic：应该遵守的规则

```
System messagePathSystem = {
…
        Connector MessagePath = {
        Roles {sources; sink;}
        Property expectedThroughput : float = 512;
        Invariant ( queueBufferSize >= 512) and
        (queueBufferSize<= 4096);
        Heuristic expectedThroughput <= (queueBufferSize / 2);
        }
}
```

设计约束

```
Component Type naïve-client = {

    Port Request = {
        Property protocol = rpc-client };

    Property request-rate : integer
        << default = 0; units = “rate-per-sec” >>;

Invariant forall p in self.Ports |
        (p.protocol = rpc-client);
Invariant size(Ports) <= 5;
Invariant request-rate >= 0;
Heuristic request-rate <= 100;
}
```

设计约束

System simpleCS = { …

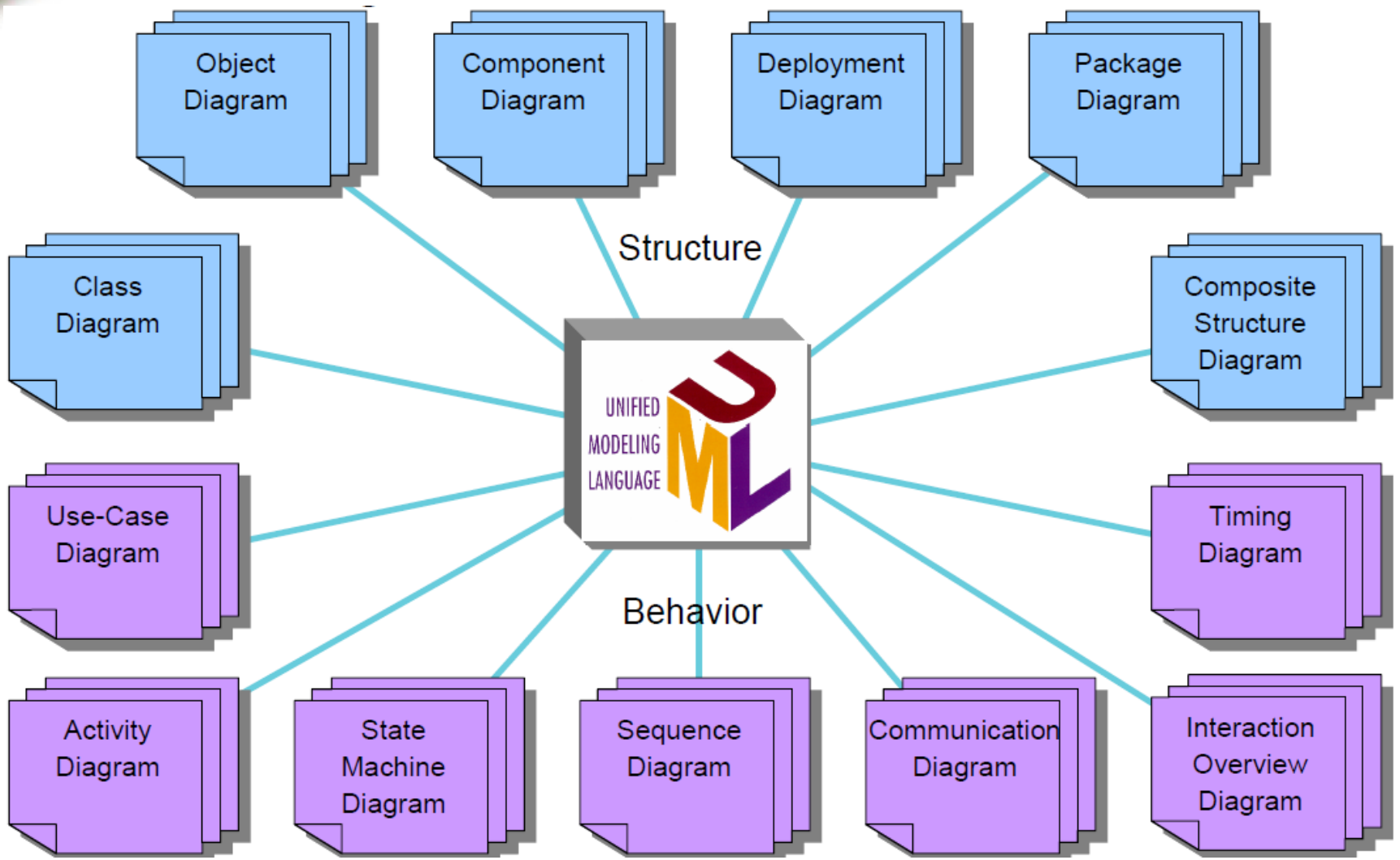
```
// simple rule requiring a primary server
Invariant exists c : server in self.components |
.isPrimaryServer == true;

// simple performance heuristic
Heuristic forall s : server in self.components |
s.transactionRate >= 100;

// do not allow client-client connections
Function no-peer-connections(sys : System) =
forall c1, c2 in sys.components |
connected(c1, c2) ->
!(declaresType(c1,clientT)
and declaresType(c2, clientT));
```

…
};

ADL描述SA的优缺点

- ADL的**优点**：
  - 表达架构的一个正式方式
  - 可做到人机可读
  - 在一个比以前更高的水平上描述系统
  - 允许在完整性、一致性、歧义性和性能等方面分析和评估架构
  - 支持自动生成软件系统
- ADL的**缺点**：
  - 使用类计算机高级语言的形式描述，表达**不够直观**，难以理解
  - 对于ADLs应该表达什么，**没有一个普遍共识**，特别是关于架构的行为
  - 目前使用表达解析相对比较困难，**没有很好的商业工具提供支持**

UML描述方法


Object Diagram
Component Diagram
Deployment Diagram
Package Diagram
Structure
Class Diagram
Composite Structure Diagram
UNIFIED MODELING LANGUAGE
Use-Case Diagram
Timing Diagram
Behavior
Activity Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram

内容

- 4.1 为什么要进行SA建模？
- 4.2 常用SA描述方法
- 4.3 Kruchten 4+1视图模型
- 4.4 其他常用视图
- 4.5 接口建模（参考《Paul Clements-Documenting Software Architectures Views and Beyond (2nd Edition)》自学）

回顾：视图模型

- 视点（View point）
  - ISO/IEC 42010:2007 (IEEE-Std-1471-2000)中规定：视点是一个有关单个视图的规格说明。视图是基于某一视点对整个系统的一种表达。一个视图可由一个或多个架构模型组成
- 架构模型
  - 架构意义上的图及其文字描述（如软件架构结构图）
- 视图模型
  - 一个视图是关于整个系统**某一方面的表达**，一个视图模型则是指**一组用来构建系统或软件架构的相关视图的集合**，这样一组从不同视角表达系统的**视图组合在一起构成对系统比较完整的表达**

Multi-View Architecture

- Systems are composed of many structures（系统由许多结构组成）
  - Code units, their decomposition and dependencies
- Processes and how they interact
- How software is deployed on hardware
- Many others

A View Is a Representation of a Structure, That is, a Representation of a Set of System Elements and the Relations Associated With Them

使用“多视图模型”进行SA建模的原因（1）

SA核心模型只是刻画了SA中应具备的**基本要素**(构件、连接件、配置等);
而一个复杂的软件系统还包括**结构、行为**等其他方面的特性，如何在SA模型中刻画它们?

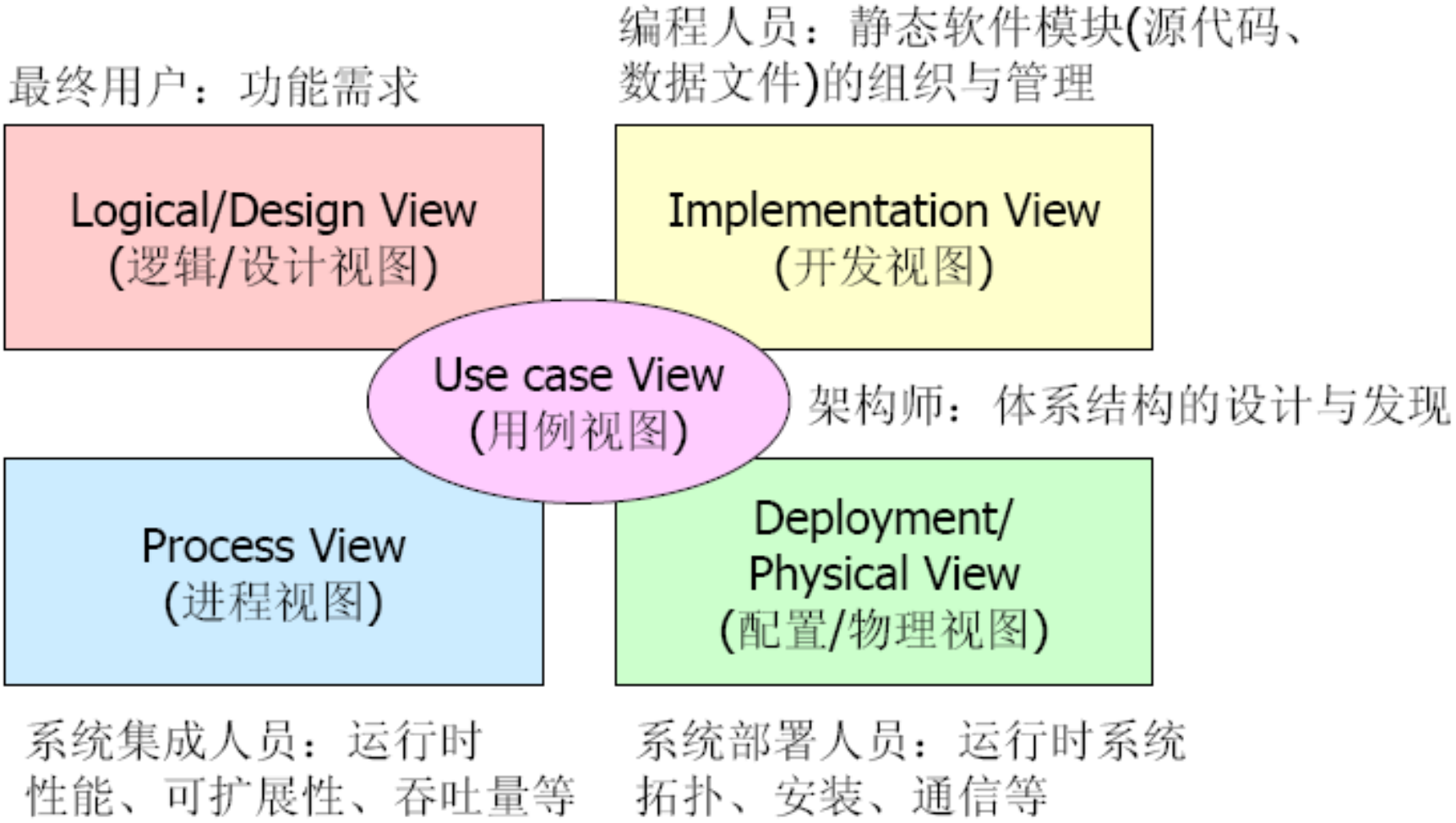
A software architecture is a complex entity that cannot be described in a simple onedimensional fashion (**软件体系结构非常复杂，无法用简单的一维模式加以描述**).

使用“多视图模型”进行SA建模的原因(2)

- 考虑建筑体系结构的描述：
  - the room layouts (房间布局图)
  - the elevation drawings (正面图)
  - the electrical diagrams (电路图)
  - the plumbing diagrams (管道图)
  - The security system plans (安全系统计划)
  - …
- 多视图SA模型：**从多个不同角度建立SA的模型，分别刻画SA某一方面的性质**。
  - 系统的每一个不同的视图反映了一组系统相关人员所关注的系统的特定方面；
  - 多视图体现了“**关注点分离**”(separation of concerns)的思想，使系统更易于理解，方便系统相关人员之间进行交流，并且有利于系统的一致性检测以及系统质量属性的评估。

Kruchten's 4+1视图

1995年，由Kruchten在其博士论文中提出



RSA支持的十种图

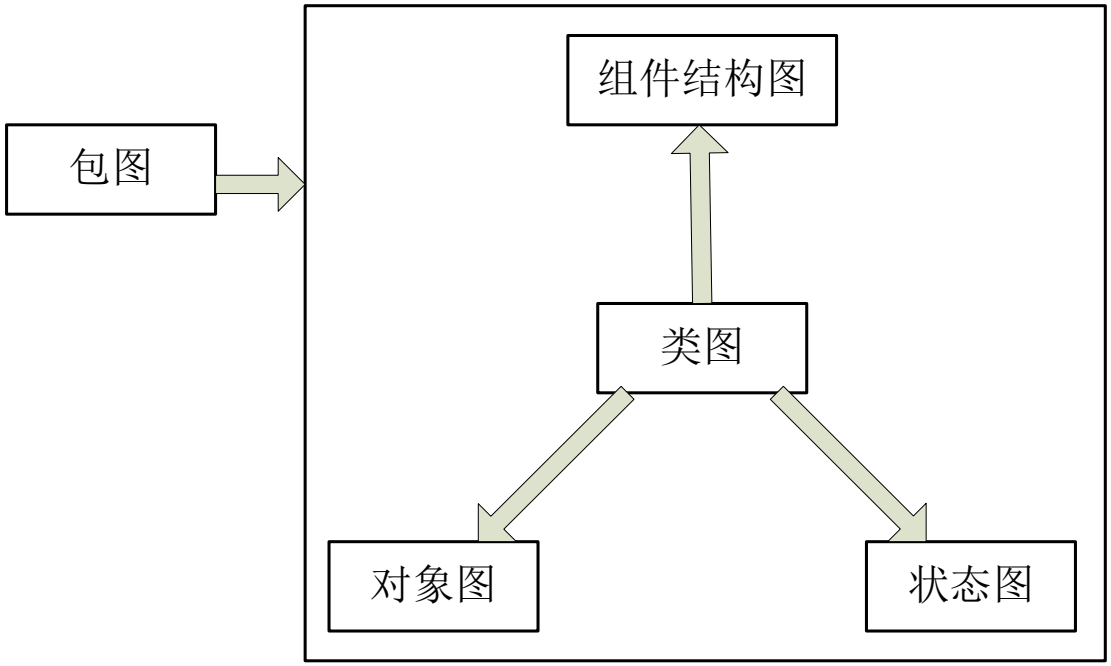
名称	用途
类图(Class Diagram)	类图是使用得最为广泛的UML图之一。它使用类和接口来描述组成系统的实体以及它们之间的静态关系。利用类图可以生成源代码作为搭建系统的框架。
组件图(Component Diagram)	组件图描述了系统实现的组成和相互依赖。它能够将小的事物（例如类）组装成更大的、可以部署的部件。组件图的详细程度取决于你想展现什么。
组合结构图(Composite Structure Diagram)	组合结构图是UML2.0中新出现的图。随着系统变得越来越复杂，事物之间的关系也变得复杂了。从概念上讲，组合结构图将类图和组件图连接了起来。它并不强调类的详细设计和系统如何实现。它描述了系统中的事物如何联合起来实现某一个复杂的模式。
部署图(Deployment Diagram)	部署图描述了你的系统是如何实际运行的，同时还描述了系统是如何应用到硬件上的。一般情况下，使用部署图说明组件是如何在运行时进行配置的。
对象图(Object Diagram)	对象图使用了和类图一样的语法，同时还展示了在一个特定的时间类的实例。
活动图(Activity Diagram)	活动图记录了从一个行为或活动到下一个的转化。
通信图(Communication Diagram)	通信图是一种交互图，它关注的是一个行为中涉及到的事物以及它们之间反复传递的消息。
序列图(Sequence Diagram)	序列图是一种交互图。它关注的是在执行的时候，在事物之间传递的消息的类型和顺序。
状态机图(State Machine Diagram)	状态机图描述的是事物内部状态的转化。这个事物可能是一个单独的类，也可以是整个系统。
用例图(Use Case Diagram)	用例图描述了系统的功能性需求。

（1）逻辑视图

- **逻辑视图（logical view）**主要支持系统的**功能需求**，即系统提供给最终用户的服务。常通过结构元素、核心抽象、关注点分离和职责的划分来实现系统的逻辑功能。
- 系统被分解成一系列的**功能抽象（如层、类、接口）**，这些抽象主要来自用户需求所在的问题领域，同时也表达了软件领域的相应概念。
- 如以前所学应用程序三层逻辑架构：表现层、业务逻辑层、数据层

使用UML2.0构建逻辑视图

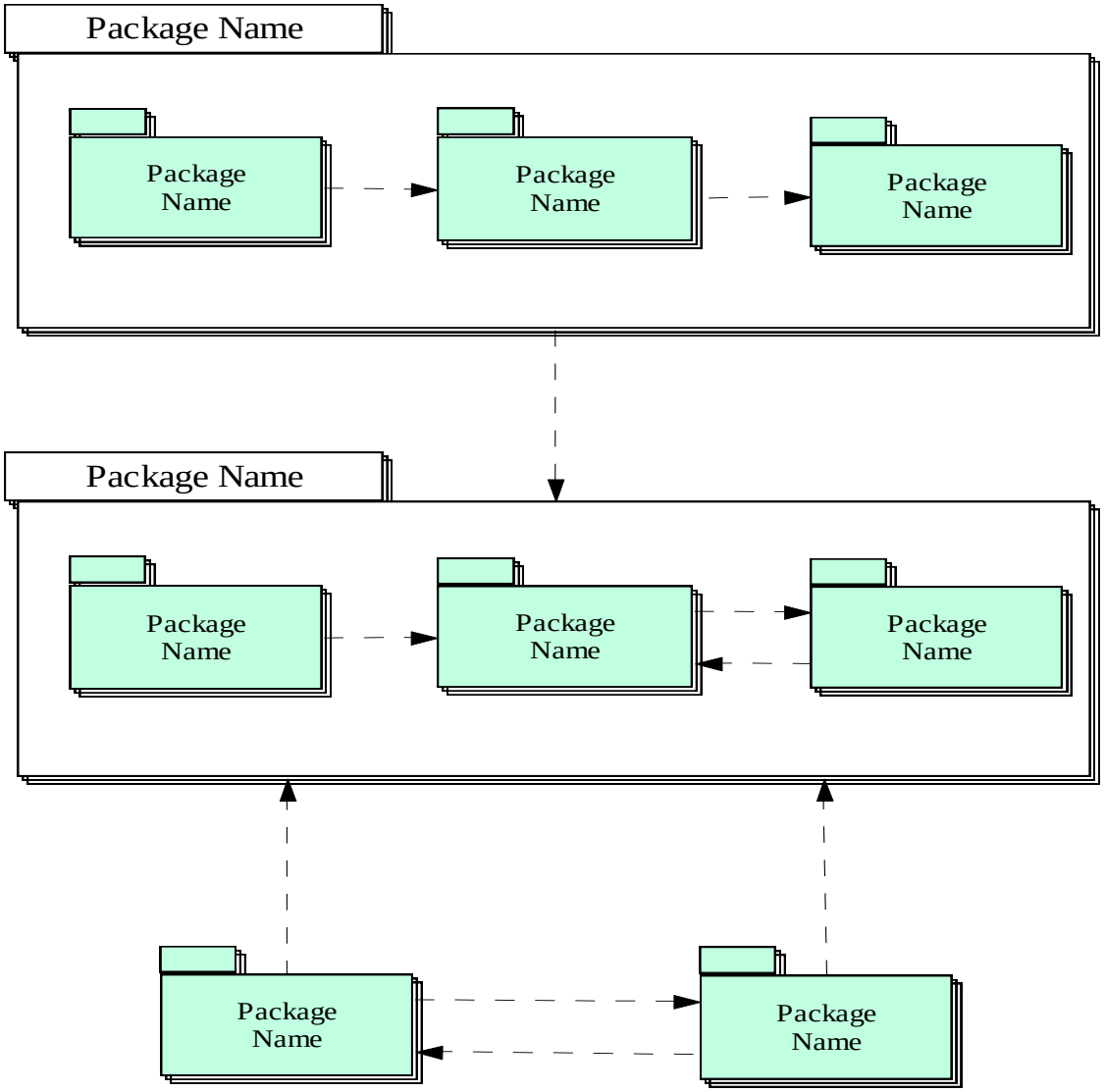
使用UML2构建逻辑视图



1、从类图建模系统
2、使用包图进行逻辑分组

可选的使用
3、对象图，类之间的关系需要通过实例解释
4、状态图，当需要解释特定类内部的一个状态
5、组合结构图，当部分类和类之间的关系需要建模

逻辑视图的UML表示——包图



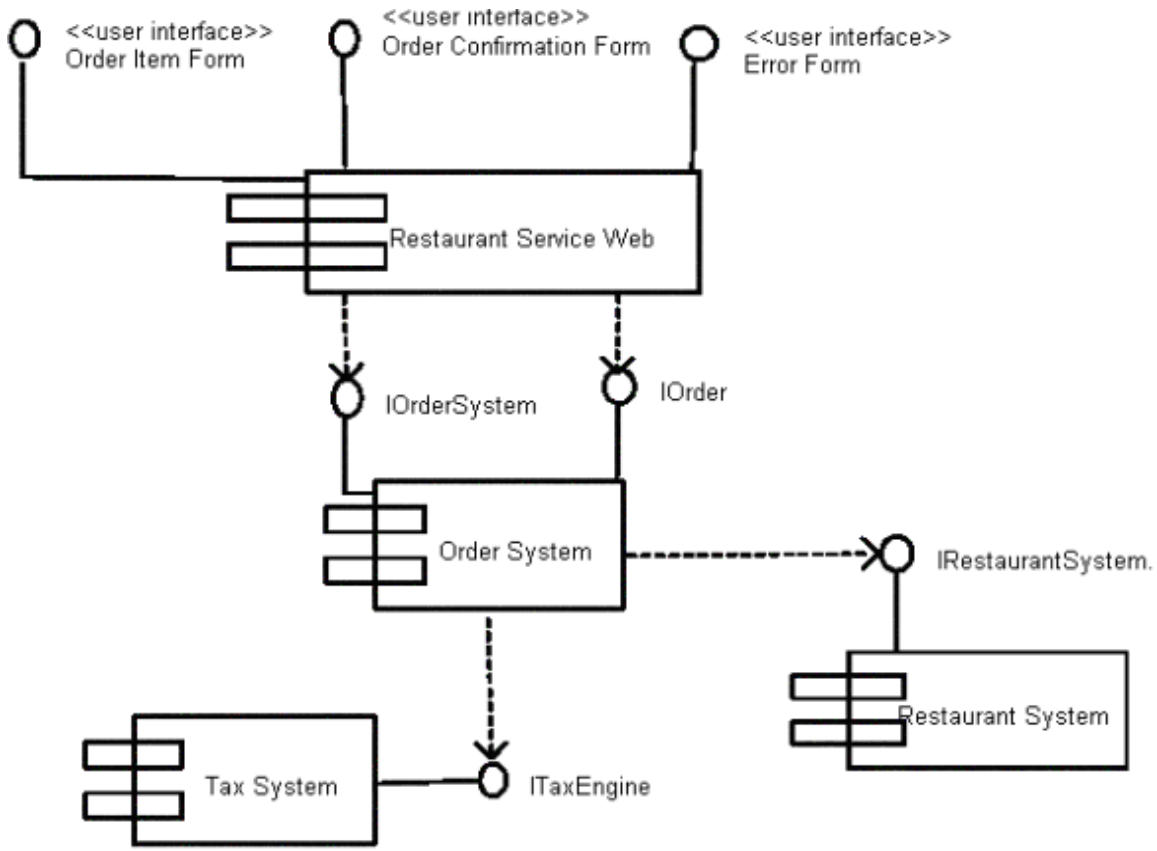
UML Package Diagram

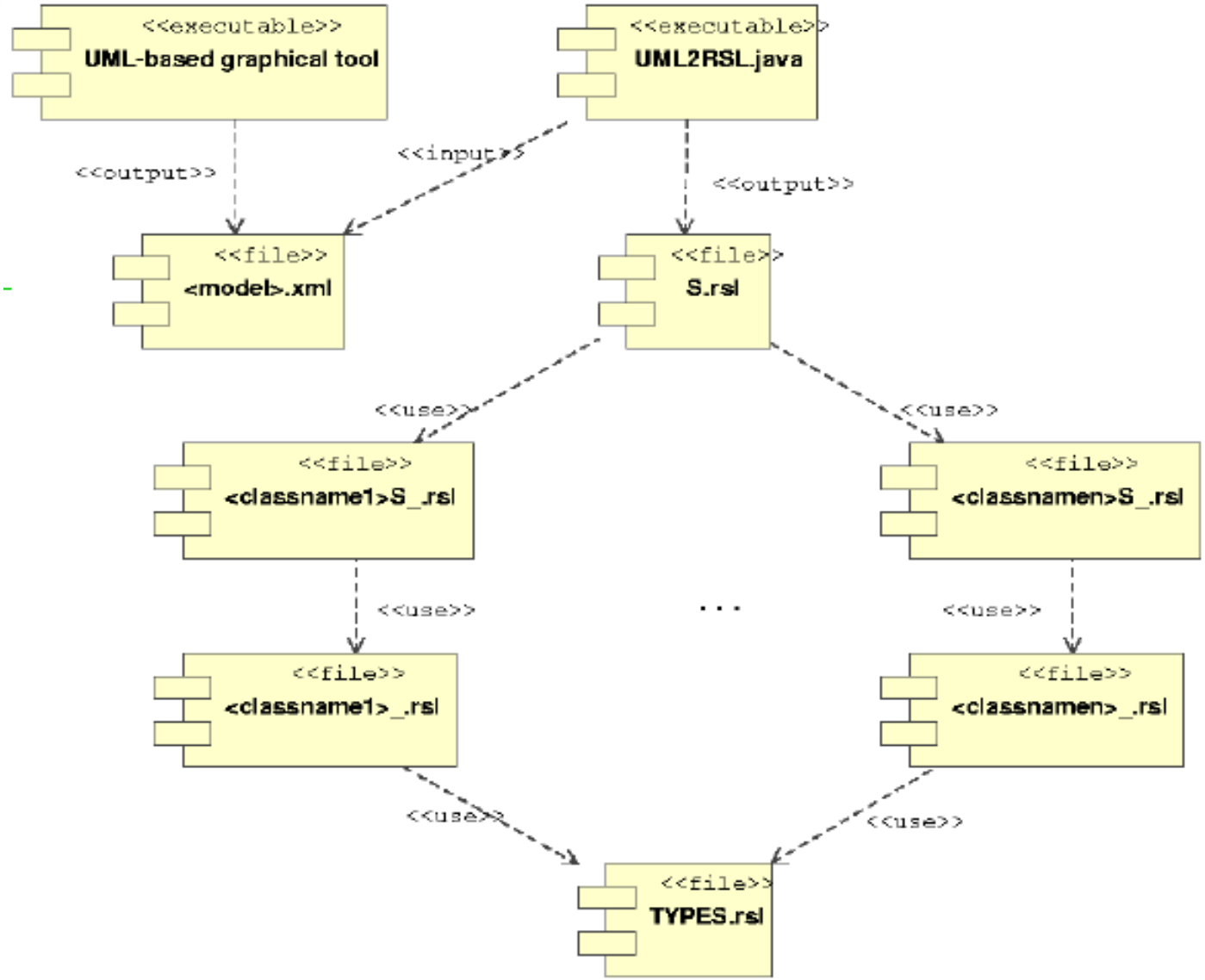
（2）开发视图

- **开发视图**也称**实现视图（implementation view）**、**模块视图(module view)**，主要侧重于软件模块的组织和管理。例如：一个构件的实现依赖于哪些其他构件、哪些源文件实现了哪些类，等等。
  - 系统-子系统-模块(构件、文件、资源等)，并组织成层次结构
  - 开发视图的另一个焦点在于：描述各模块之间的依存关系。
- 开发视图要考虑**软件内部的实现需求**，如软件开发的容易性、软件的复用和软件的通用性，要充分考虑由于具体开发工具的不同而带来的局限性。

开发视图的UML表示

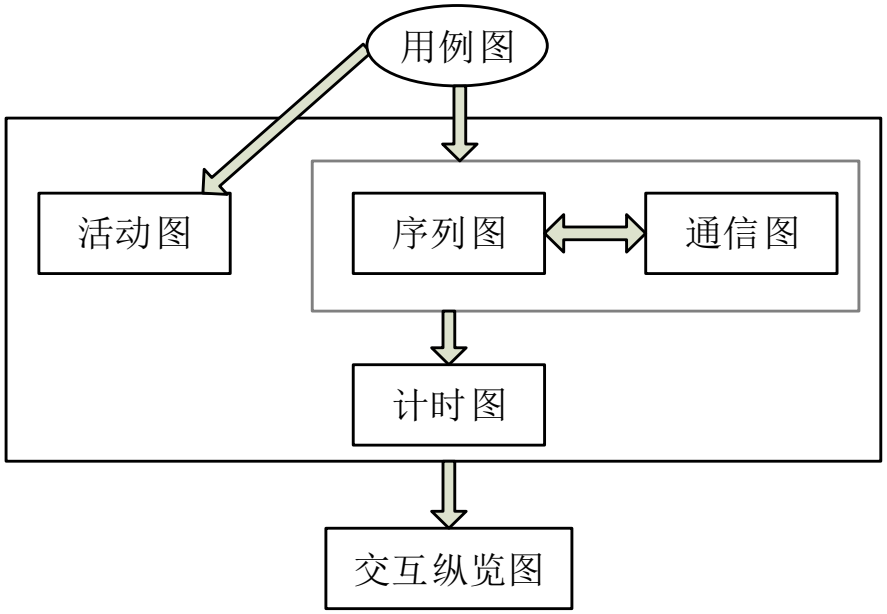
- 在UML中，开发视图主要使用构件图、包图进行描述，并辅之以交互图、状态图、组合结构图




<<executable>>
UML-based graphical tool
<<executable>>
UML2RSL.java
<<output>>
<<input>>
<<output>>
<<file>>
<model>.xml
<<file>>
S.rsl
<<use>>
<<use>>
<<file>>
<classname1>S_.rsl
<<file>>
<classnamen>S_.rsl
<<use>>
<<use>>
<<file>>
<classname1>_.rsl
<<file>>
<classnamen>_.rsl
<<use>>
<<use>>
<<file>>
TYPES.rsl

（3）进程视图

- **进程视图（process view）**侧重于**系统的运行特性**，主要关注一些非功能性的需求，例如系统的性能和可用性。
- 并发性、分布性、系统集成性和容错能力；
- 逻辑视图中的各个抽象概念如何在进程中被执行；
- 逻辑视图中的各个类的操作具体是在进程中的哪一个线程中被执行的。



1、使用序列图或通信图去建模用例实现中的简单交互

可选的使用

2、添加活动图来实现场景，其中的业务逻辑是一系列动作，包括分支和并行处理

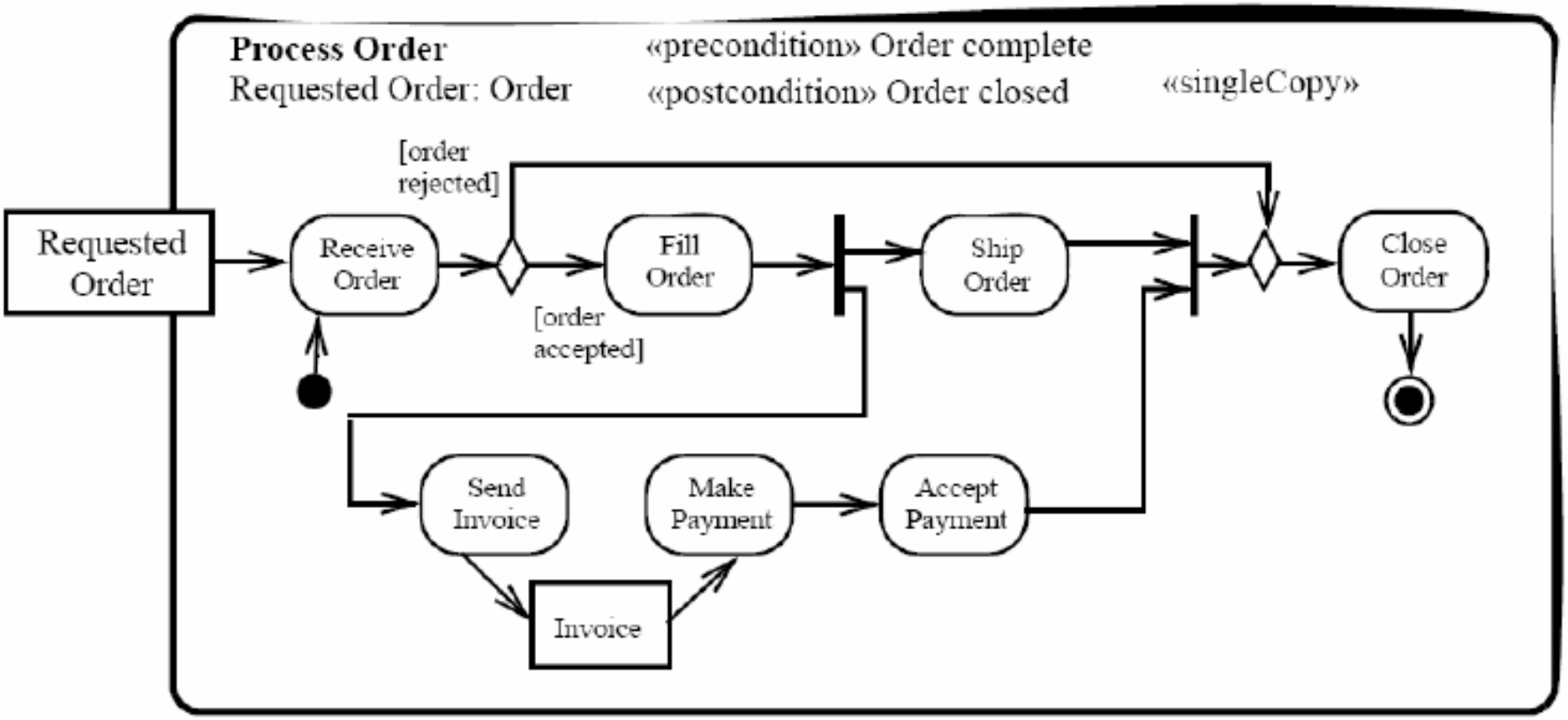
3、添加计时图来对性能建模

4、对于复杂场景，可以与其他场景组合，使用交互概览图

进程视图的UML表示

- 进程视图通常使用以下UML图来描述一个系统的运行时行为：
  - Activity diagrams (活动图)
  - Interaction diagrams (状态图)
  - Sequence diagrams (序列图)
  - Communication/Corporation diagrams (通讯/协作图)
  - Interaction overview diagrams (交互纵览图)
  - Timing diagrams (计时图)

进程视图的UML表示



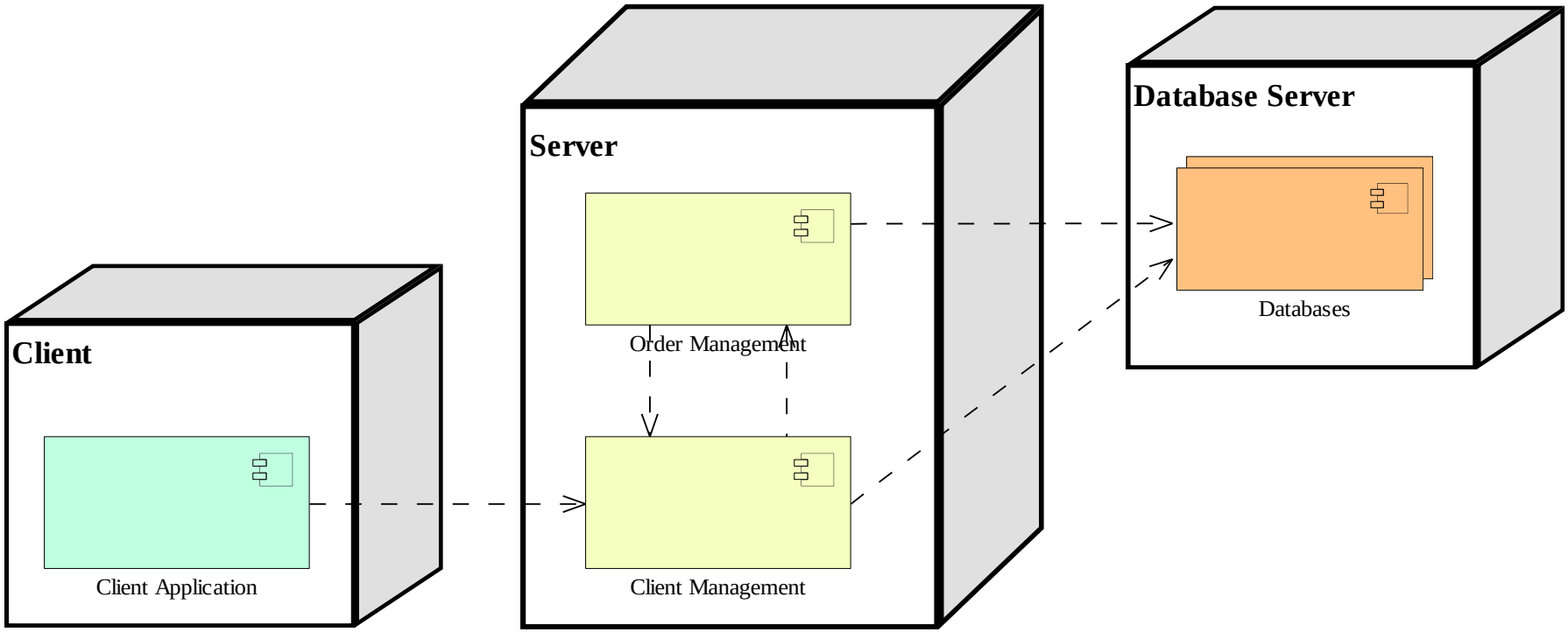
UML Activity Diagram

（4）物理视图

- **物理视图（physical view）**也称**部署视图（Deployment view）**，主要考虑如何**把软件映射到硬件上**；
- 通常要考虑系统性能、规模、可靠性等；
- 解决软件系统配置、安装、系统部署后的物理拓扑结构、系统执行过程中的通讯问题等。

物理视图的UML表示

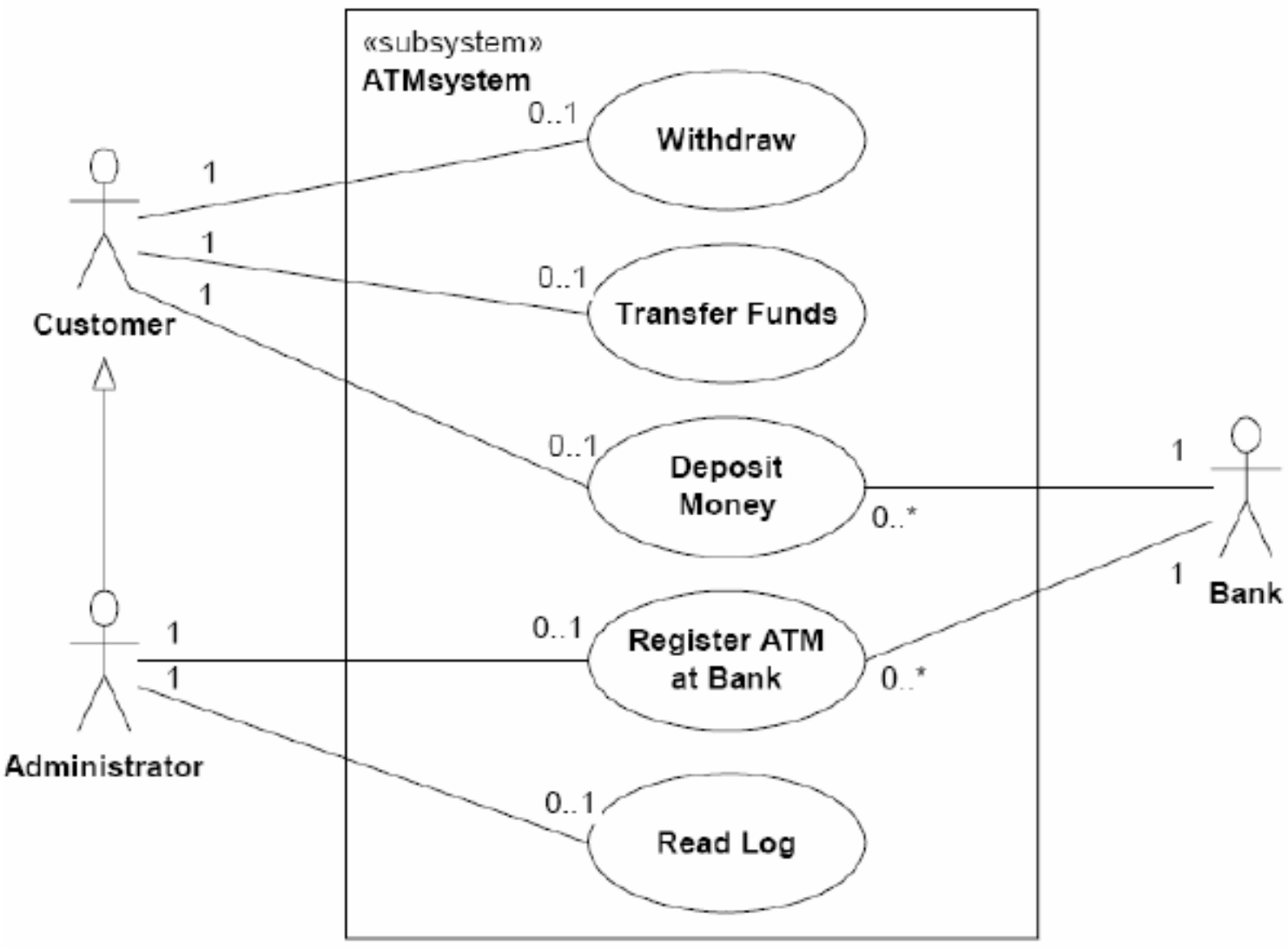
SIMPLE DEPLOYMENT DIAGRAM



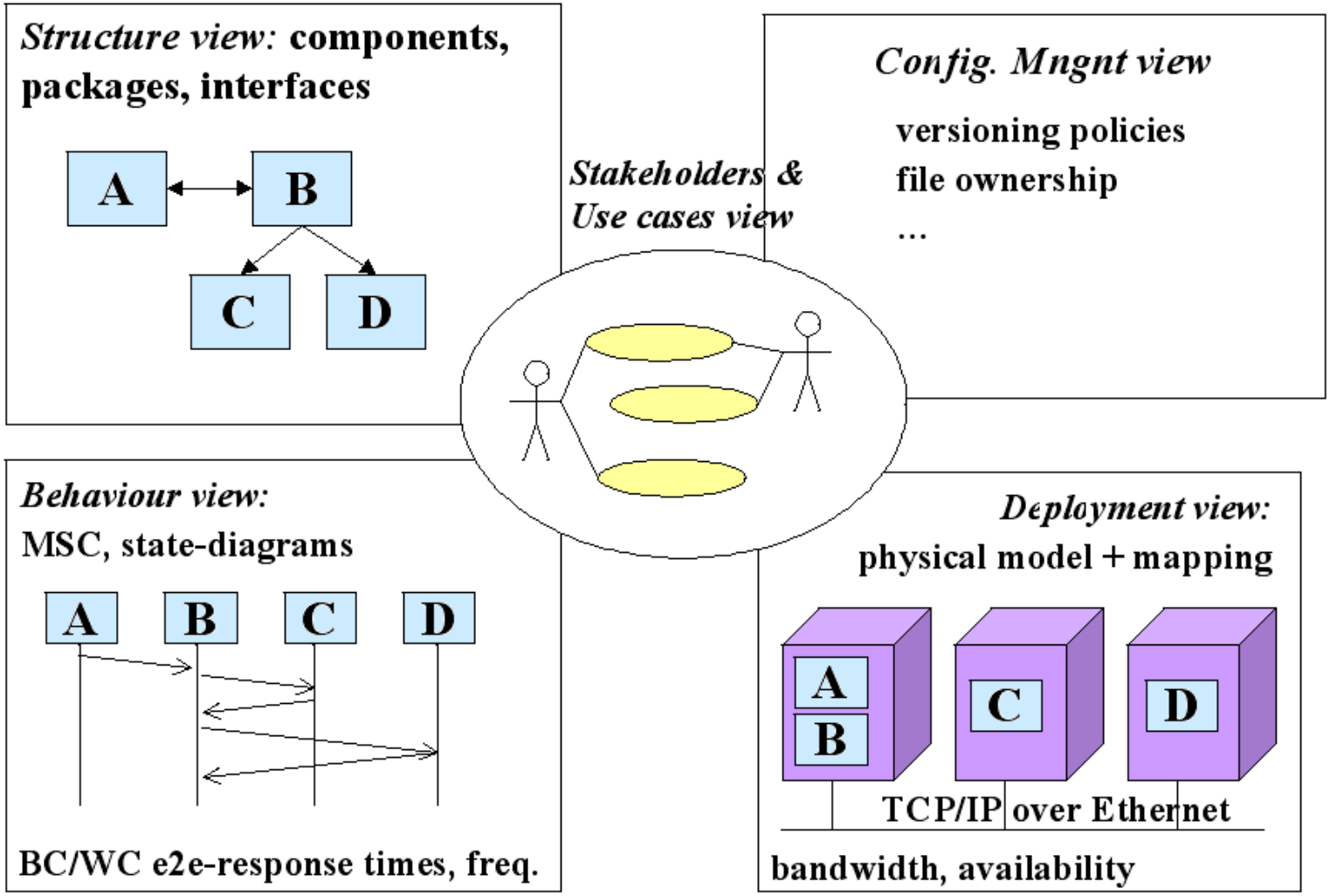
（5）用例视图

- 用例视图（use-case view）用来捕获最终用户所需求的功能性，即“系统应该做什么”
- 用例视图使其他四个视图有机的联系起来；
- 用例视图也称为场景(Scenario)；
- 用例视图主要包括用例图，然后使用若干个交互图来展示每个用例内部的细节。

用例视图


«subsystem»
ATMsystem
Withdraw
Transfer Funds
Deposit Money
Register ATM at Bank
Read Log
Customer
Administrator
Bank
1
0..1
0..*

总结：(Kruchten)4+1视图模型



内容

- 4.1 为什么要进行SA建模？
- 4.2 常用SA描述方法
- 4.3 Kruchten 4+1视图模型
- 4.4 其他常用视图
- 4.5 接口建模（参考《Paul Clements-Documenting Software Architectures Views and Beyond (2nd Edition)》自学）

SA的“视图观”

- **(Kruchten) 4+1模型**
- RSA(统一建模语言UML)
- (CMU-SEI) Views and Beyond模型
- (Hofmesiter) 4视图模型
- (ZIFA)Zachman框架
- 开放分布式处理参考模型(RM-ODP)
- …

CMU SEI
Views and Beyond模型

如何描述SA？

Architectural structures (and hence views) can be divided into three types:

1. **“module” structures** – consisting of elements that are units of implementation called *modules*

2. **“component-and-connector” structures** – consisting of runtime components (units of computation) and the connectors (communication paths) between them

3. **“allocation” structures** – consisting of software elements and their relationships to elements in external environments in which the software is created and executed

Example of Module views

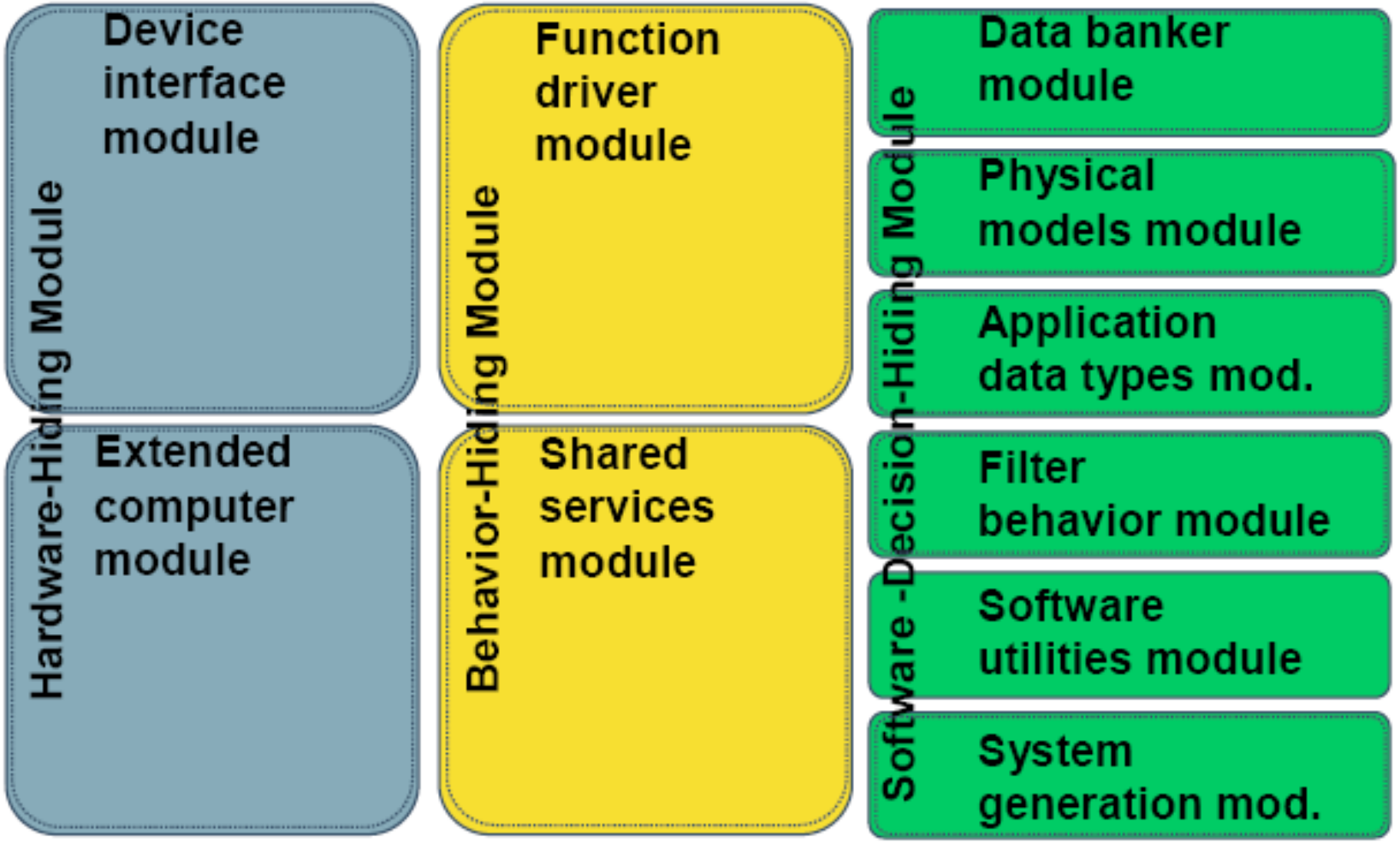
Decomposition view – shows modules that are related via the *"is a submodule of"* relation

Uses view – shows modules that are related via the *"uses"* relation (i.e., one module uses the services provided by another module)

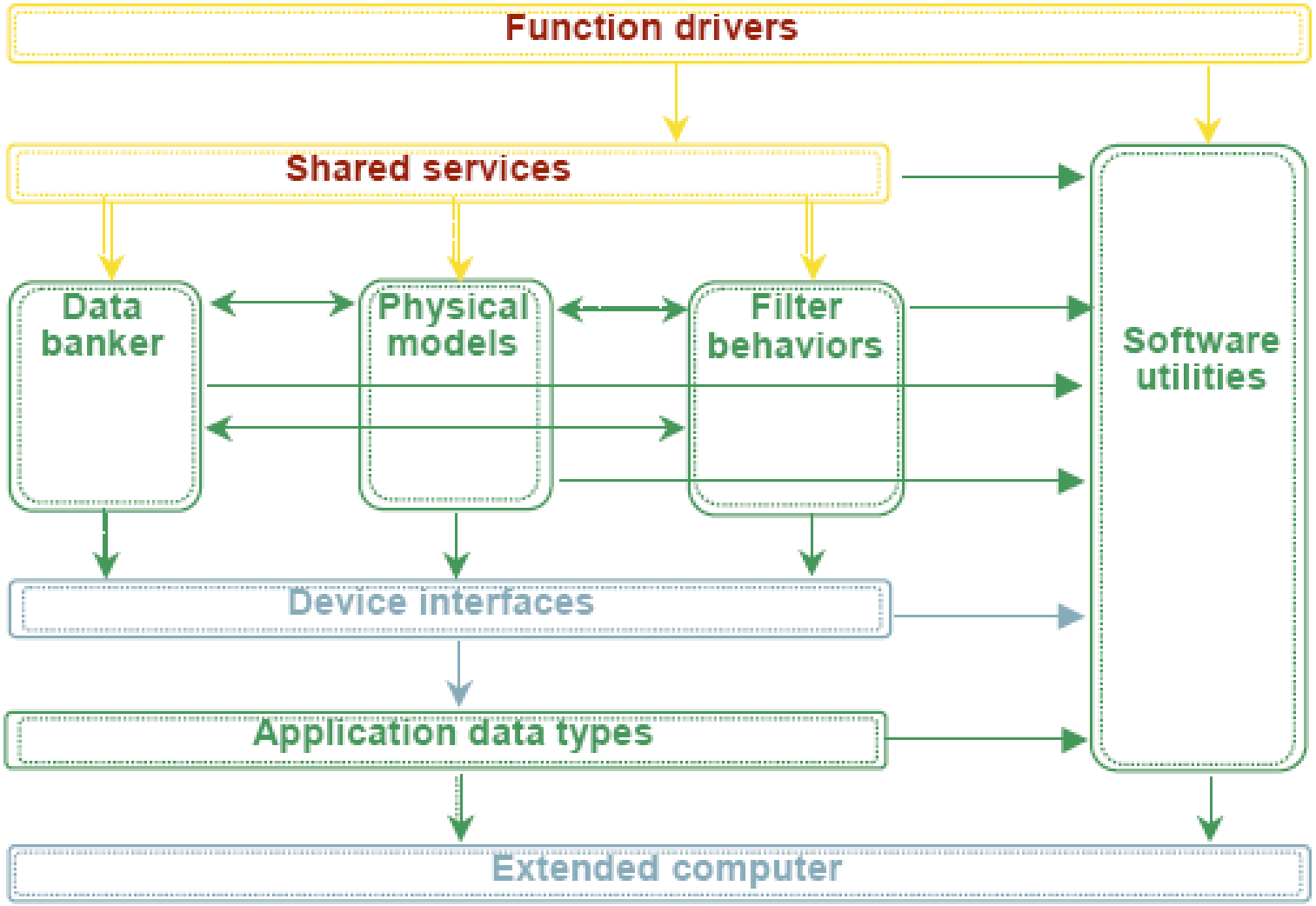
Layered view – shows modules that are partitioned into groups of related and coherent functionality. Each group represents one layer in the overall structure.

Class/generalization view – shows modules called classes that are related via the *"inherits from"* or *"is an instance" of* relations

Module Decomposition View (2 Levels)



Layers View


Function drivers
Shared services
Data banker
Physical models
Filter behaviors
Software utilities
Device interfaces
Application data types
Extended computer

Example of Component-and-Connector Views

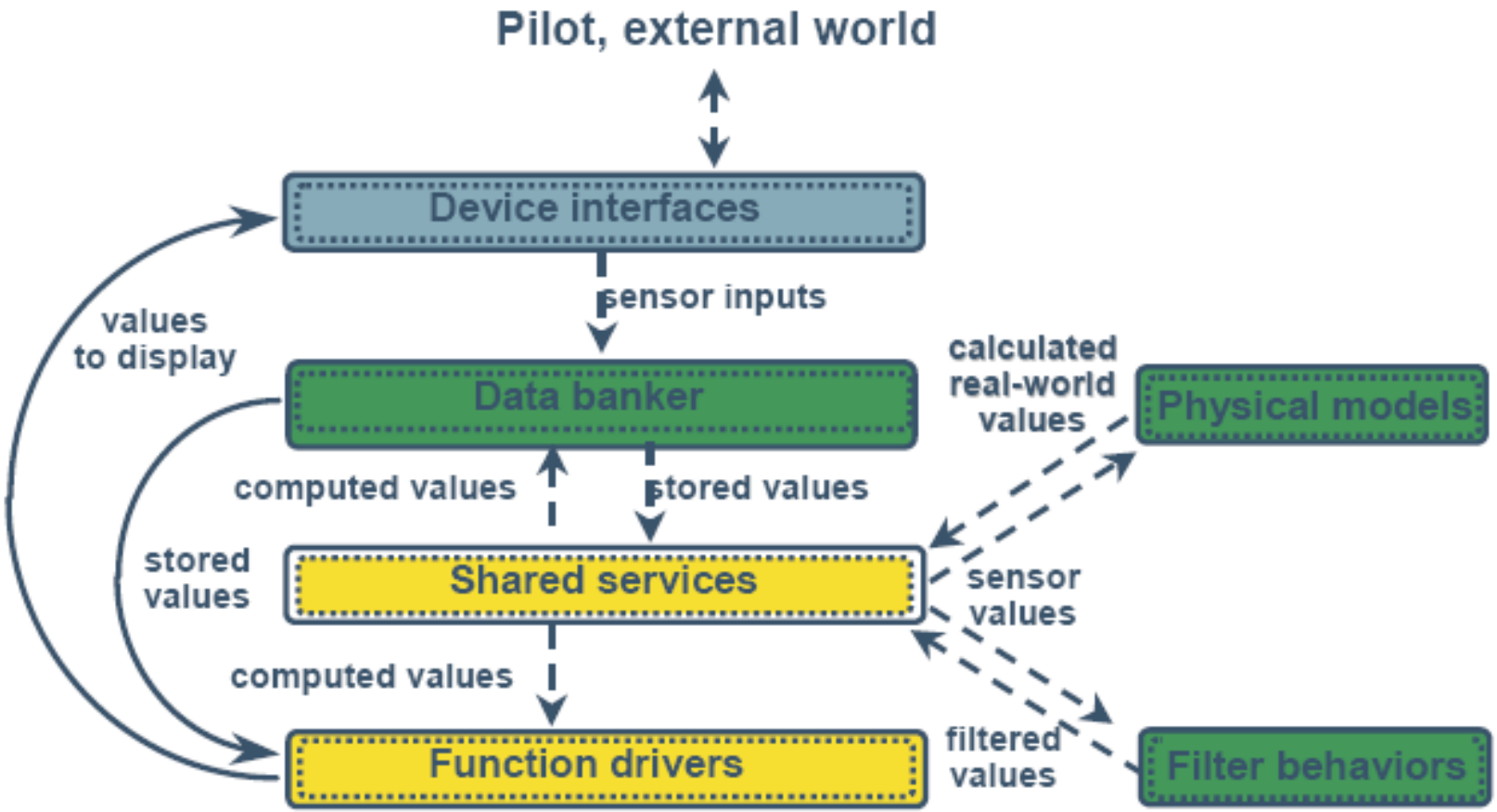
Process view – shows processes or threads that are connected by communication, synchronization, and/or exclusion operations

Concurrency views – shows components and connectors where connectors represent "logical threads"

Shared-data (repository) views – shows components and connectors that create, store, and access persistent data

Client-server view – shows cooperating clients and servers and the connectors between them (i.e., the protocols and messages they share)

C&C Data Flow View



Example of Allocation Views

Deployment view – shows software elements and their allocation to hardware and communication elements

Implementation view – shows software elements and their mapping to file structures in the development, integration, and configuration control environments

Work assignment view – shows modules and how they are assigned to the development teams responsible for implementing and integrating them

Other Views-1

- Siemens Four-Views (Hofmeister, Nord,Soni, Applied Software Architecture, 2000):
  - **Conceptual view**
  - **Module interconnection view**
  - **Execution view**
  - **Code view**

Other Views-2

- Herzum & Sims (Business Component Factory, 1999):
  - **Technical architecture**
  - **Application architecture**
  - **Project management architecture**
  - **Functional architecture**

究竟需要多少视图?

- 适合需要的简单模型
- 不是所有的系统都需要所有的视图
  - 单处理器: 不用分布图
  - 简单处理过程: 不用过程视图
  - 很小的程序: 不用实现视图
- 增加视图:
  - 数据视图
  - 安全视图

究竟需要多少视图？

- An architect can consider the system in at least four ways:

1. How is it structured as a set of code units?

   Module Views

3. How is it structured as a set of elements that have runtime presence?

   Runtime Views

5. How are artifacts organized in the file system and how is the system deployed to hardware?

   Deployment Views

7. What is the structure of the data repository?

   Data Model

内容

- 4.1 为什么要进行SA建模？
- 4.2 常用SA描述方法
- 4.3 Kruchten 4+1视图模型
- 4.4 其他常用视图
- 4.5 接口建模（参考《Paul Clements-Documenting Software Architectures Views and Beyond (2nd Edition)》自学）