软件架构设计的方法论
——分而治之与隔离关注面

胡协刚

中国软件架构师网

首席软件架构师

szjinco@public.szptt.net.cn

Tel: 13828737199

中国软件技术大会

Agenda

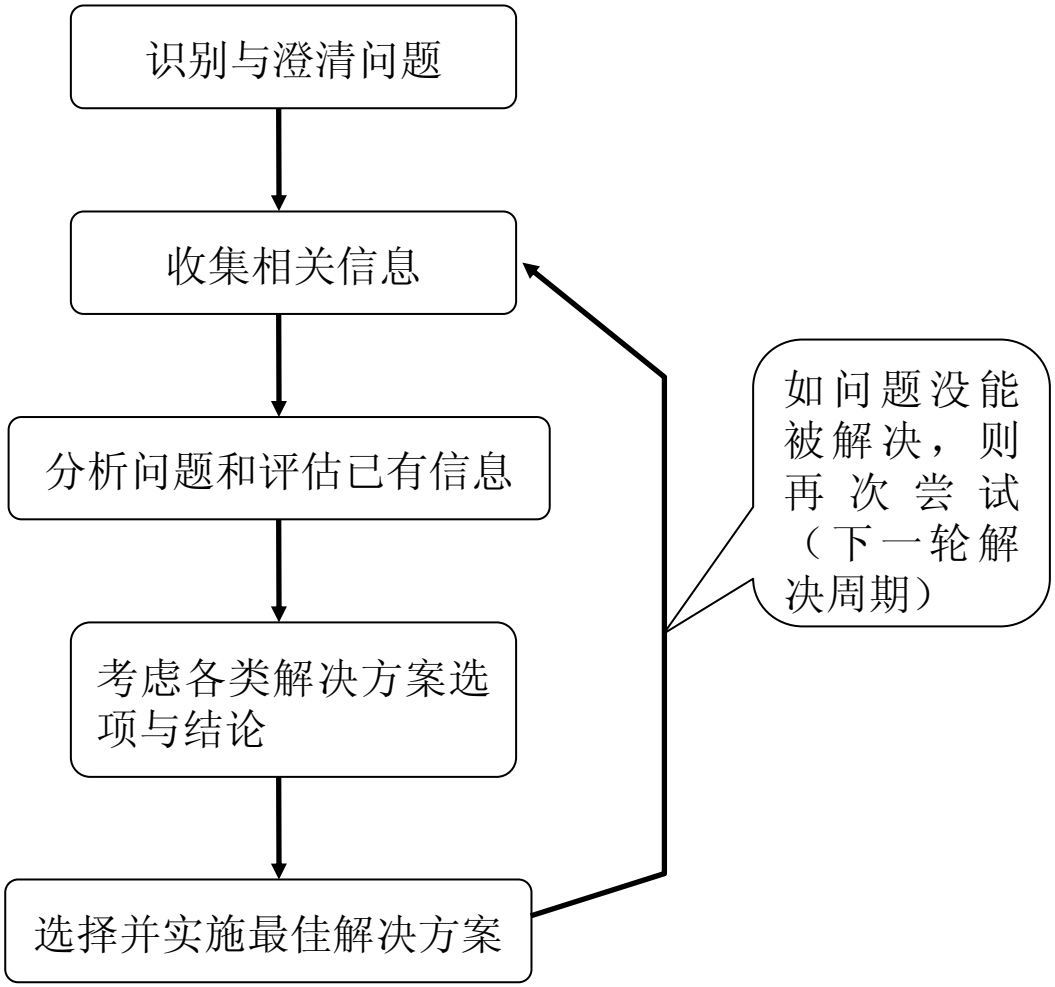
- 问题解决规律
- 软件架构设计中的方法论
- 抽象与软件的表述
- 模块化与分而治之
- 封装和层次化
- 隔离关注面
- 隔离关注面的主要示例——架构中的分层
- 隔离关注面的主要示例——架构机制的抽取
- 隔离关注面的主要示例——AOP与分割横切面

问题解决规律
——人类如何解决复杂问题

问题解决 Problem Solving

- 问题解决是思维的一种形式；当个人或组织不知道如何从已知条件得到目标（期望结果）的时候，“解决问题”这一智力活动便开始了。

问题解决流程示意图


识别与澄清问题
收集相关信息
分析问题和评估已有信息
考虑各类解决方案选项与结论
选择并实施最佳解决方案
如问题没能被解决，则再次尝试（下一轮解决周期）

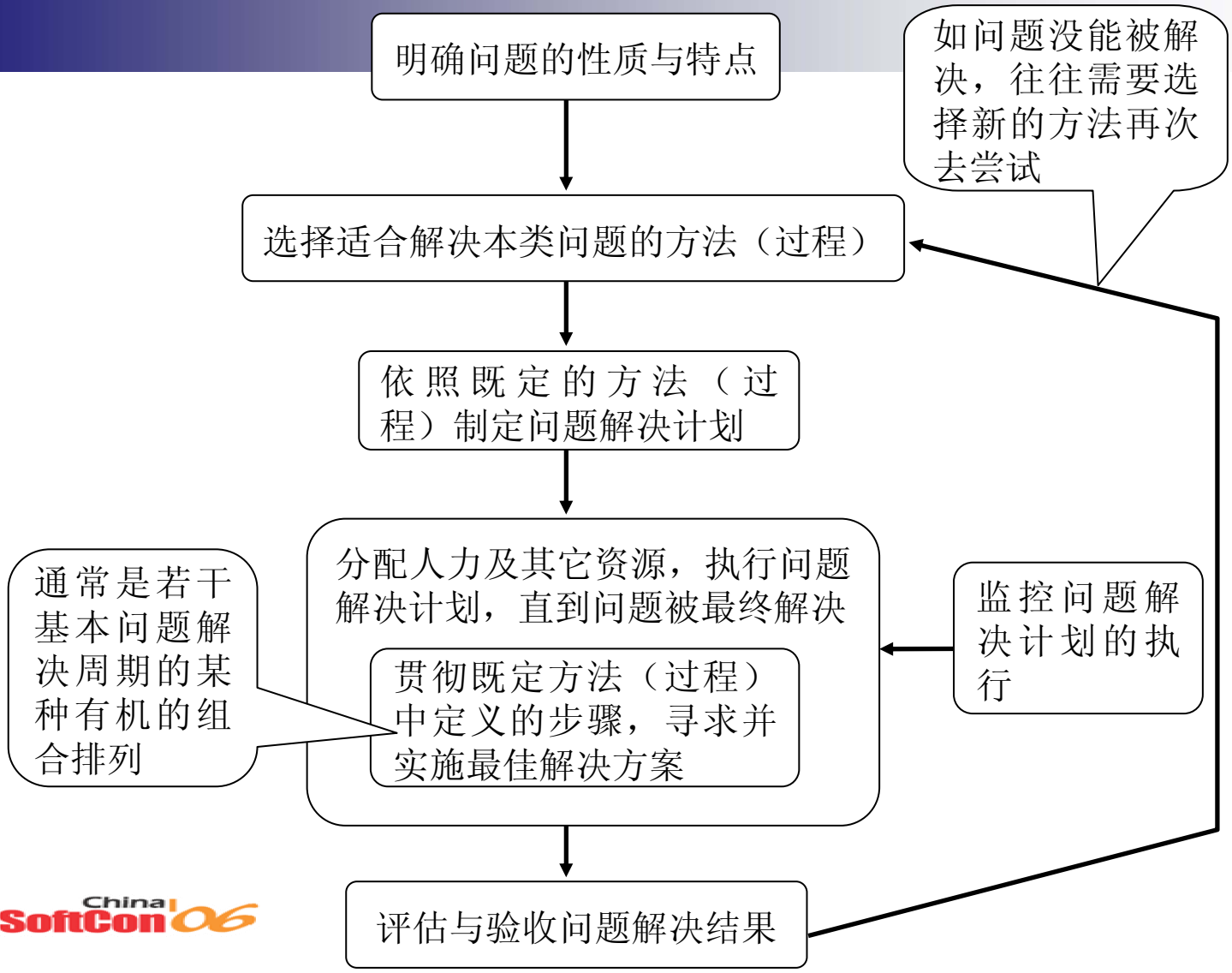
软件开发的方法（理）论框架基础

- 软件被用来解决人们的业务或领域问题，开发软件的过程就是去获得解决业务问题结果的过程。
- 人类在问题解决规律研究方面已经有大量的现成成果；传统的问题解决步骤，实际上为软件开发（解决软件问题）提供了现成的（战略级的）理论方法框架。
- 现代的软件工程技术，其基础实际上就建立在这些问题解决规律之上。

简单问题解决步骤的不足

- 前段中所描述的问题解决步骤，存在一个隐含的假定前提，就是所有问题都能够采用同样简单的方法、步骤来加以解决。对于那些普通、单一的问题而言，这个假定是成立的；但对于软件开发这种复杂和多方面问题交织在一起的情况，假定就不再成立了。
- 实际上，软件具有不一致性与多样性，不同性质的软件，不相同的地方很多，所适应的开发方法也可能完全不同。

先选择方法的问题解决流程示意图



软件开发过程原理

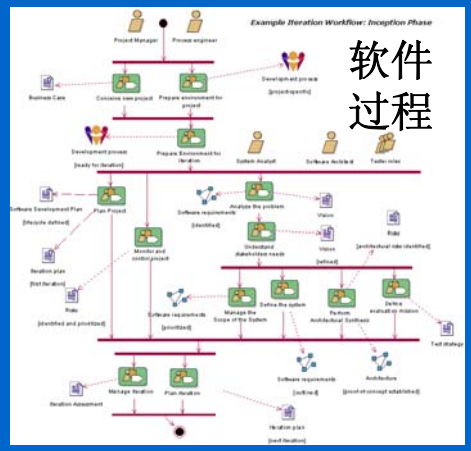
以往软件项目

以往软件项目

业界经验

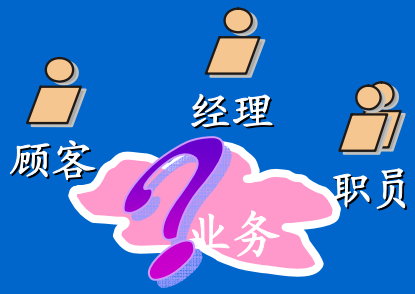
经验总结为规范化软件过程

规范化



QA监督实际开发活动执行了过程规范——确保开发过程质量

实例化

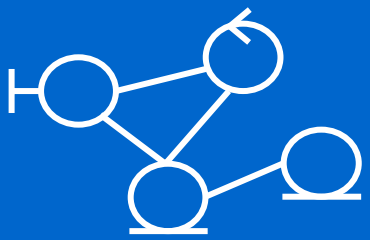


表述（抽象）

QC确保每个中间制品的质量

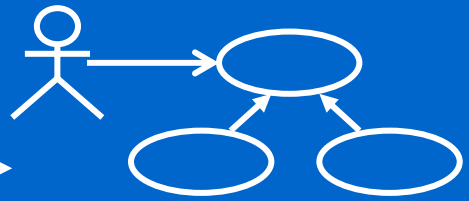
定义（引申）

实现（分析）

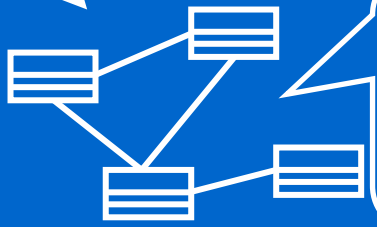


Analysis Model

精化（映射）



Use-Case Model



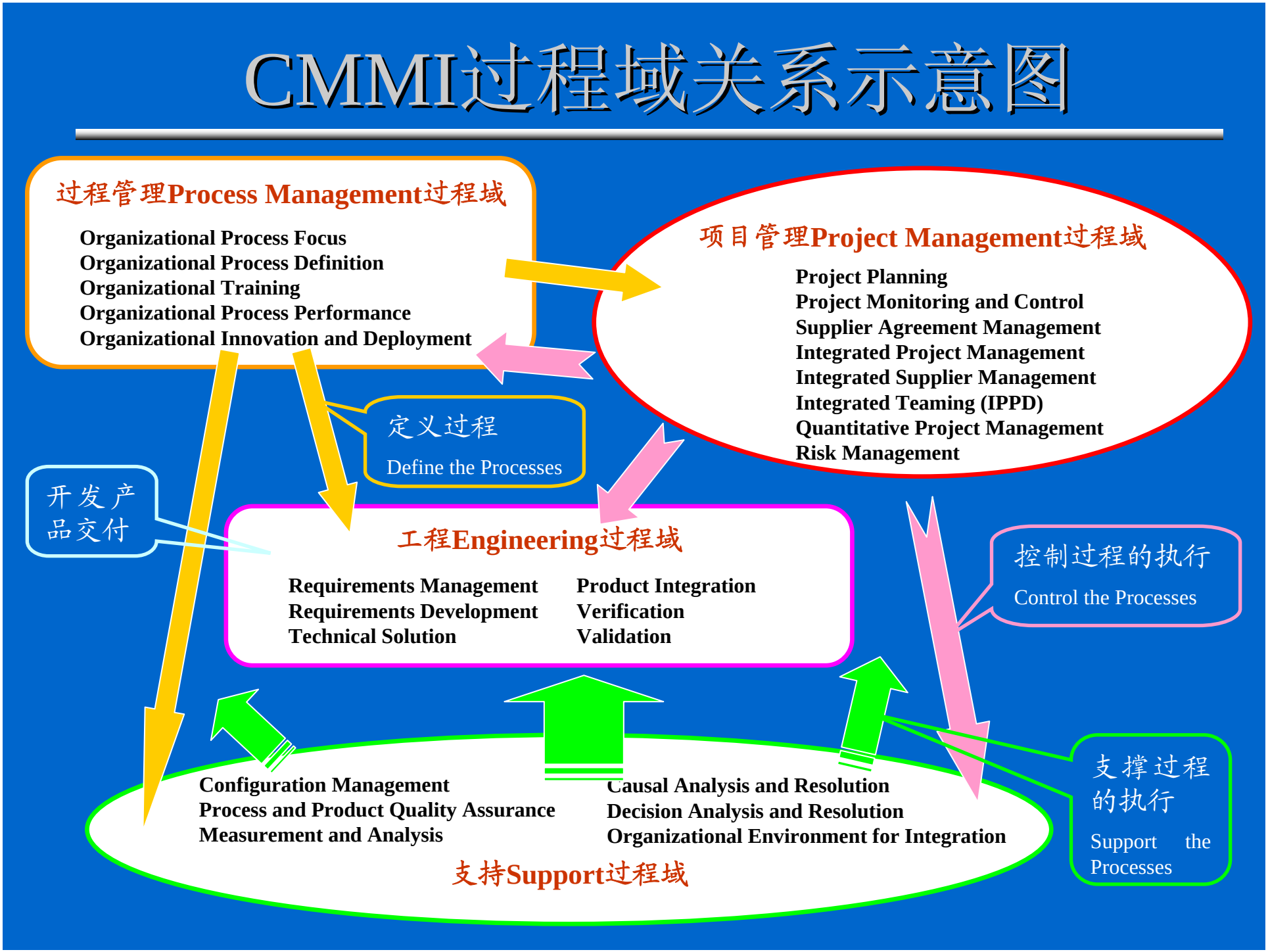
Design Model

实施（转化）

软件交付

QC确保每个中间制品的质量

CMMI过程域关系示意图



软件架构设计中的方法论
——微观思维方法在软件开发中的运用

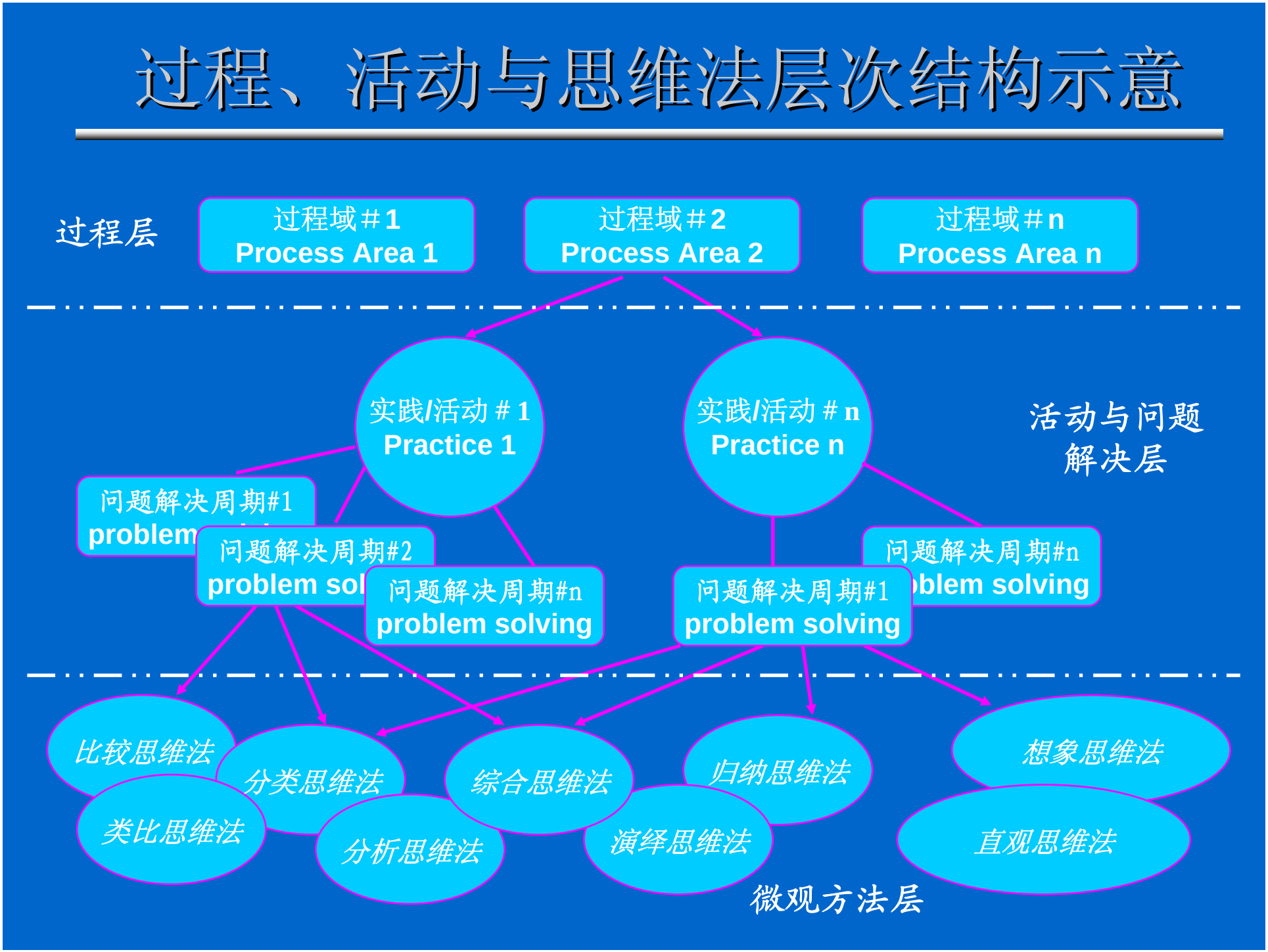
破除软件开发中的神秘主义

- 国内软件界存在一定程度的神秘主义倾向，在自己无力使用常规软件工程途径解决软件问题后，往往简单地将软件开发归于艺术化、玄学化。
- 经常看到的一种典型现象——某个高手熬了几个通宵，终于拿出了一个精巧的设计方案；团队其他成员都很钦佩他，并想向他学习其中设计的技巧；但高手吹嘘说这完全是靠其灵感所得，思考过程毫无逻辑可言。

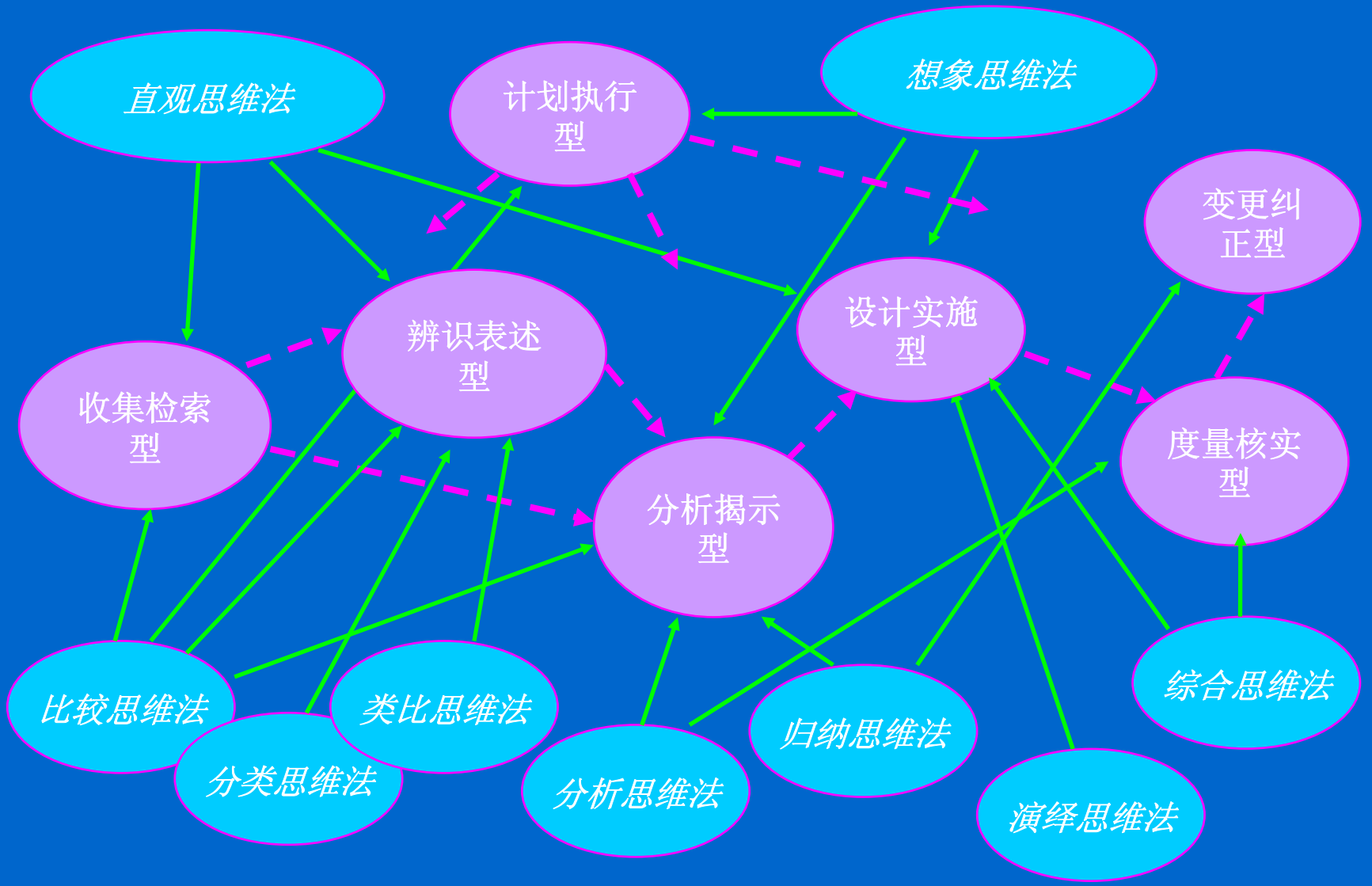
光有宏观的过程、方法还不够

- 无论是简单的问题解决步骤，还是包含了大量实践活动的软件过程域（科目），它们都属于宏观的解决途径。然而，任何软件问题，其最终解决还是要落实到具体的细节之上，这些问题细节需要采用与之相对应的微观方法来解决。
- 例如，架构设计作为技术解决过程域中的一项实践（或者分析设计科目中的一项活动），涉及到大量的模块划分问题；而帮助人们进行划分的方法则主要有比较、分类等思维法。

过程、活动与思维法层次结构示意



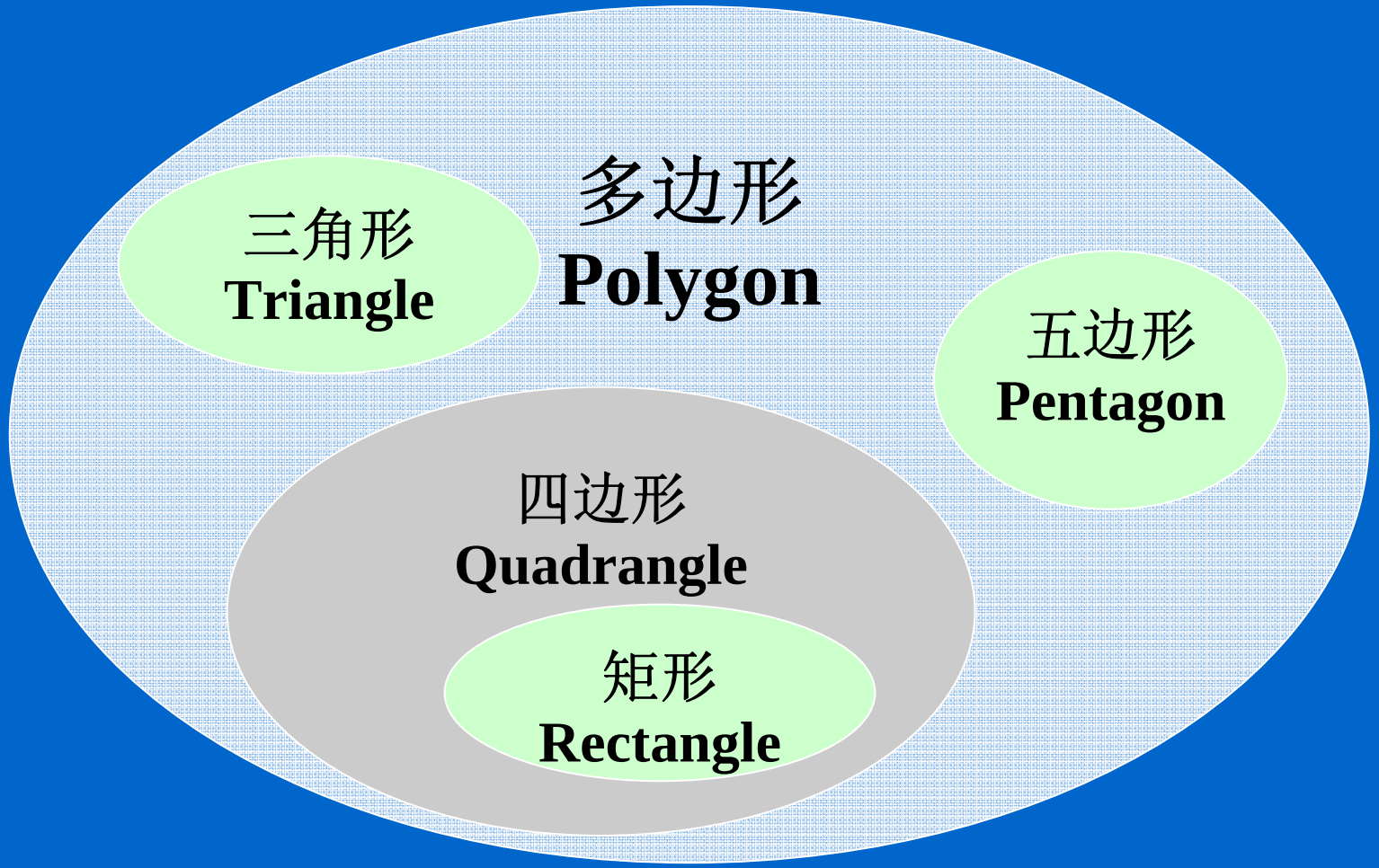
问题解决类型与思维法关系示意



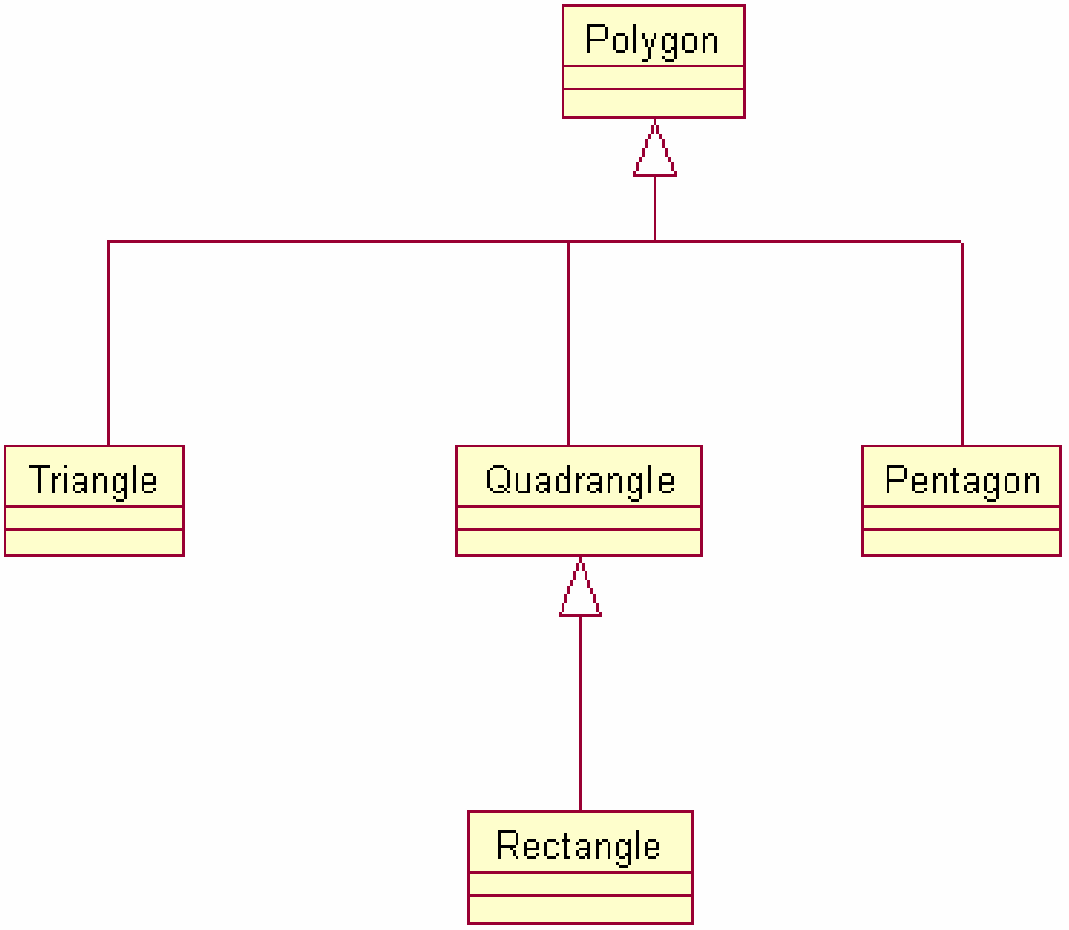
示例：归类法在软件开发中的运用

- 所谓分类是指根据研究对象的共同点和差异点，将其区分为不同种类的一种逻辑思维方法。
- 在分类时，按共同点将研究对象归为较大的类，而按差异点将研究对象区分为不同的较小类（即大类集合中的较小集合、或元素）。

多边形对象集合关系示意

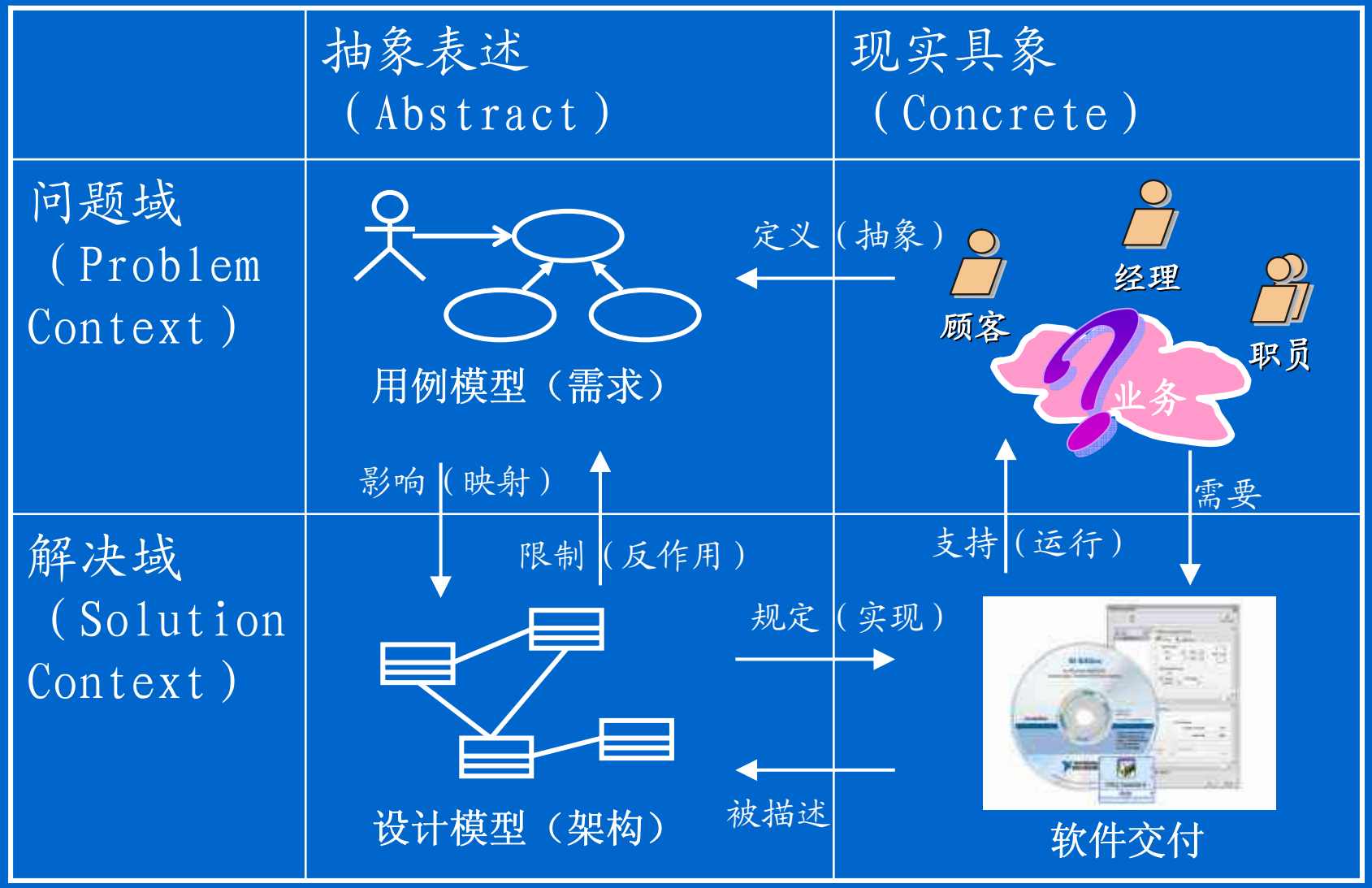


归类法识别的多边形类层次

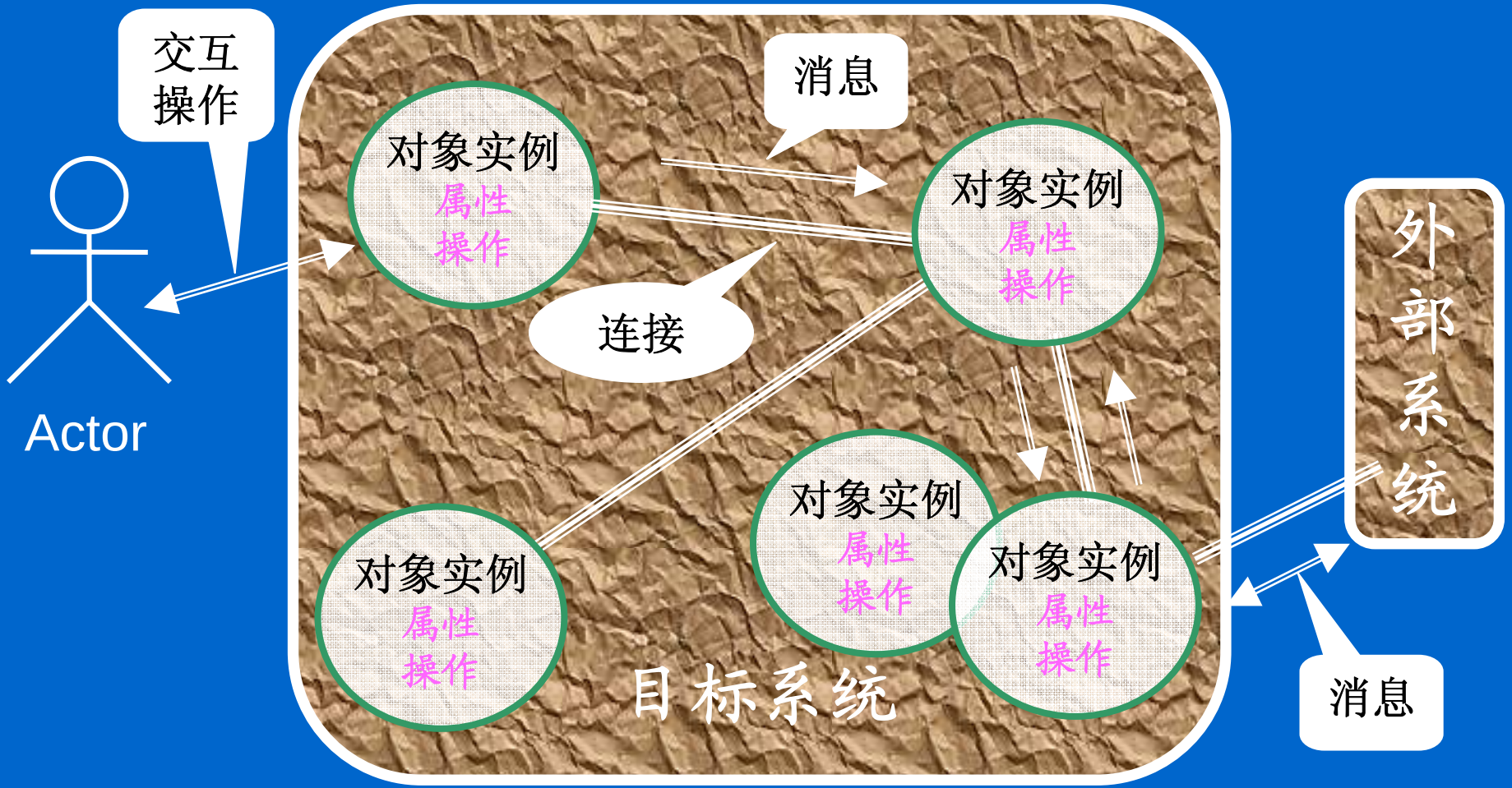


抽象与软件的表述
——抓住软件问题的要害与根本，简化构架设计处理的内容

软件表述——开发的前提条件



基于对象的软件系统观


交互
操作
Actor
消息
对象实例
属性
操作
连接
对象实例
属性
操作
对象实例
属性
操作
对象实例
属性
操作
对象实例
属性
操作
目标系统
外部系统
消息

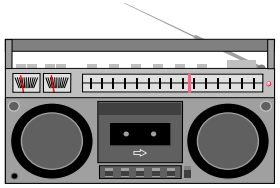
抽象abstraction

- ✓所谓抽象是指关注事物中与问题相关的部分（通常是一个角度或方面），而忽略其它不相关内容（细节）的一种思考方式;
- •抽象依赖于选择的问题领域和角度;
- •在面向对象的范式中，使用从问题域的抽象来表达目标系统（例如对象和类）。

示例：抽象

Salesperson

Customer



Product

模块化与分而治之
——构架设计中的管理学，以分工协作来扩展团队解决复杂软件问题的能力

抽象解决不了软件中的规模问题

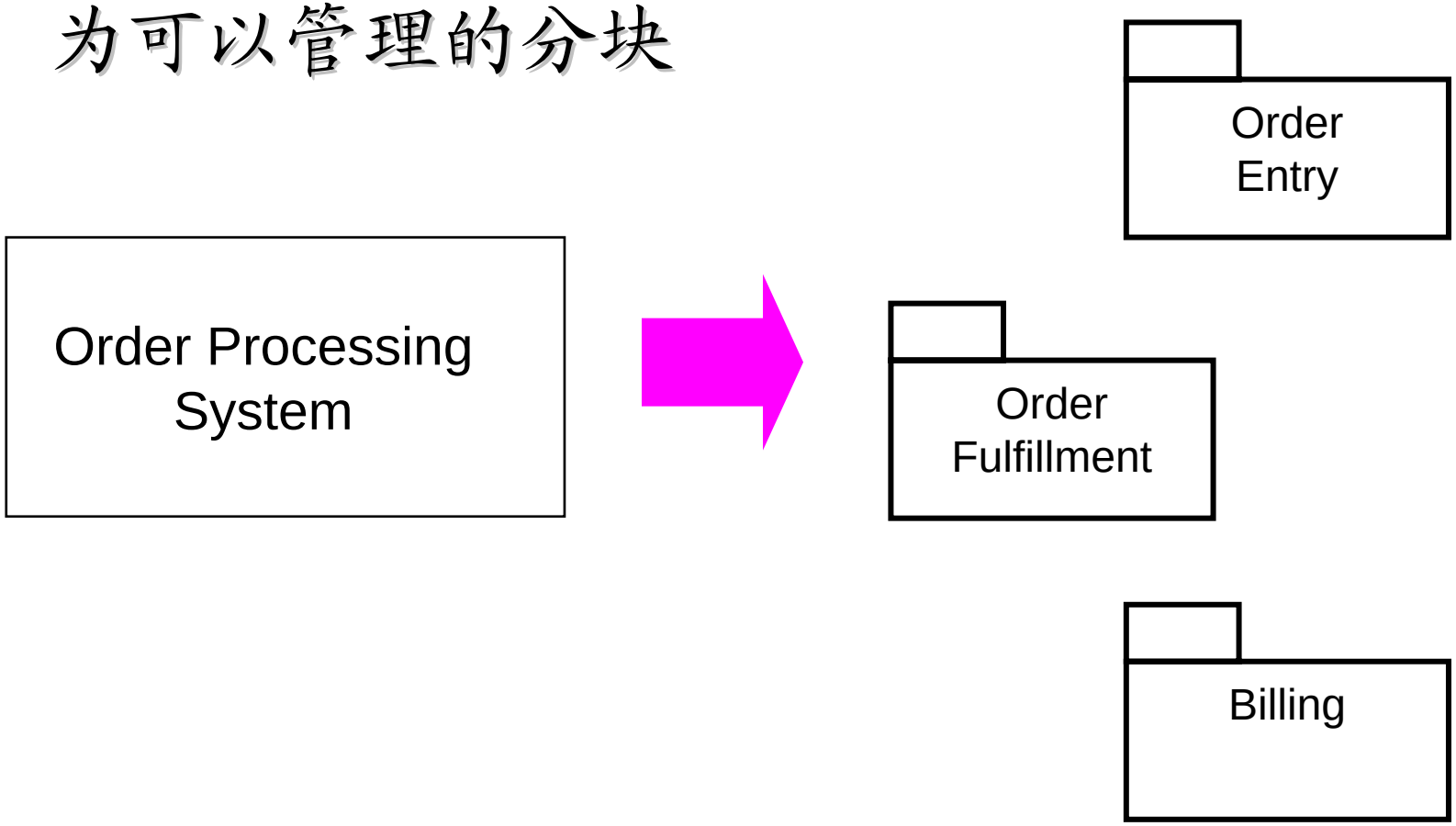
- ✓抽象能够很好地降低单个软件元素的复杂度，但却解决不了软件中的规模问题
- 根据7＋/－2原则，人能够同时处理的元素是5～9个，当软件中对象（种类）的数目超过一定限度后，人们就无法同时去思考和处理这些对象了。
- ✓模块化通过划分，帮助人们将同时要考虑的对象数目减少到普通智力的人能够胜任自如的地步。

模块化Modularity

- ✓所谓模块化是指逻辑和物理上将事物分解为更小、更简单的分组（例如需求和类），从而满足软件工程化管理的需要;
- •模块化的实质——“分而治之”;
- •在面向对象的范式中，子系统、包、构件和对象本身都是模块化的实体元素。

示例：模块化

- 将较大规模、复杂的系统分解为可以管理的分块



封装和层次化
——构架设计中的划分与组织技巧，帮助团队不断提升所开发软件的规模数量级

模块化带来模块之间关系处理问题

- ✓将系统划分为若干模块后，人们可以分别在各模块内部处理那些数量已大幅减少的元素；但为了确保系统的完整一致，人们同时还必须在系统层面来处理这些模块及其之间的关系。
- 根据7 + / – 2原则，人能够同时处理的元素是5 ~ 9个，但此间实际上还有一个隐含的条件，就是每个元素本身的复杂度也不超过一定限度。由此，如果模块对外展现的内容过多，人们仍然是无法在系统层面处理这些模块（及其关系）的。
- ✓封装在模块化划分的基础上进一步简化了模块本身对外暴露的复杂度。

封装Encapsulation

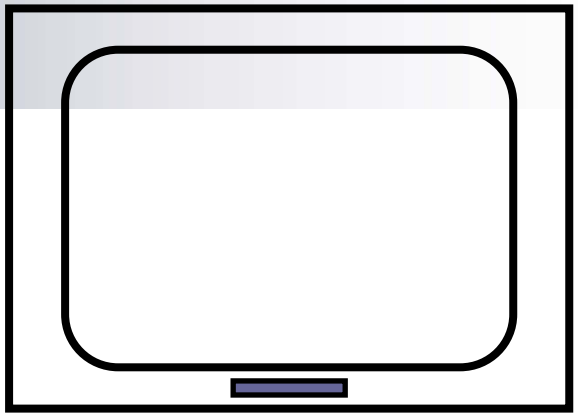
- ✓所谓封装是指将特性（属性、行为）在物理上局限于一个单独的黑盒抽象中，且将它们的实现（和相关的设计决定）隐藏于公共接口背后;
- ✓在面向对象的范式中，从系统、子系统到对象都拥有封装的特性。

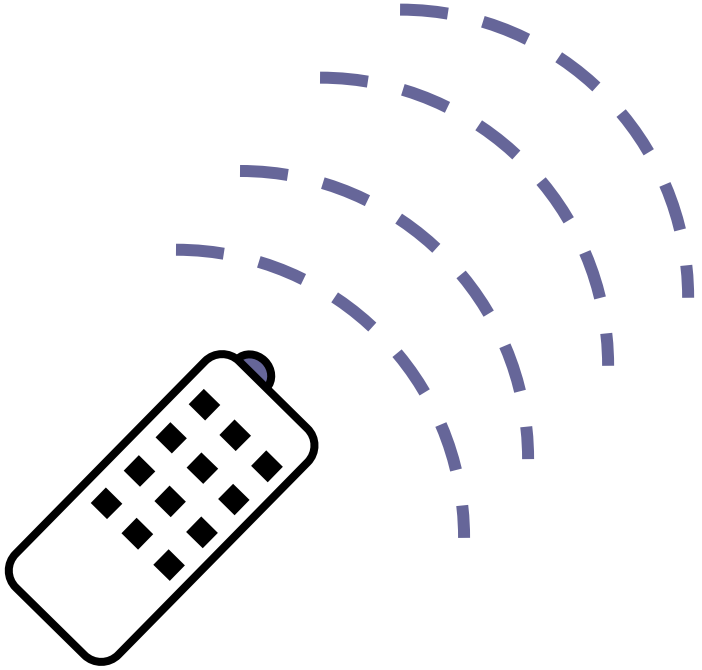
●抽象类（还有接口）使得客户代码client code只需要关注对象外在行为中与它相关的部分（忽略对象固有的但并不相关的其它外在行为，例如从其它类继承的操作）；

●对象的封装机制则封闭了对象所有的内部结构和内部行为，使得客户代码无需关注对象内部的细节。

示例：封装

- 将实现从客户角度隐藏
  - 客户只依赖于接口





抽象 vs 封装

✓抽象和封装都是一种简化问题的思维模式，但它们实现简化的方式不同:

- 抽象剔除了不相关的内容，以突出重点;
- 相对应地，封装则隐藏了相关但不需要被知道（不能被剔除）的内部细节，以减少依赖。

✓剔除与隐藏对于外部（客户）而言，其效果是等价的

简单模块化仍处理不了大规模问题

- ✓模块化通过划分来简化问题，但是简单的模块化途径仍然处理不了大规模的事物
- 根据7 + / – 2原则，人能够同时处理的元素是5 ~ 9个；那么针对一个具有1000个元素的系统，我们当然可以将其划分为150个模块，这样每个模块的元素不超过7个，可以交给150个人来分担完成；但是同时为了确保系统的完整和一致，需要有人来协同150个人的工作，而这显然...
- ✓层次化为模块化划分提供了一种递归途径，以支持处理更大规模的事物

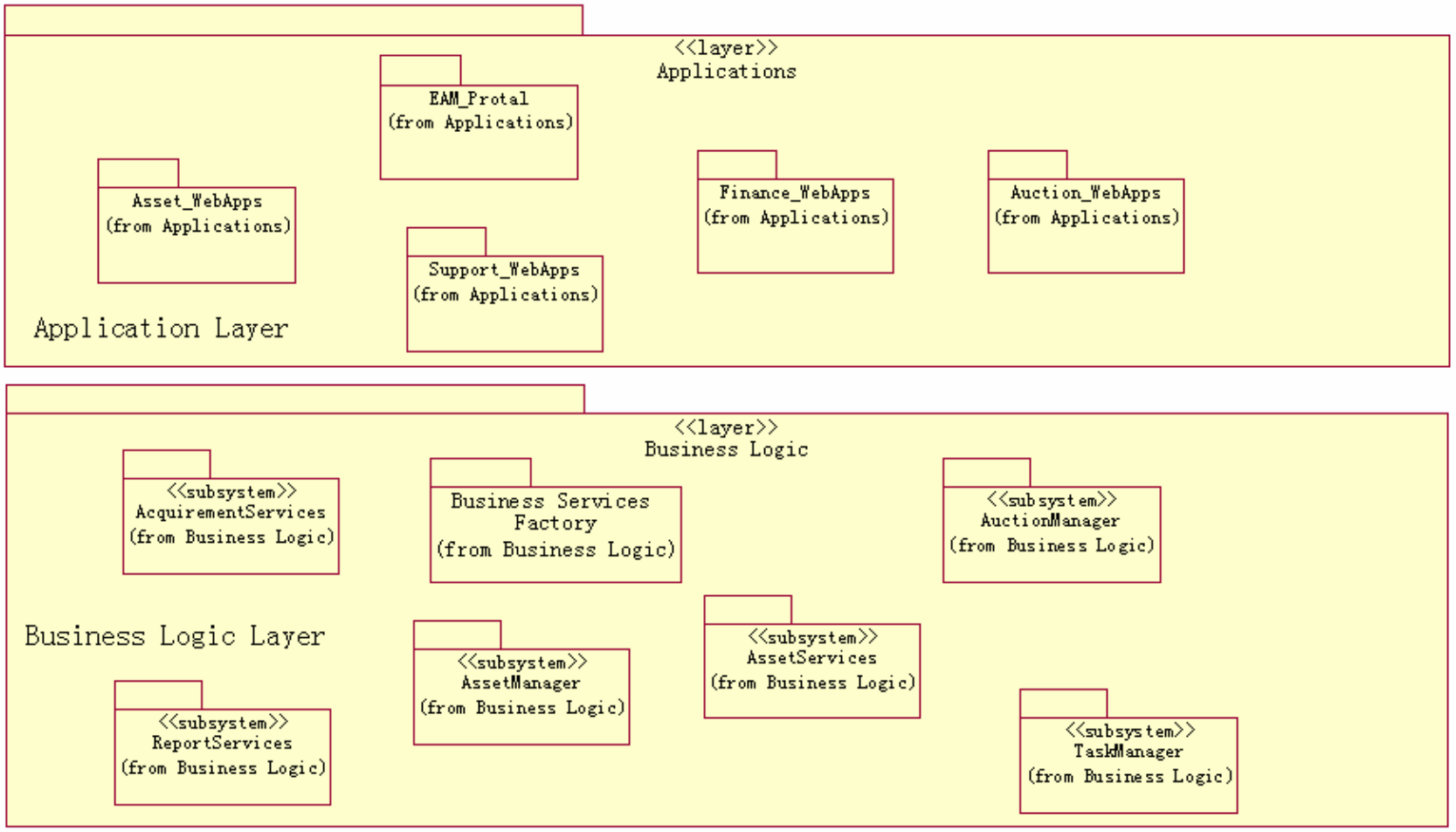
层次化Hierarchy

✓所谓层次化是指任何组织成为一种树状结构的抽象级别或顺序;

✓在面向对象的范式中，层次结构包括:

- 聚合层次结构Aggregation hierarchy、
- 包容层次结构containment hierarchy、
- 类层次结构class hierarchy、
- 泛化层次结构generalization hierarchy、
- 继承层次结构inheritance hierarchy、
- 具体化层次结构specialization hierarchy、
- 划分层次结构partition hierarchy、
- 类型层次结构type hierarchy;

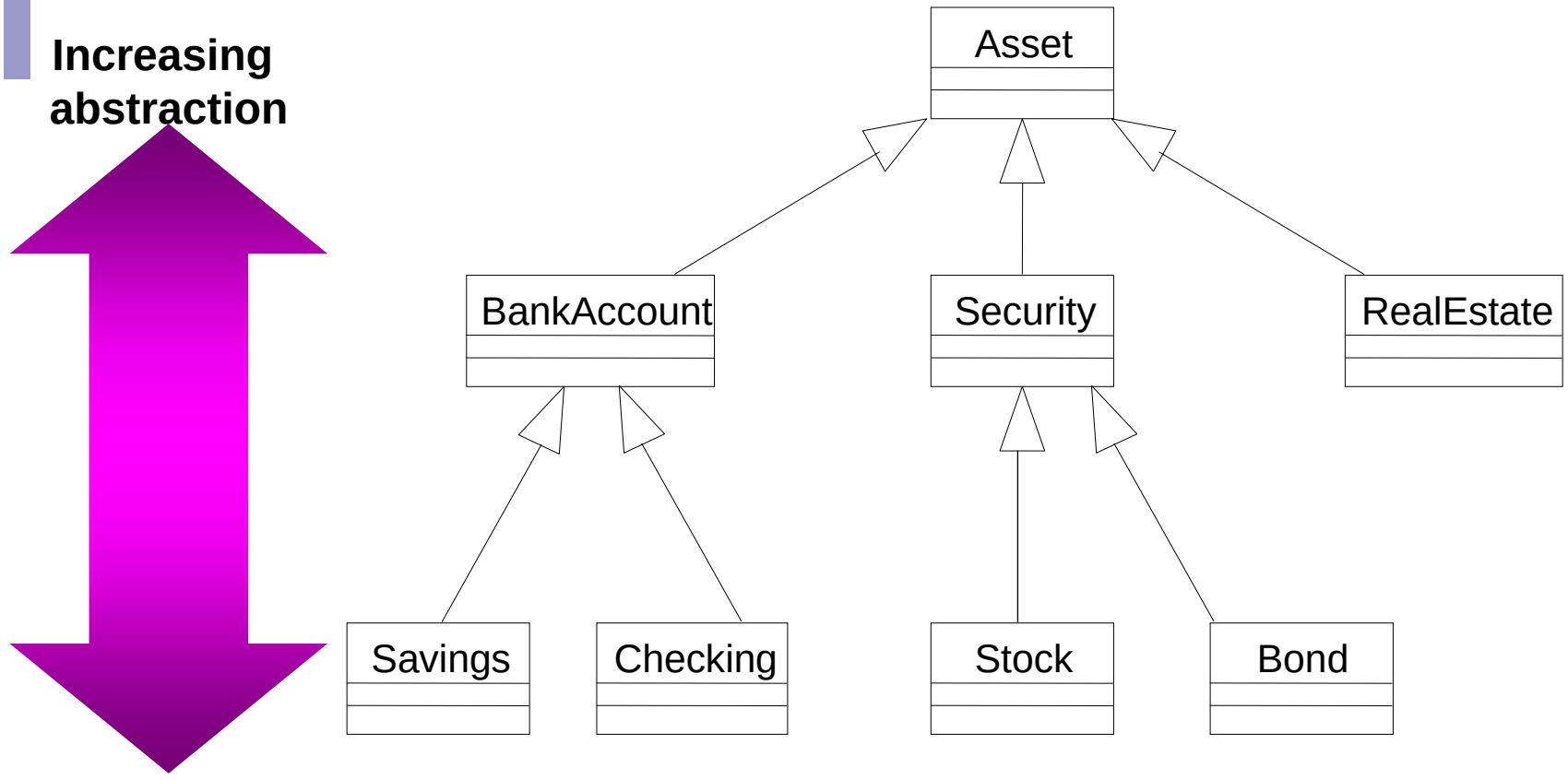
示例：划分（包含）层次化



应用层次化的概念来实行模块化的递归划分—
—粒度相同的包分布于同一级的父包中

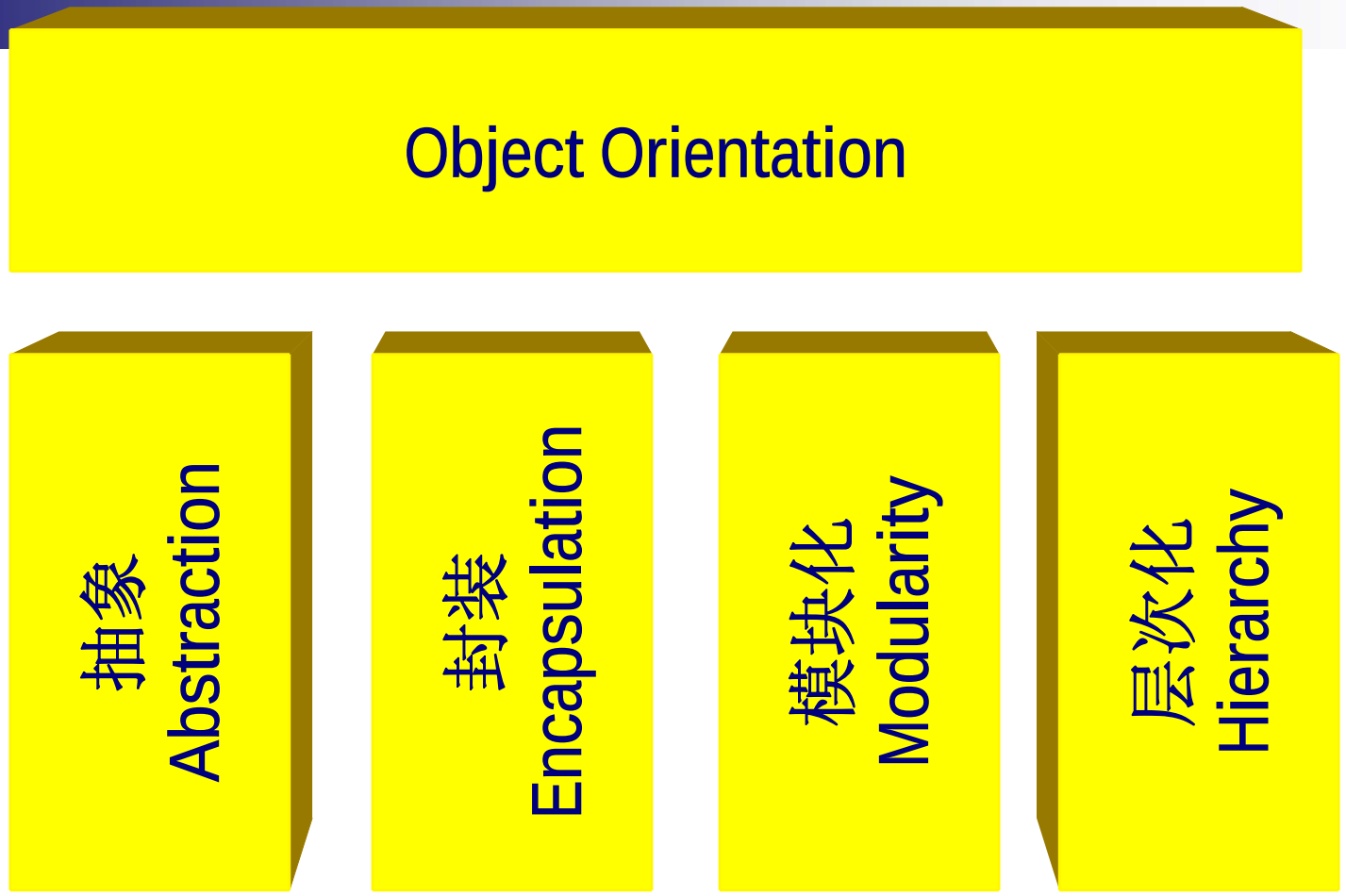
示例：泛化（抽象）层次化

- 抽象的层次



在层次结构中级别相同的类应当处于同样的抽象级别——本质上是一种职责的划分

面向对象的基本原理Principles



隔离关注面

——构架设计中的指导原则，将交织、混沌的软件元素关系梳理清楚，使得性质不同的关注面被分割而独立

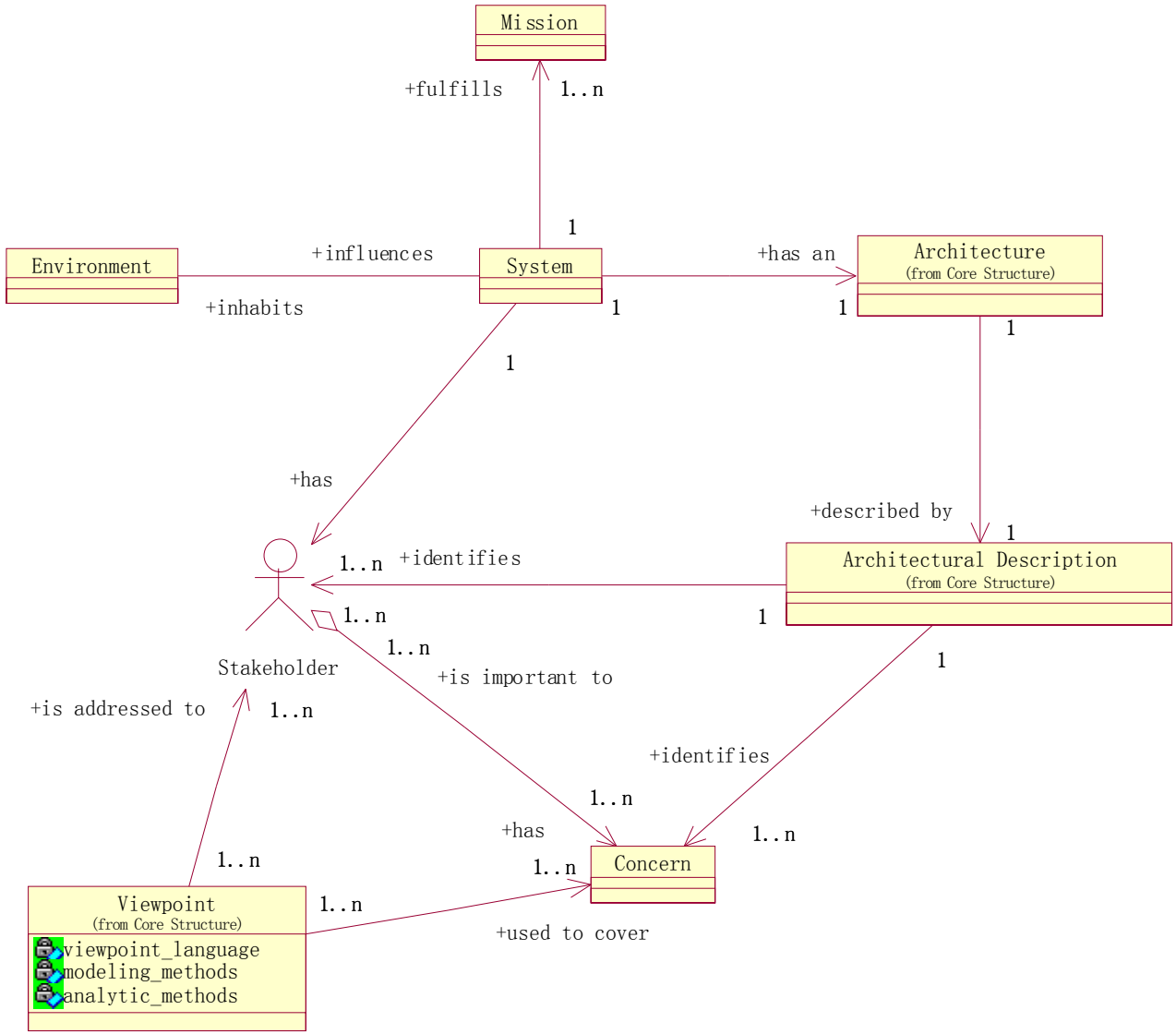
软件中的关注面Concerns

- 软件中包含的内容纷繁复杂、规模巨大；前述的抽象、模块化、封装、层次化等途径，实际上是从有形的实体结构方面帮助人们简化问题；然而软件还包含其它非实体（例如功能、性能、经济性等）方面Aspects的内容，为了简化这些方面的问题解决，人们需要研究其它的方法和途径。
- 软件中的不同方面，往往被不同的涉众所关心，于是将其抽象为所谓的“关注面”。

涉众对目标系统的关注面示意

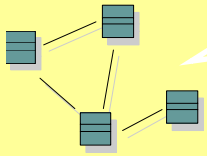
任何系统（System）都生存于特定环境（Environment）中，以实现某种任务（Mission），它拥有自身的构架（Architecture），并被构架描述（Architecture Description）所定义；

系统的涉众（Stakeholder）则对系统的构架存在不同的关注焦点（Concern），而视点（Viewpoint）提供了涉众观察构架的视角，并覆盖了他们的关注焦点。



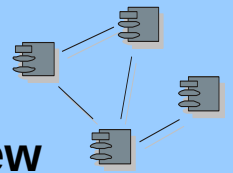
不同视图代表了不同涉众关注面

抽象、模块化、封装、层次化等途径最适合解决的就是逻辑上的结构方面问题



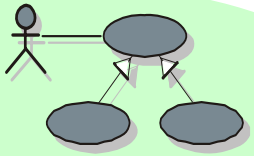
Logical View

Analysts/Designers
Structure



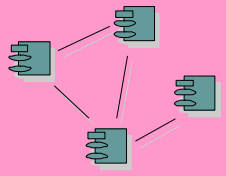
Implementation View

Programmers
Software management



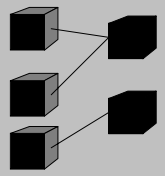
End-user
Functionality

Use-Case View



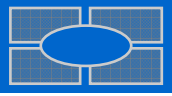
Process View

System integrators
Performance
Scalability
Throughput

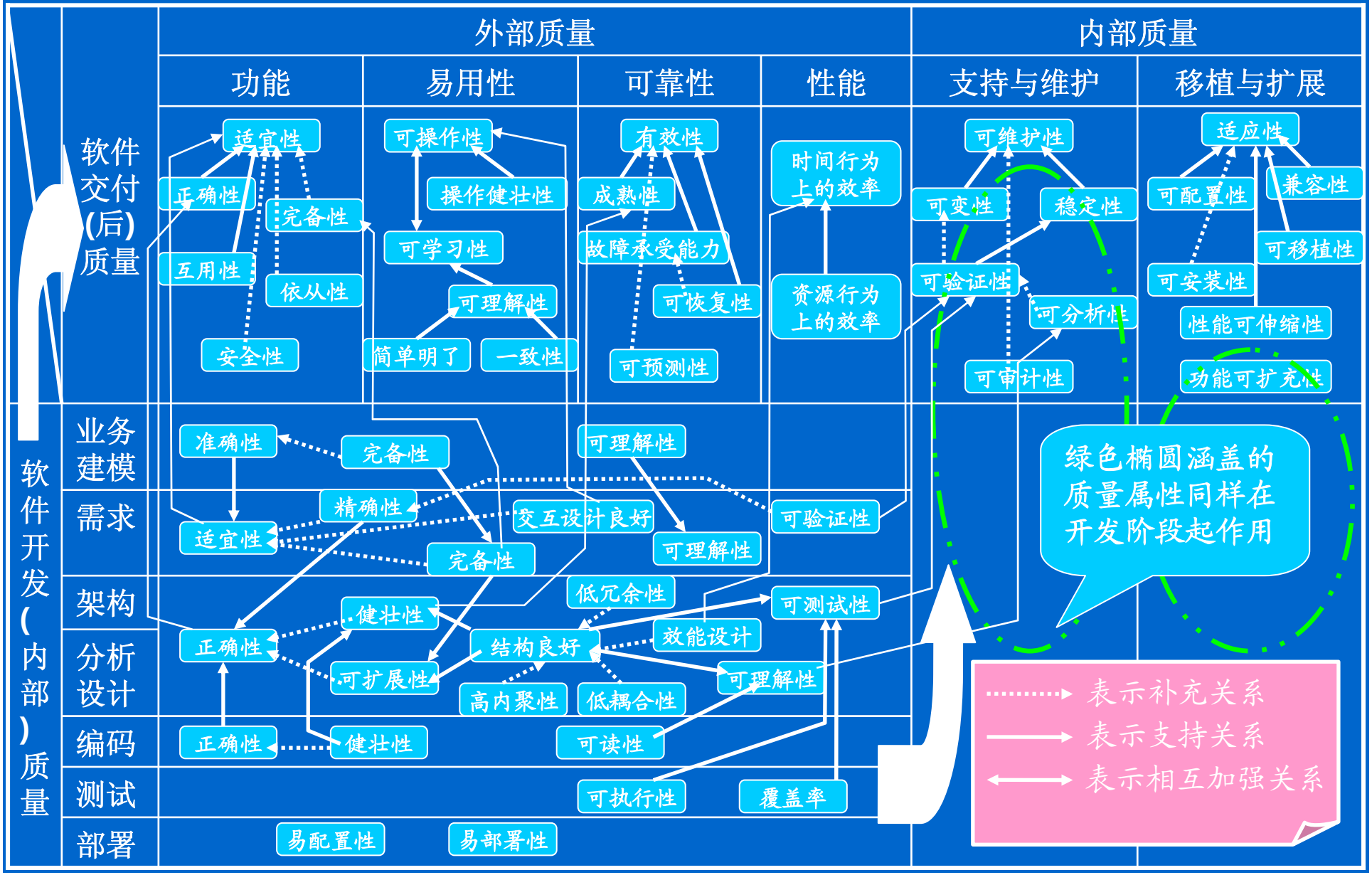


Deployment View

System engineering
System topology
Delivery, installation
communication



不同质量属性被不同涉众所关注



软件中胶着一体的各个方面Aspect

- 实体对象有明显的边界，可以方便地被分割。而方面却不是实体，其实质上只是附着在某个实体上的各类属性事物。例如，一个业务应用中的实体类，它需要被持久化（保存在数据库中），还要满足权限控制的要求，最后为了方便测试，还要支持日志记录等等；这些都是同一客体的不同侧面，它们不能独立于客体而存在。
- 另外，软件中的不同方面Aspects内容，常常相互渗透而交织一起。例如，上述实体类中的实现持久化代码与安全的就可能混合在一块；又如目标系统的进程组织就受其逻辑结构影响。

隔离关注面Separation of Concerns

- 软件中的众多方面同时存在，并相互胶着，这使得开发活动变得更为困难和复杂。
- 前述“分而治之”已经给了我们启示——必须将复杂和大规模的问题分解开来，分别一一加以解决。
- 然而方面不是实体，不能在物理上进行分割；不过，我们还是可以从逻辑上来将同一方面的内容组织在一起，并与其它方面的内容分隔开来。
- 无论是实体还是逻辑上的方面，都是涉众的“关注面”，由此我们引入一个统一的概念——“隔离关注面”。它使得人们能够分别地处理被分解的小问题，或同一问题的不同方面。

隔离关注面的不同途径

- 在隔离关注面时，对于关系明确的场合，简单地分割就行了；但对于关系交织一起，需一同考虑的场合，则要先做概略的总体考虑，然后再分开做处理。
- 隔离关注面的途径包括:

内外的隔离——部件对外公开的行为、及其与其它部件之间的关系，与其自身的内部细节，被分开来处理。

隔离关注面的不同途径

- 隔离关注面的途径包括:

空间上的隔离——软件部件的分割（模块化）、同一部件不同侧面的分割（例如某个类的逻辑包结构与其源码目录文件被分别考虑）等等。

时间上的隔离——划分良好的部件，可以安排在不同的时间来完成；由于部件之间的依赖较弱，进而能够并行来进行（开发）。时间上的隔离首先依赖于空间上的隔离。

隔离关注面的不同途径

- 隔离关注面的途径包括:

质量因素的隔离——同时满足不同的质量要求是非常困难的。一则，同时为实现多个质量约束而设计将过于复杂而难以开展。二则，不同的质量属性之间可能是相互冲突的，要同时满足它们几乎是不可能的。因此，我们应当在不同的时机，分别去考虑实现不同的质量要求，即隔离质量关注点。

- 隔离关注面的其它途径还有：问题域与解决域（实现域）分离、人员职责与技能要求的划分等等

隔离关注面的主要示例
——架构中的分层

层次模式与架构中的分层

- 分层模式是一种将系统的行为或功能以层为首要的组织单位来进行分配（划分）的结构模式
  - 通常在逻辑上进行垂直的层次Layer划分，
  - 在物理上则进行水平的层级Tier划分
- 分层要求
  - 层内的元素只依赖于当前层和之下的相邻层中的其它元素

    （注意这并非绝对的要求）

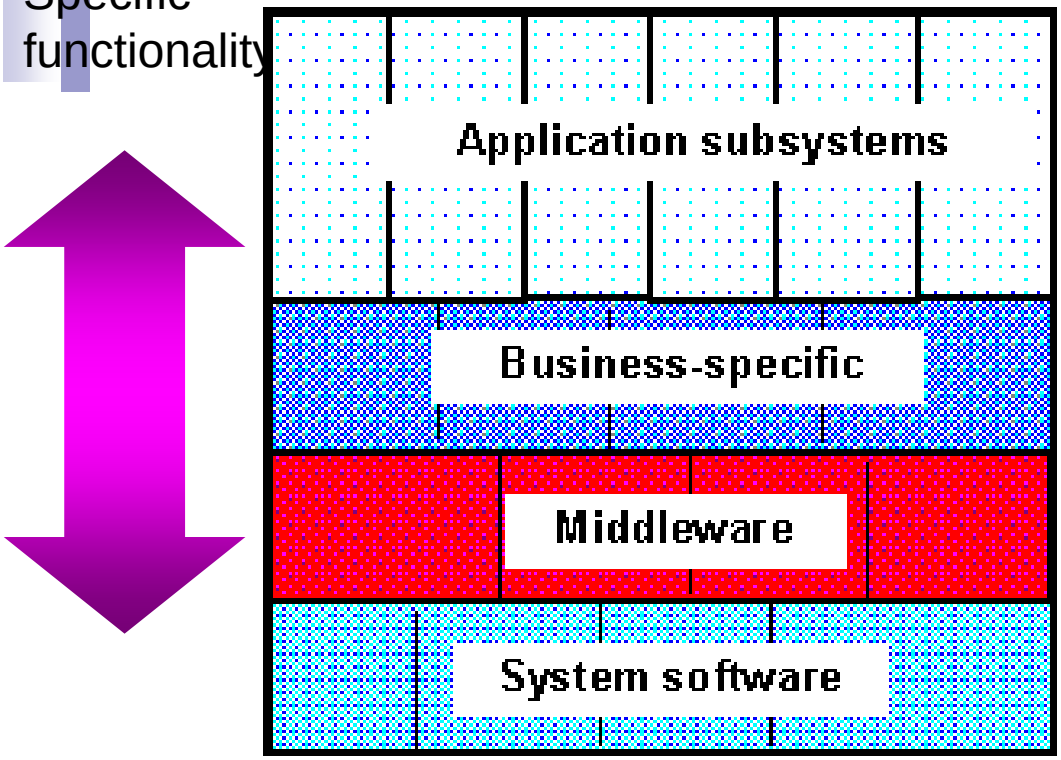
分层是为了隔离各层的关注面

- 分层的标准是:
  - 将相似的事物分组在一起
  - 将不同的事物分开
- 分层的目标在于隔离关注面:
  - 不同层的元素与不同领域相关，例如，业务层关注业务领域问题、中间件层关注适配操作系统的底层差异等软件自身问题。
  - 各层的元素属于同一抽象级别，而不同抽象级别的特性不同，要求的处理方式也不一样。
  - 稳定与变化的隔离。

顶层结构：典型的层次化途径

特殊的功能
Specific functionality

受应用的功能性需求影响更为显著



Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business specific - contains a number of reusable sybsystems specific to the type of business.

Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

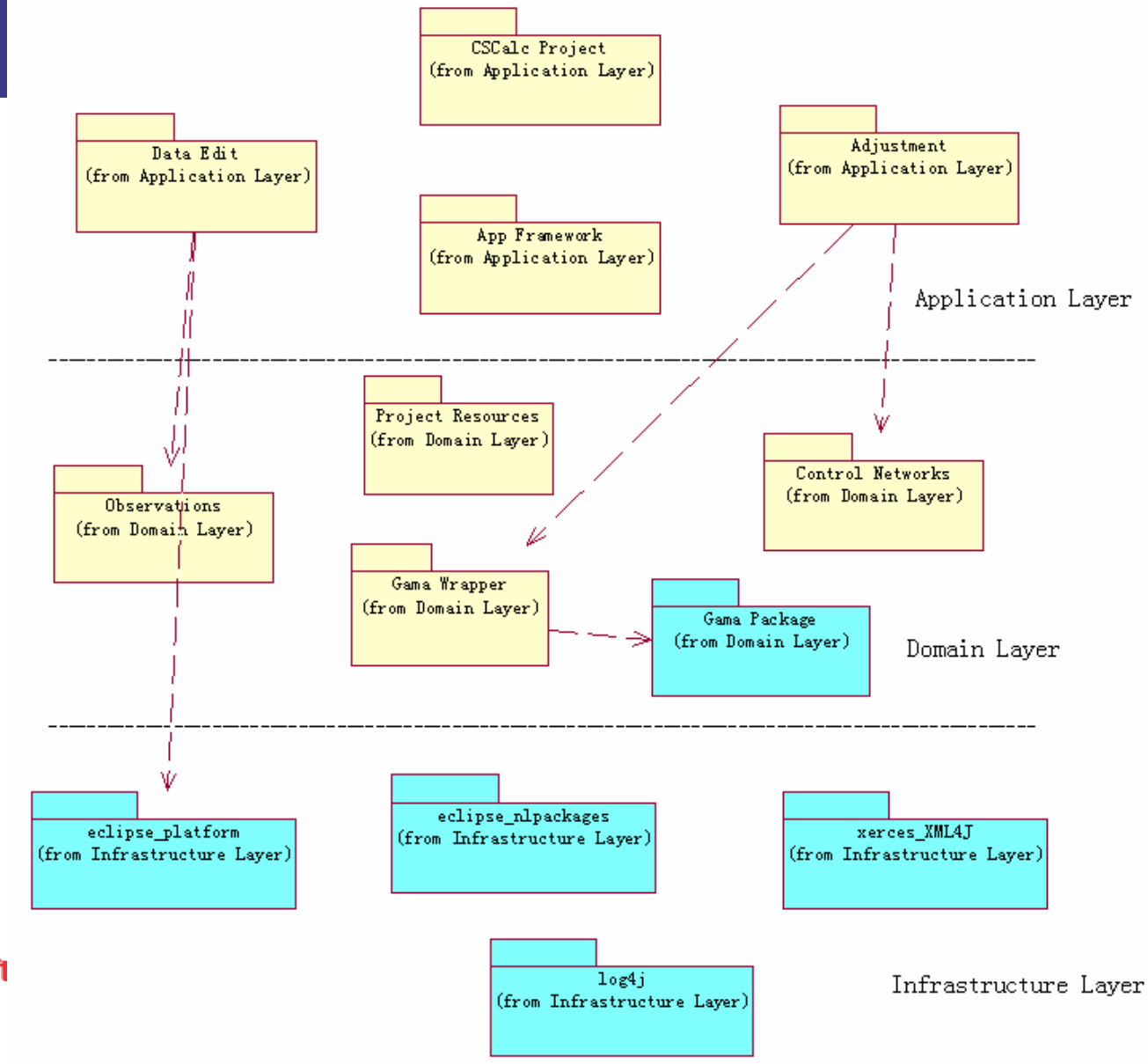
通用的功能
General functionality

受性能、部署等非功能需求，以及实施的平台环境的约束的影响较为显著

分层的不同类型

- 精化Refinement - （例如RM-ODP参考模型）
  - –各层表达的内容相同，只是抽象级别（关注面）不同
  - –底层较之上层增添了更为具体的内容，是对上层的进一步细化（精化）
- 实现Realization - （例如OSI - 7层通讯模型）
  - –各层包含了不同的内容，整个系统缺少任何一层的元素都不再完整
  - –底层较之上层更加技术化，距离最终实现（硬件）更为接近，并为上层提供服务
- 部署Deployment - （例如J2EE开发部署模型）
  - –与“实现Realization类型”基本等价

实例：CSVCalc系统的初始构架层



指南：层次模式的优缺点

- 分层结构的优点：
  - 上层可以直接使用下层，而不需要去了解下层的实现细节
  - 上层对下层透明，因而可以改变底层的实现，而不影响上层的应用
  - 可以减少不同层之间的依赖
  - 容易制定出层标准
  - 下层可以用来建立提供给上层的多项服务
- 分层结构的缺点：
  - 层不可能封装所有的功能，一旦有功能变动，可能会波及所有的层（例如领域对象增加一个属性）
  - 穿过层次的调用，将引起效率降低；对于水平分布的层级Tier划分，效率降低则极为明显

指南：划分应用层子系统

- 考察的因素
  - 用户的分类
  - 用户界面的粒度
  - 用户界面交互逻辑及其关联的复杂度
  - 用户界面的部署约束
- 划分方式
  - 围绕系统的用户执行者Actor，来划分应用子系统（物理上隔离的可执行体）
  - 使用工作视图模式，通过统一的主界面来访问应用子系统（逻辑上的）

指南：划分实体层设计包

- 考察的因素
  - 业务领域
  - 性能需求（实体的数量、粒度、并发访问数等）
  - 可伸缩性
- 划分方式
  - 按照业务领域来划分实体包
  - 考虑实体的数量级和存取频度等性能需求，围绕缓存等优化途径来划分
  - 考虑实体的粒度，将大量细粒度的实体从属于某个主实体，从而组成实体包（例如EJB中的本地接口EntityBean和远程接口EntityBean ）

隔离关注面的主要示例
——架构机制的抽取

软件架构不仅仅只是顶层结构

- 人们通常对架构的理解是指（软件）系统进行分解的顶层结构，包括其组成元素，元素之间、元素与外部的关系。
- 然而，结构只是系统的一个方面，系统中还存在诸如对象持久化等数量众多的各类问题；这些问题在系统中重复出现，我们应当给这些关键的重复问题提供通用的解决方案。这些方案关注构架的动态方面，已经包含了微观细节，例如具体的代码。

构架机制

- 所谓构架机制，就是一种模式、机制，它表达了对一类频繁遇到的问题的一种共通的解决方案；
  - 它们可能是关于结构、行为、或同时两者的模式；
  - 它们是在系统要求的功能，和在给定的实施环境限制下，如何实现这些功能之间的黏合剂；

示意：什么是构架机制

要求的功能

Required Functionality

实施环境

Implementation Environment

Supplementary Specification

“realized by client classes using”

机制

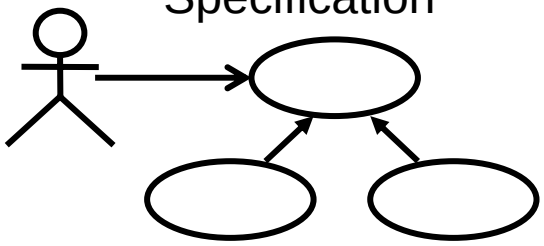
Mechanisms

“constrained by”

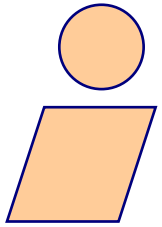
商业产品、数据库、进程间通讯技术等

COTS Products
Databases
IPC Technology
etc.

“responsible for”



Use-Case Model



Architect

构架机制的三个范畴

构架机制的范畴

–分析机制Analysis mechanisms （概念）

以一种与实现无关的方式记录解决方案中关键方面的因素；它们为一个领域相关的类赋予了特定的行为，或明确了一组类之间协作的实施要遵从的条件（设计约束）；它们可以被实现成为一种框架

–设计机制Design mechanisms （具体）

比分析机制更具体，它们呈现了实施环境的一些细节，但不与一种特定的实现绑死

–实施机制Implementation mechanisms （实际）

详细说明了机制的实际实现；实施机制与一种确定的技术、实现语言和厂家绑定

指南：分析机制

分析机制代表了一种为通用问题构建通用解决方案的模式（但不包含方案本身的实施细节）：

- 这种模式分别涵盖通用的抽象行为、结构、或它们两者
- 机制将在分析过程中用来降低分析模型的复杂度，并通过为设计师提供一种表述复杂行为的通用词汇，来提高分析和设计成果的一致性
- 机制的引入，使得分析活动聚焦于如何将功能需求转换为软件概念（分析类及其协作），而不陷入那些非核心的、但需要用来支持功能实现的相关复杂行为的细节
- 分析机制表达了一种占位符，它代表了构架中底层中的某种复杂技术（例如对象持久化），实际上将充当从分析演进到设计的一座桥梁

构架分析时所面对的通用问题

- 开发健壮的构架（特别是分布式）通常要解决以下问题
  - 选择候选层级Tier
  - 如何保持会话状态
  - 确定共通的用户界面交互机制
  - 确定共通的数据存取机制（OR-Mapping）
  - 解决并发和同步冲突
  - 支持事务处理
  - 接口的定位与实例化机制（常用途径：名字服务）
  - 设计统一的异常机制
  - 安全机制的实施
- 上述问题中的大部分将抽象为分析机制

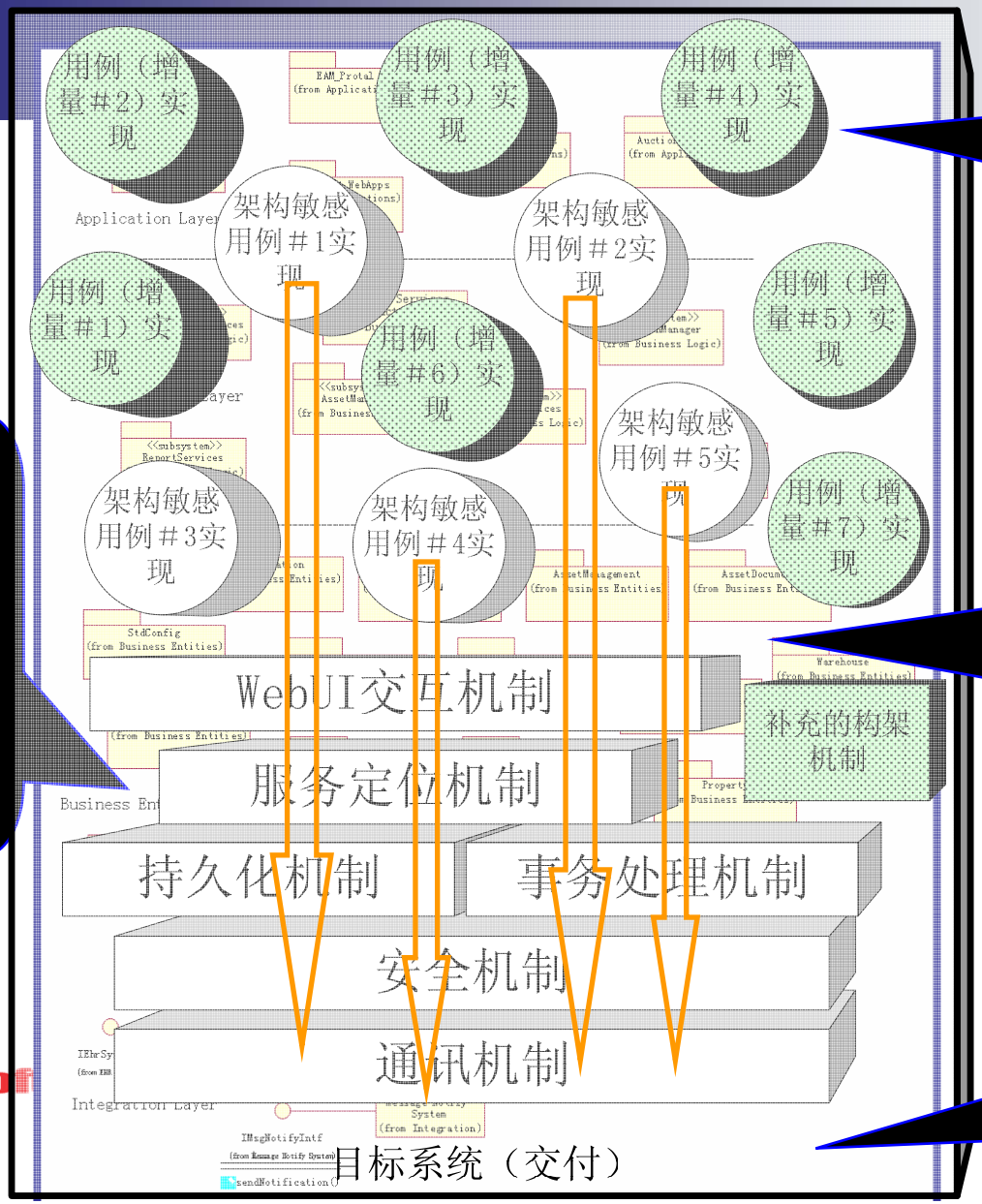
通用问题被总结为分析机制列表

- GUI交互
- 遗留接口Legacy Interface
- 持久化Persistency
- 通讯Communication (IPC and RPC)
- 消息路由Message routing
- 分布式计算Distribution
- 事务管理Transaction management
- 进程控制与同步 （资源争用）
- 信息交换、格式转换
- 安全Security
- 错误侦察/处理/报告
- 冗余备份Redundancy

隔离类功能与通用问题实现细节

- 几乎所有的类都可能遇到诸如对象持久化、权限控制、事务处理等类似的通用问题，显然同时思考每个类所特有的功能，和这些共通的问题，将增加开发的难度。
- 将分析与设计分开，能够简化问题。

架构基线的纵向与横向分割原理



目标系统（交付）

目标系统迭代交付：实现了迭代计划中的用例

构架基线原型（交付）：蕴含了“构架主体”、实现了构架敏感用例的最小可运行交付

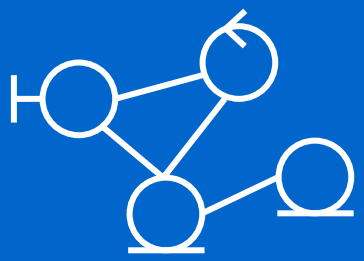
目标系统最终交付：实现了所有的用例

注意：
上面那些概念元素最终要映射到底下正交的构架逻辑层次划分之中

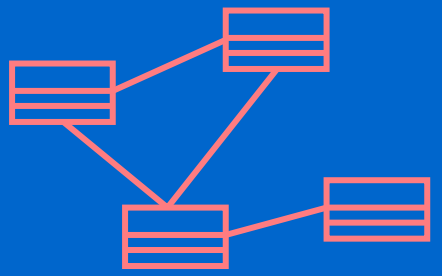
从分析到设计/实施

理想化的概念设计
（仅仅满足功能需求）

具体的设计
（同时满足非功能需求）



Analysis Model



Design Model

修饰分析机制特征
（设计约束）

贯彻设计机制
（面向实施）

分析机制

设计与实施机制

示例：实体类分析机制的特征

- 持久化Persistency
  - –粒度Granularity
  - –数量Volume
  - –持续时间Duration
  - –存取机制Access mechanism
  - –访问频度Access frequency (creation/deletion, update, read)
  - –可靠度Reliability

构架行为：修饰分析机制

在进入设计之前，需要明确分析类如何贯彻构架分析中所确定的分析机制（支持性的复杂行为）；其实质是明确了分析类的设计约束，此后通过分析机制向设计机制的映射，可以帮助我们将分析类顺利地演化成设计类。

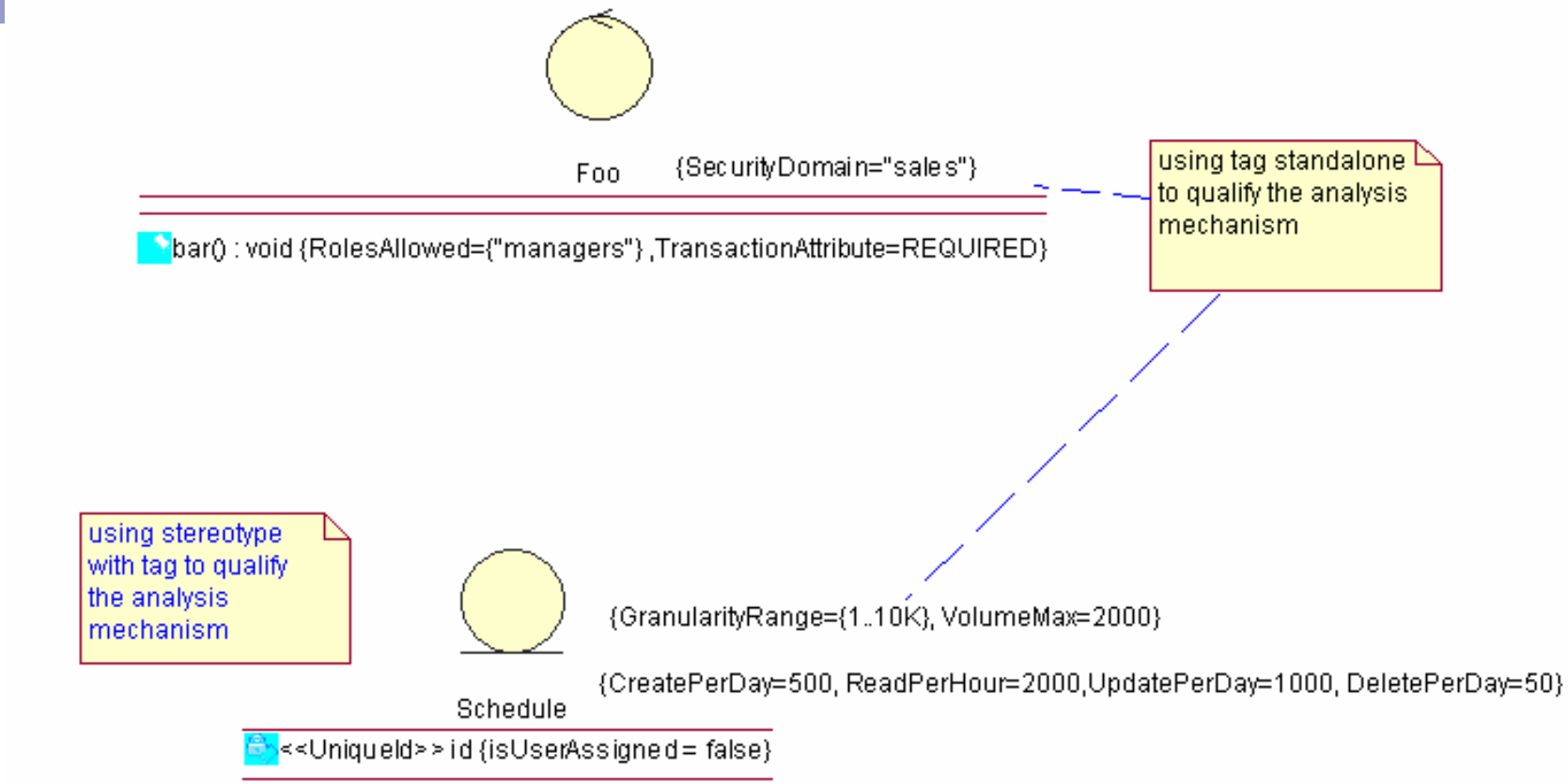
- 说明qualify（修饰）分析机制的目的:
  - 识别分析类所适用的分析机制;
  - 提供分析类如何应用分析机制的附加信息;
- 修饰分析机制:

对于每个这样的机制，我们需要尽可能地修饰更多的特性，而当还存在很多不确定性时，给出合适的范围

实例：修饰分析机制特征

- 日程表的持久化分析机制特征characteristics明细(Persistency for Schedule class):
  - 粒度Granularity: 1 to 10 Kbytes per product
  - 数量Volume: up to 2,000 schedules
  - 访问频度Access frequency
    - Create: 500 per day
    - Read: 2,000 access per hour
    - Update: 1,000 per day
    - Delete: 50 per day
  - Etc.

实例：利用UML扩展机制来修饰分析机制



前瞻：利用注释来修饰分析机制

- 利用EJB 3.0 annotation机制可以在代码级修饰分析机制（这实际上已经对应到实施机制层面了）

```
@SecurityDomain(“sales”)  //安全机制
public class Foo {

   @RolesAllowed({"managers"})  //安全机制
   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   //事务处理机制
   public bar () {
     // do something ...
   }
}
```

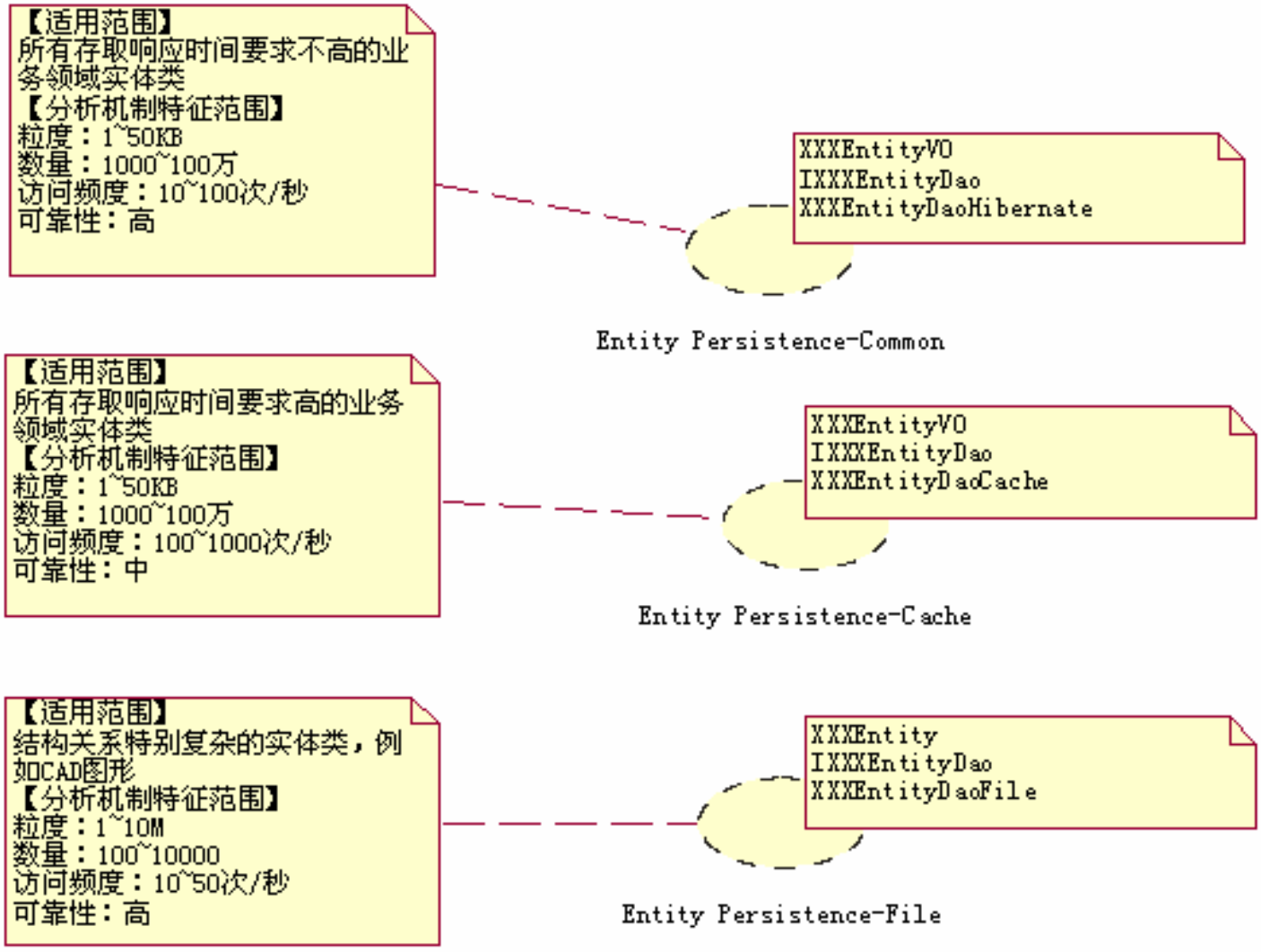
归纳应用分析机制的各类要求

- 分析机制实质上是为那些适用的分析类提供了概念上的一组服务（从而支持其最终完成功能需求）
- 其代表的复杂支持性行为，在分析活动中常被忽略，但最终在设计时必须被考虑进来
- 不同的分析类对分析机制的要求并不一定相同，这从修饰的特征上就可以看出，例如一个万数量级的实体类，和一个千万数量级的比较，其对持久化机制的性能要求肯定是完全不同的
- 将对分析机制的各种需求进行归纳，并依据这些划分，对那些将要应用机制的客户类（分析类）进行分组
- 最终将要开发不同的设计机制来满足不同的分析机制需求

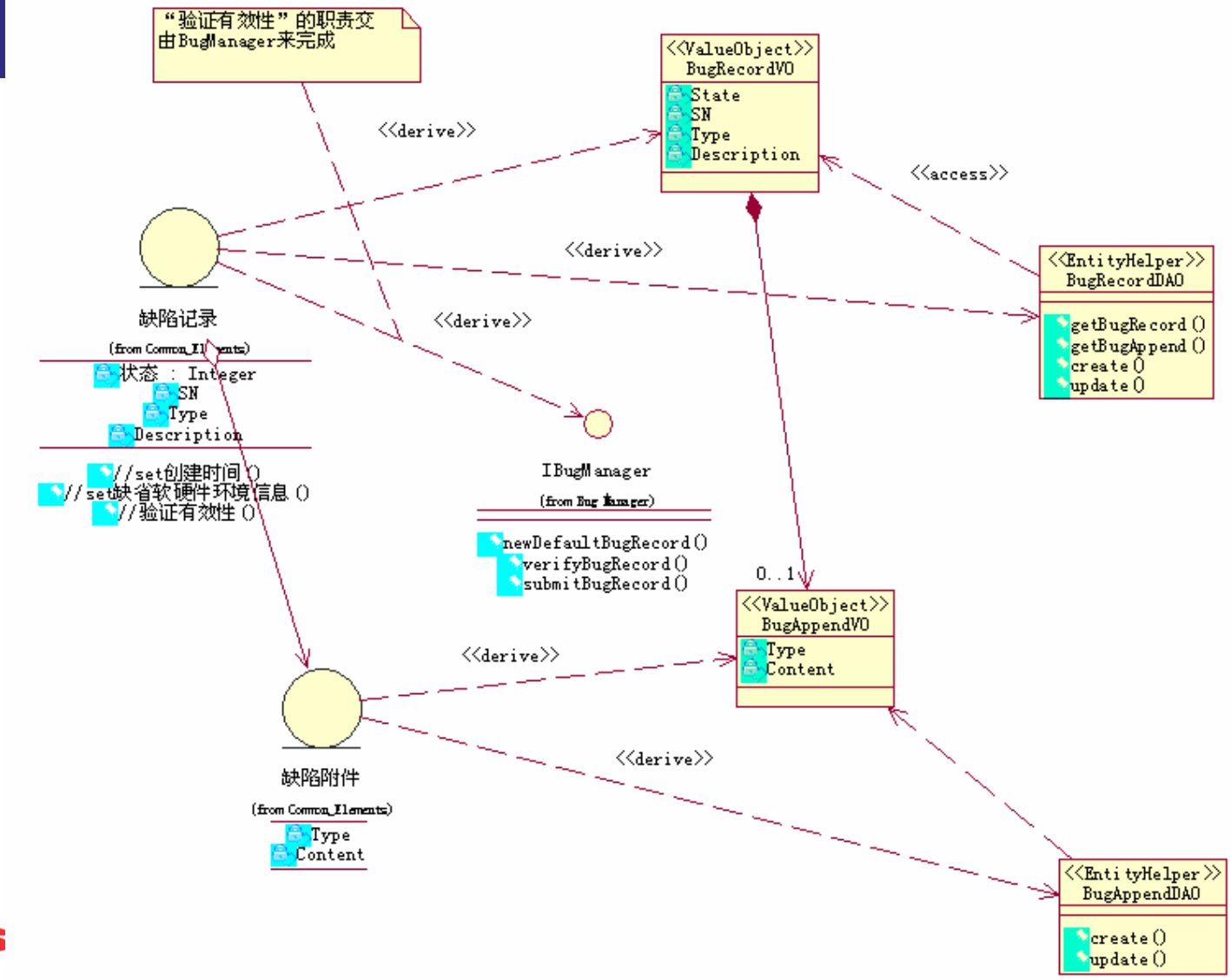
设计机制

- 分析机制只是代表了某种通用解决方案的行为，而设计机制则为这些纯粹的概念增添了必要的实现细节（但忽略特定的实施技术）
  - 修饰分析类的分析机制特征规定了对应的设计约束，设计机制必须满足这些约束，并对分析机制进行细化
  - 设计机制通常被总结成为一种设计模式或框架
  - 设计机制的引入，实际上提供了将分析类向设计元素映射的一种通用机制
  - 设计机制规格化了构架中底层中的某种复杂技术（例如对象持久化）

实例：构架机制映射指南


【适用范围】
所有存取响应时间要求不高的业务领域实体类
【分析机制特征范围】
粒度：1~50KB
数量：1000~100万
访问频度：10~100次/秒
可靠性：高
XXXEntityVO
IXXXEntityDao
XXXEntityDaoHibernate
Entity Persistence-Common
【适用范围】
所有存取响应时间要求高的业务领域实体类
【分析机制特征范围】
粒度：1~50KB
数量：1000~100万
访问频度：100~1000次/秒
可靠性：中
XXXEntityVO
IXXXEntityDao
XXXEntityDaoCache
Entity Persistence-Cache
【适用范围】
结构关系特别复杂的实体类，例如CAD图形
【分析机制特征范围】
粒度：1~10M
数量：100~10000
访问频度：10~50次/秒
可靠性：高
XXXEntity
IXXXEntityDao
XXXEntityDaoFile
Entity Persistence-File

实例：利用DAO模式实现持久化



隔离关注面的主要示例
——AOP与分割横切面

AOP支持分析与设计的分离

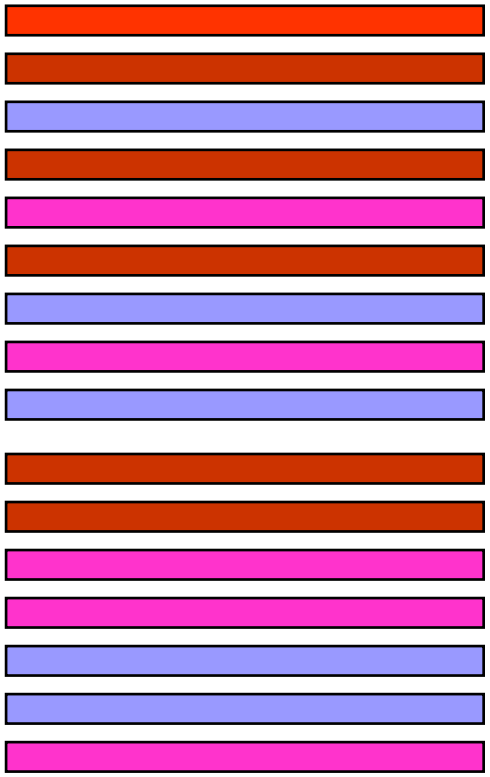
- 从方法论的角度来看，区分分析与设计的终极目的在于简化开发，分析关注目标系统所处理的领域问题（或称业务功能），而设计要关注系统完成上述领域功能时，在性能、部署等方面应当满足的运作需求（非功能需求）
- AOP(Aspect-oriented programing)面向方面的编程，正是一种实施层面的技术，它直接在代码级别上，支持将处理运作需求方面的代码与处理业务逻辑的代码相隔离，使得开发人员将注意力集中于核心的业务逻辑上
- AOP、MDA的模型转换技术等都是支持分析与设计分离方法论的绝佳途径

AOP降低了设计机制风险

- 设计机制的实现事关全局，对其的更改，往往会波及整个系统，造成多处的修改甚至是返工
- AOP技术的出现，使得我们有机会将设计机制的实施本身与系统其它部分相隔离，这时候，对设计机制的修改并不会造成系统的重大波动
- 而相反，目前似乎还没有有效的途径来隔离某个实体类本身的变更影响

示意：AOP将交织的代码分开

ordinary program



better program

Basic classes: Structure

Aspect 1

Slice of Functionality: Business logic

Aspect 2

Slice of Functionality: Operational functions

Aspect 3

Thank you!

China SoftCon 06