Chapter 8

- **Design Concepts**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

什么是软件设计

- 软件设计即根据软件需求，产生一个软件内部逻辑表示的过程。
- 这种更接近于软件的逻辑表达或模型，提供了软件系统关心的体系结构、数据结构、接口和构件等细节。
- 对比：
  - 需求分析偏重于问题域，描述软件要做什么
  - 软件设计则偏重于解决方案，描述软件究竟要如何做。

需求1：教学秘书需要将学生的综合成绩按高到低进行排序

设计1：void OrderScores (struct * scores[ ]) { 冒泡排序算法, step1; step2;… }

需求2：存储 “销售订单”

设计2：关系数据表Order(ID, Date, Customer, …),
OrderItem(ID , No , QUANTITY, ..)

软件设计的价值

- (1)设计是需求与实现代码之间的“桥梁”
  - 设计本质是满足需求，并对方案优中选优的过程。

需求2：存储 “销售订单”

设计2：关系数据表Order(ID, Date, Customer, …)，

OrderItem(ID , No , QUANTITY, ..)

如果要为不同顾客设置不同价格？

  - 设计模型比代码更易评估、更易修改。

- (2)设计是软件质量形成的关键，软件质量依赖于设计的结果。
  - 如软件质量中的效率、可靠性、高并发等。

例如：对含有上百亿信息的数据库，如何满足用户的高并发访问需求？

设计方案：分库分表、读写分离。

软件设计的价值

- “编写一段能工作的代码是一回事，设计能支持某个长久业务的东西则完全是另一回事”

什么是好的软件设计

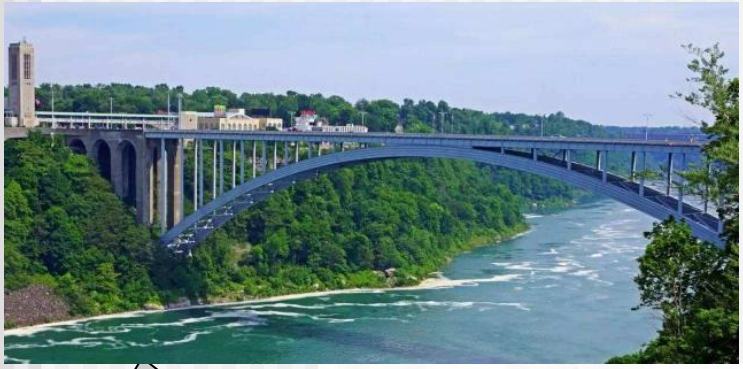
- 设计必须实现分析模型中的明确需求，包括客户期望的隐性需求.　　（承上）
- 对后续的编码、测试以及维护人员，设计必须是可读、可理解的指南。　　（启下）

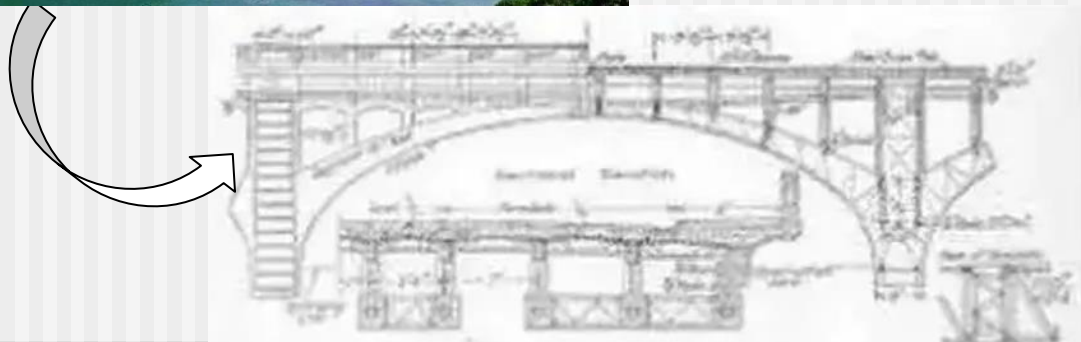
- 设计必须提供未来软件的**全貌**，从实现的角度说明其数据域、功能域和行为域。

  例如，自动排课是系统的功能需求之一，那么设计要解决数据存储问题（排课信息保存）、界面显示问题（排课及结果显示）、算法逻辑问题（排课算法）等。

软件设计如何实施

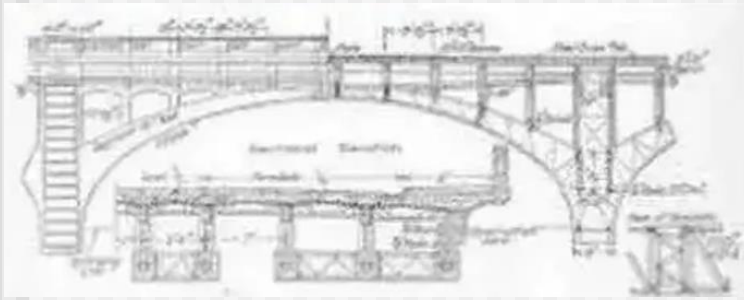
- 软件设计是一个迭代的过程。
- (1) 初始时，设计者应关注系统的整体架构（如架构风格、主要构件、连接关系）等，对应于体系结构设计阶段。

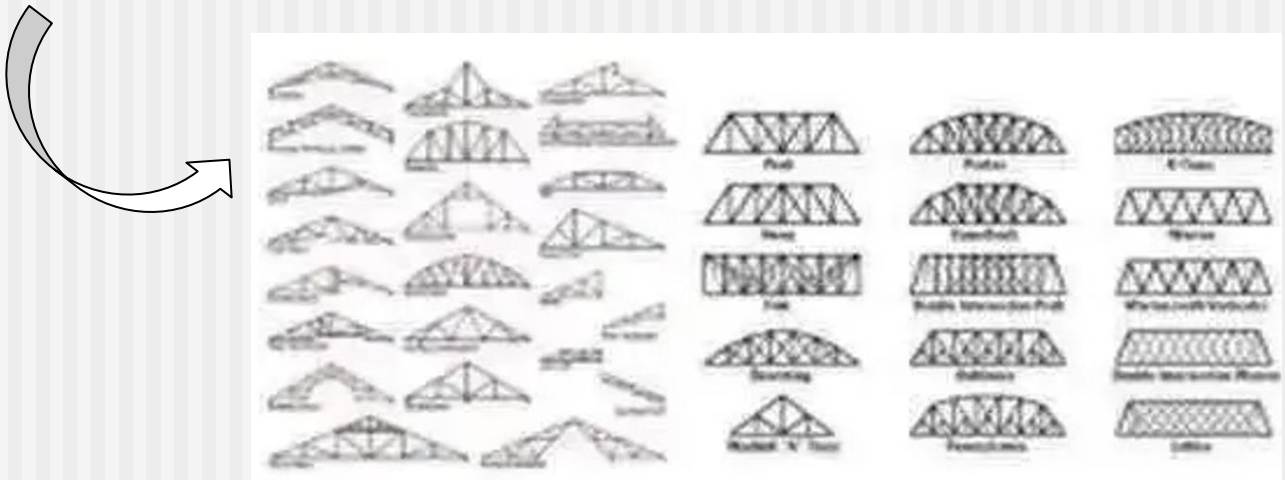




软件设计如何实施

- (2) 随着设计迭代的开始，更多的细节被关注（如数据、构件）等，对应于构件设计或详细设计阶段。





软件设计任务集

(1)体系结构设计：定义软件的主要元素（构件）以及元素之间的联系

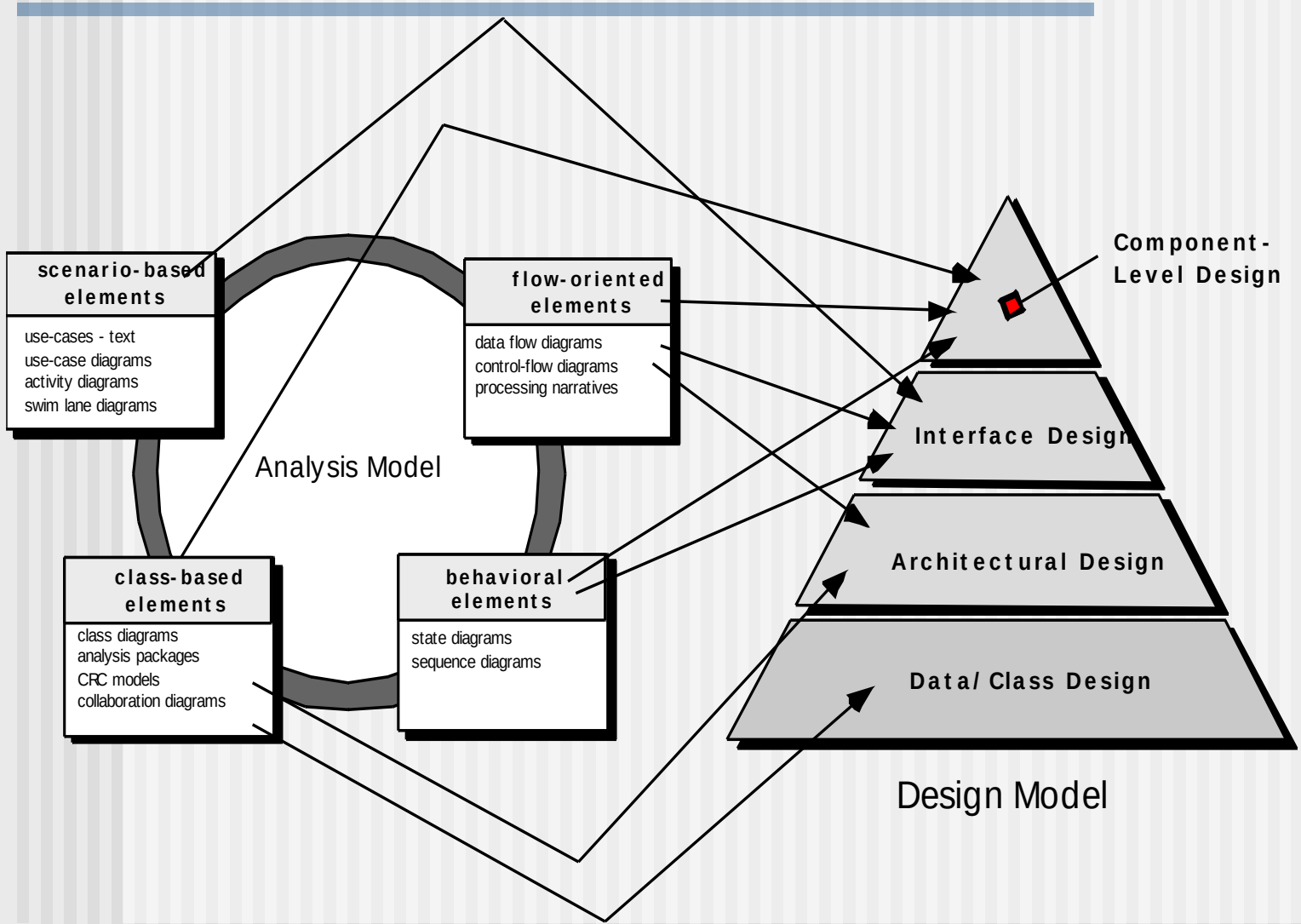
(2)数据/类设计：将分析类模型转化为设计类以及软件所需要的数据结构

(3)接口设计：定义软件与协作系统之间、软件与用户之间的通信

(4)构件设计或详细设计：定义软件元素（构件）的内部细节，如内部数据结构、算法等

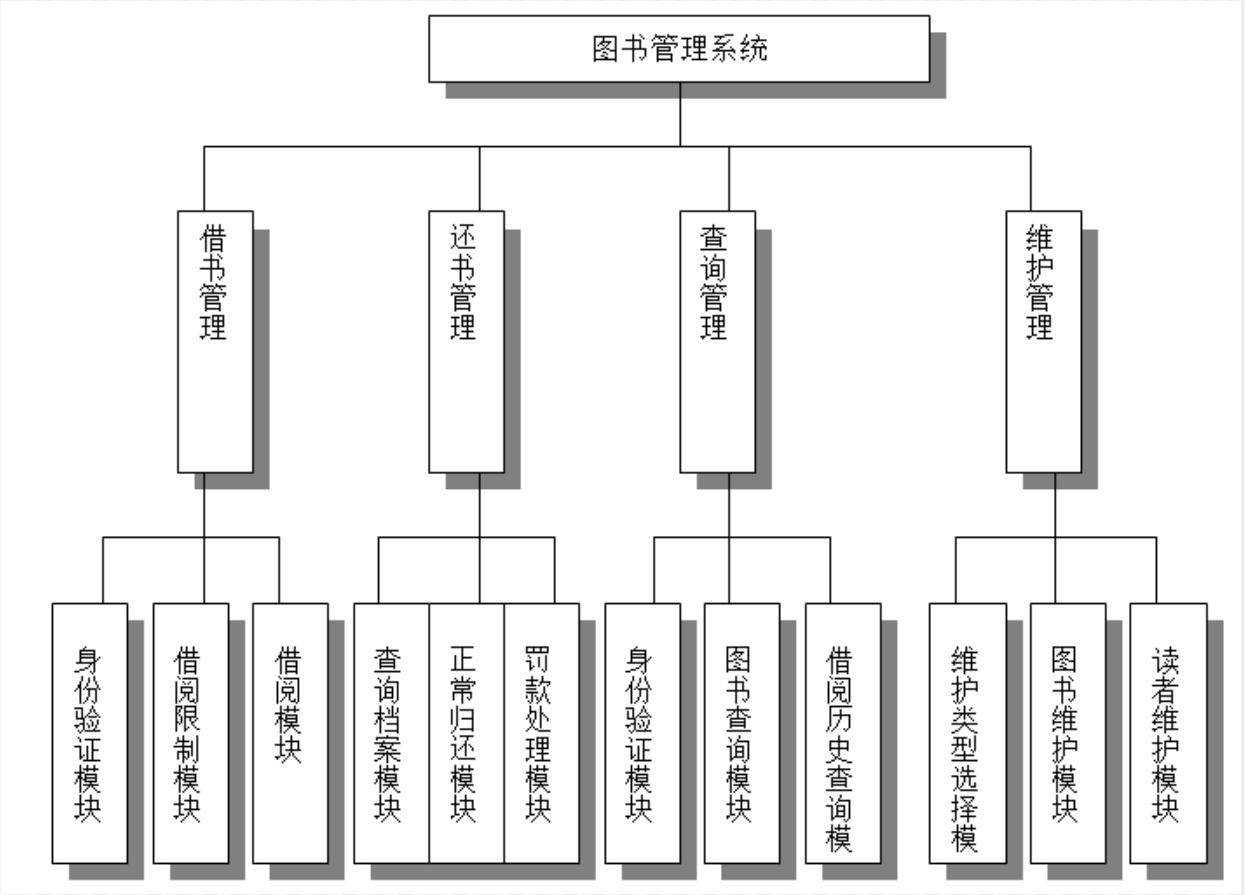
(5)部署设计：定义软件元素在物理拓扑结构中的分布

软件设计模型


scenario-based elements
use-cases - text
use-case diagrams
activity diagrams
swim lane diagrams
flow-oriented elements
data flow diagrams
control-flow diagrams
processing narratives
Analysis Model
class-based elements
class diagrams
analysis packages
CRC models
collaboration diagrams
behavioral elements
state diagrams
sequence diagrams
Component-Level Design
Interface Design
Architectural Design
Data/ Class Design
Design Model

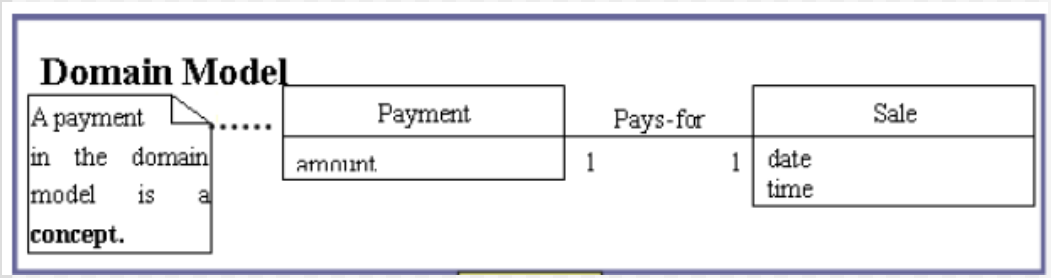
软件设计模型

- (1) 体系结构设计



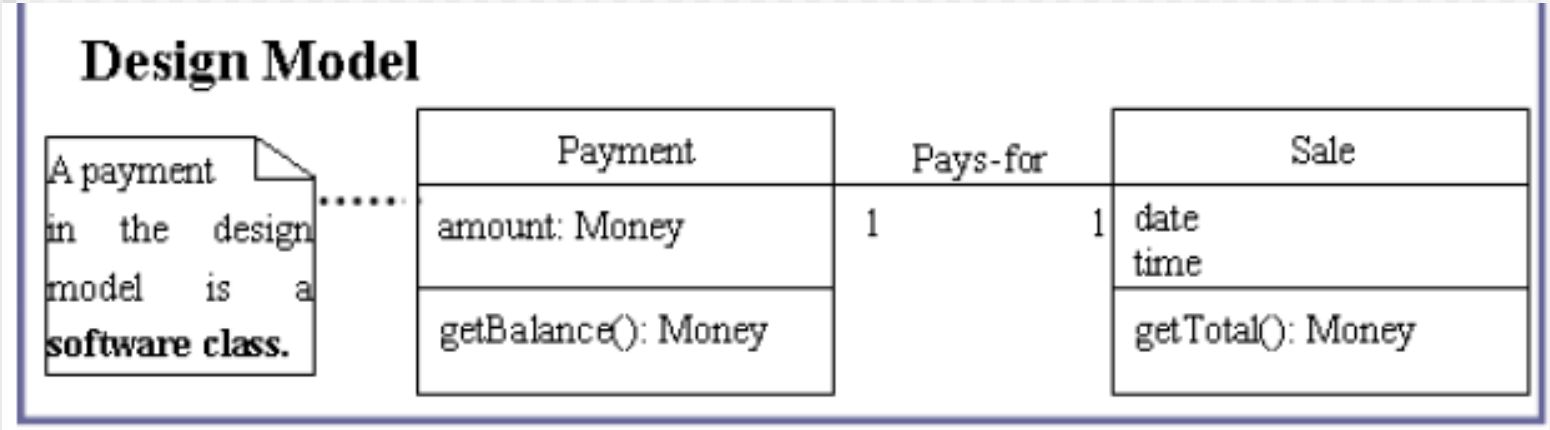
软件设计模型

- (2) 设计类



分析类

设计精化



软件设计模型

- (3) 接口设计

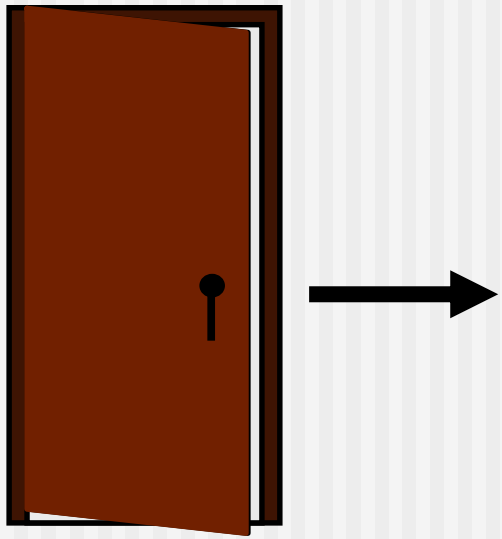
应该遵循什么原则才能保证设计质量？

Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- Design Classes—provide design detail that will enable analysis classes to be implemented

数据抽象

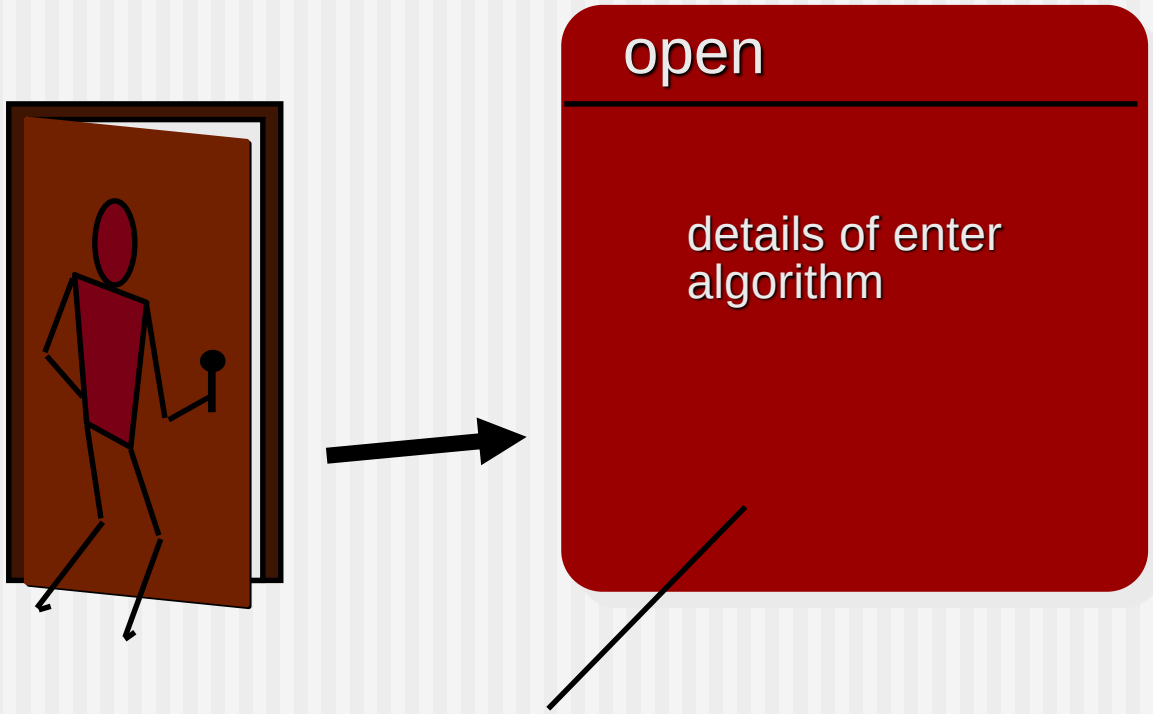
- 抽象的本质：强调关键特征，忽略无关细节
  如：对打车软件中的“车辆”，你会如何抽象？



door

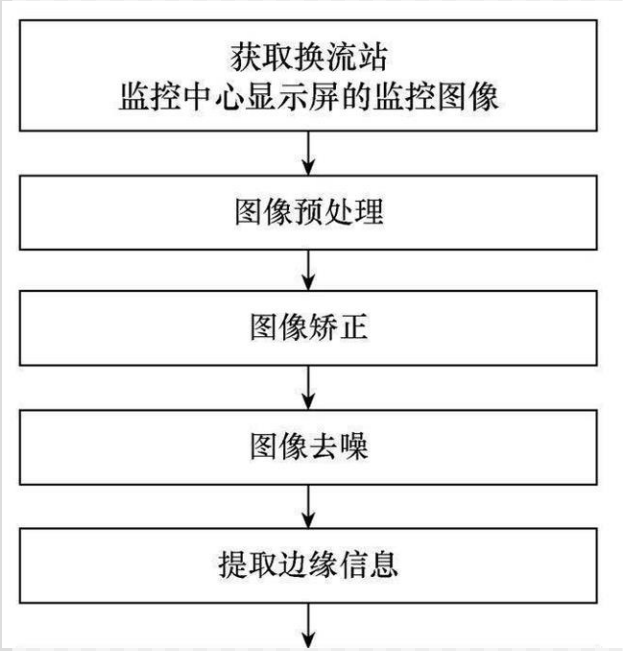
manufacturer
model number
type
swing direction
inserts
lights
  type
  number
weight
opening mechanism

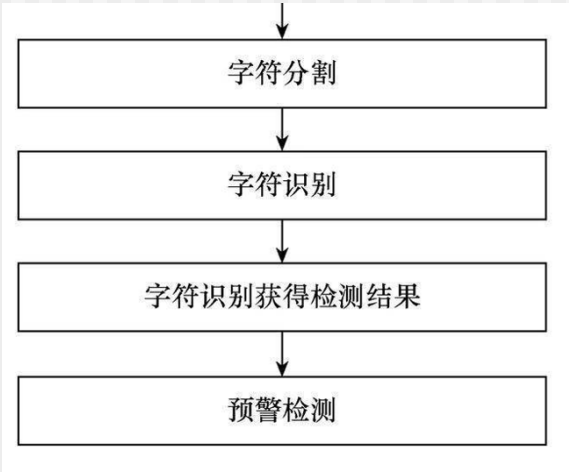
过程抽象


open
details of enter
algorithm
implemented with a "knowledge" of the
object that is associated with enter

体系结构优先

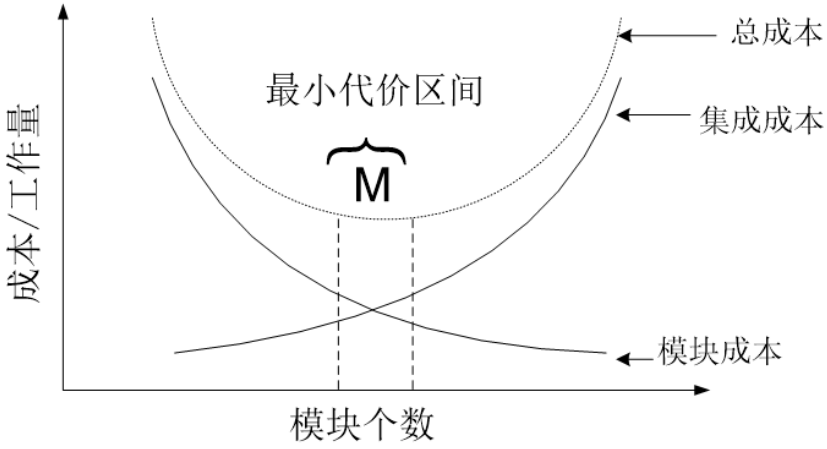
- 软件体系结构设计，即找到一种最佳的方式对系统进行划分，标识其中的构件，构件之间的联系等。
- 设计应该先勾勒全貌，不能一开始就跳入细节。
- 例如，在设计初期，为达到异常监测的目的，我们应该关注“系统的关键构件构成”，而非“图像去噪”如何实现。



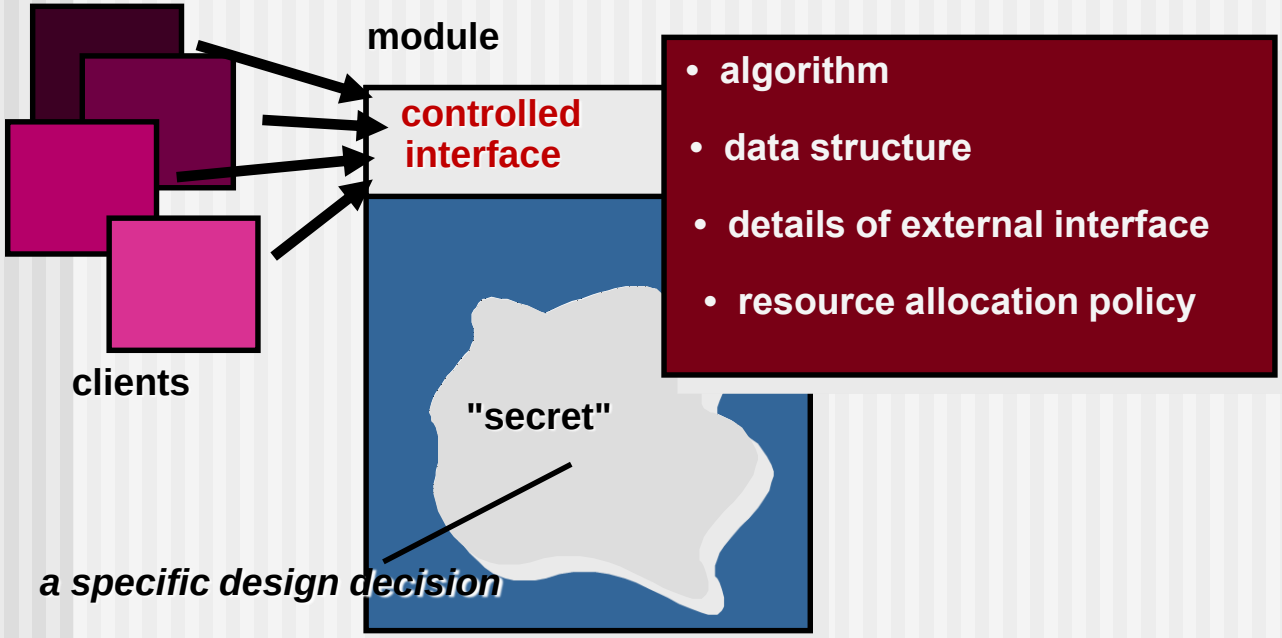


模块化

- 庞大的系统不便于并行开发，软件工程师也难以理解和把握。
- 采用分治策略，将大系统分解为若干更小、更简单的模块。
- 这样，问题理解变得更加容易，开发成本也会减小。

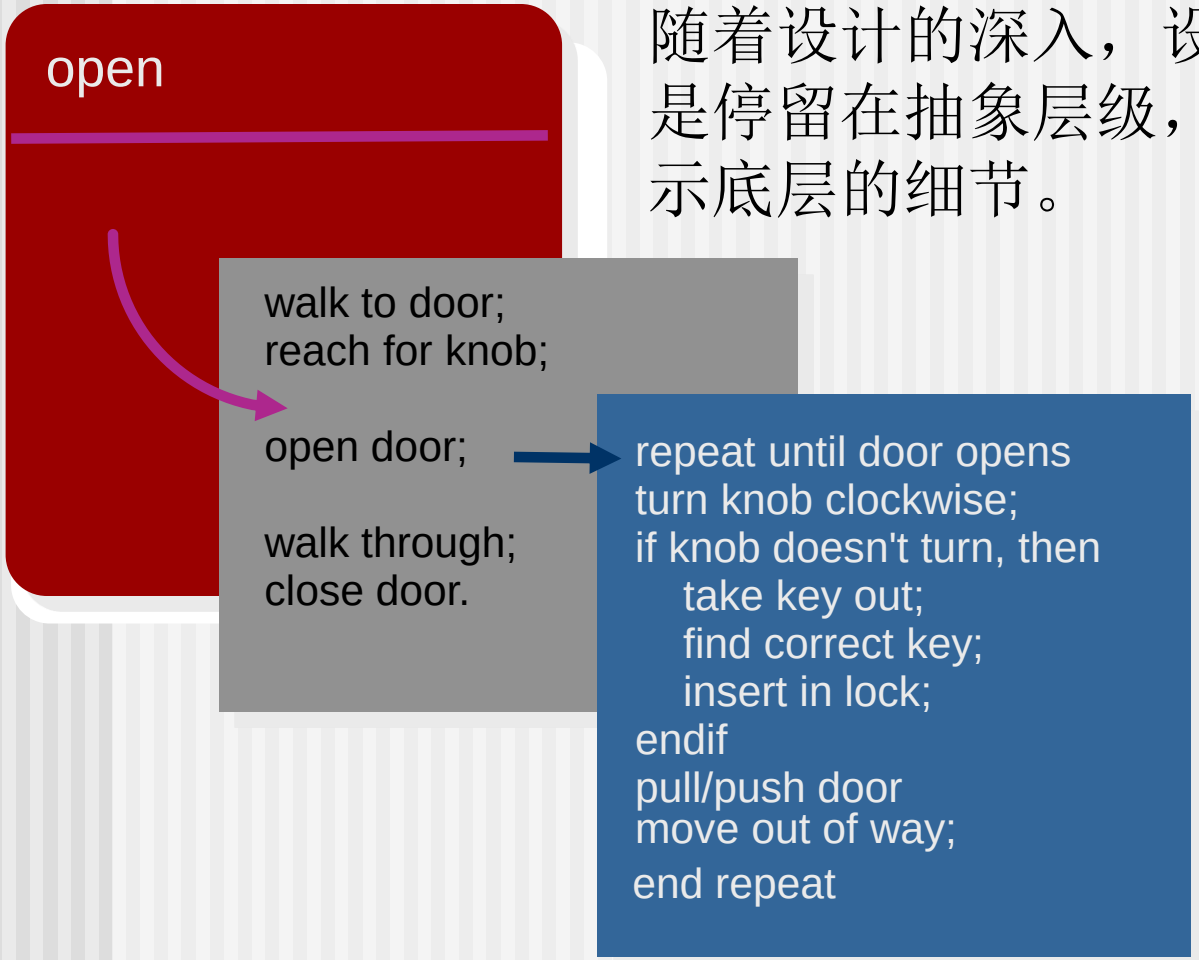


信息隐藏



- 每个模块都对其它模块隐藏自己的设计细节（就像黑盒子），这样如果模块内部发生变化，外部受到的影响最小。

逐步求精



随着设计的深入，设计者不能老是停留在抽象层级，而应逐步揭示底层的细节。

功能独立

- 功能独立性是指：模块化设计时，希望每个模块尽可能功能专一，同时避免与其它模块过多交互。
- Cohesion(内聚)： 表示模块内相关功能的强度
  - 内聚性强的模块通常只完成一件事情。
- Coupling (耦合)：表示模块之间的相互依赖性
  - 模块的耦合性取决于模块之间接口的复杂性、引用方式、传递数据的复杂性等。
- 功能独立意味着：高内聚、低耦合

功能独立

- 功能独立的优势
  - ✓ 更容易开发。因为每个功能均有效分隔，且接口简单。
  - ✓ 更容易维护。因为修改代码引起的副作用被限制，减少错误扩散。
  - ✓ 复用性更好。

重构

- 在不改变代码的外部行为的前提下，优化内部结构的过程。

- 例如：早期设计中发现某构件的内聚性较差，在保持接口不变的情况下，可通过重构将其分解为两个独立的构件，每个新的构件内聚性得以增强。

设计类

- 分析类只与业务领域有关，和具体实现技术无关。
- 设计类则面向实现，通过对分析类的细化设计，达到能提供给构造阶段编码实现的目的。

- 定义设计类的两种方式：

（1）通过对分析类提供细节设计。

（2）创建新的设计类，该设计类提供软件的基础设施以支持业务解决方案。

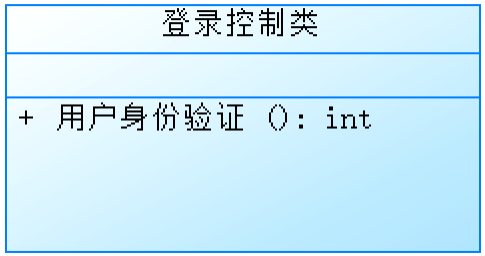
设计类

(1) 通过对分析类提供细节设计

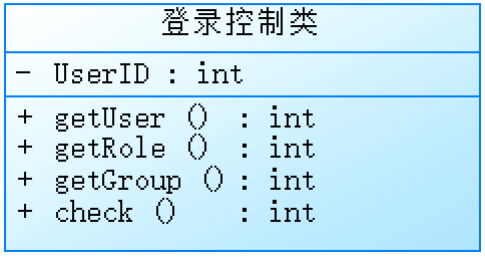
主要包括提供对类的成员属性、方法、关系等的细节设计。如：

分析类“登录控制类”中有“用户身份验证”职责。

设计类“登录控制类”中，该职责将进一步被细化为多个成员方法：



分析类



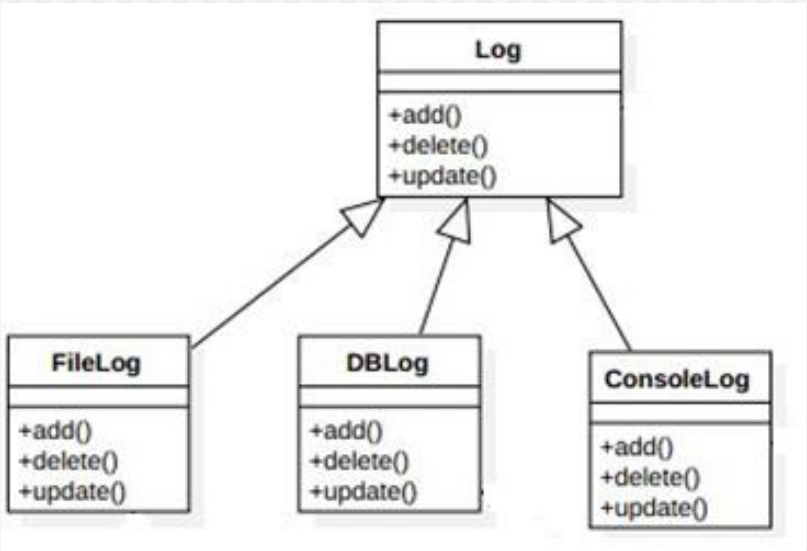
设计类

设计类

(2) 创建新的设计类

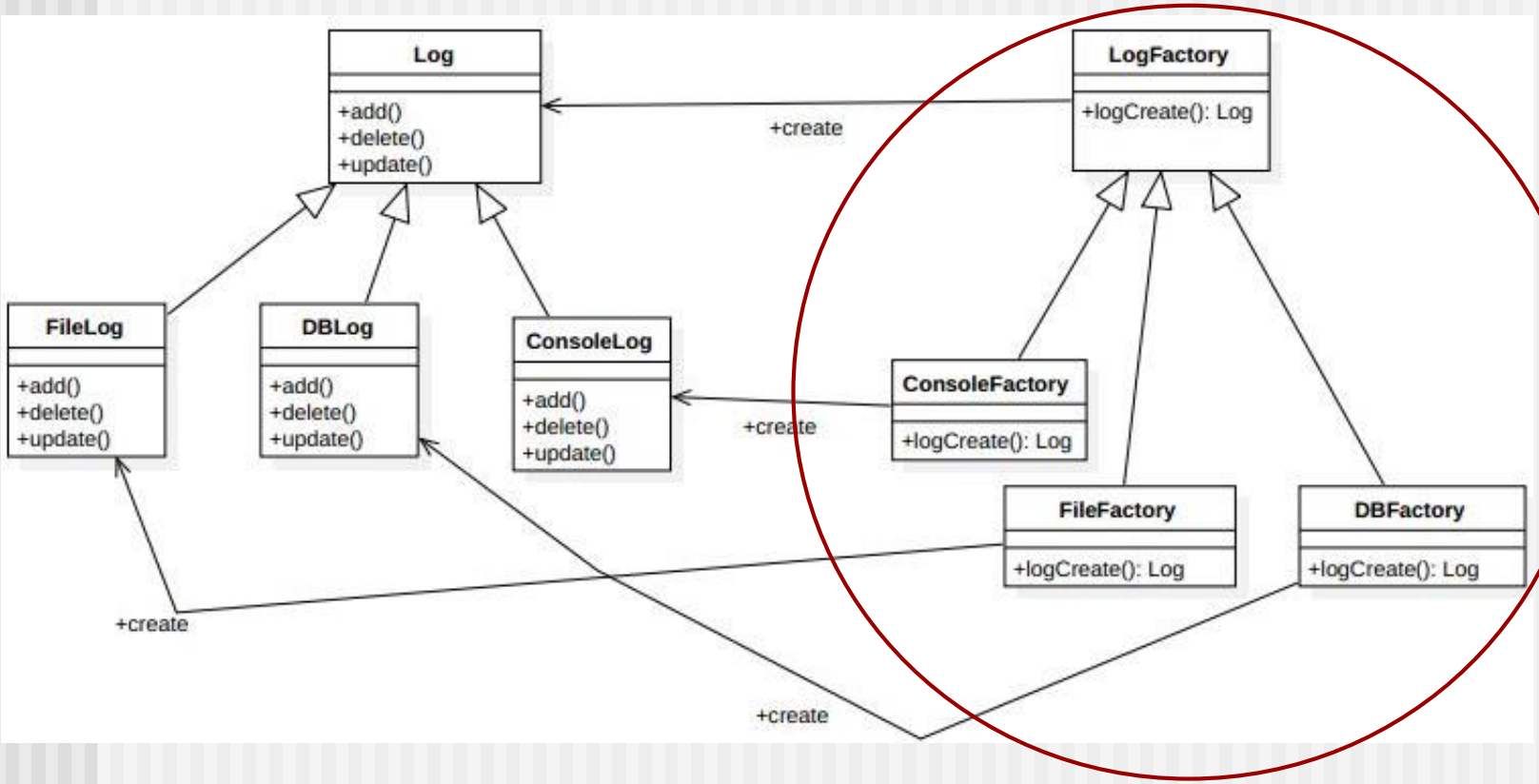
如：某日志系统包括多种日志（分析）类，且传统生成对象方式为FileLog f=new FileLog()

问题：一旦更换了日志类，每个调用的地方都要修改代码。



设计类

改进：基于工厂模式，增加新的设计类LogFactory等




```
Log  log1= LogFactory.logCreate(FileLog)
```

生成对象可通过参数灵活设置