软件设计原则

软件设计

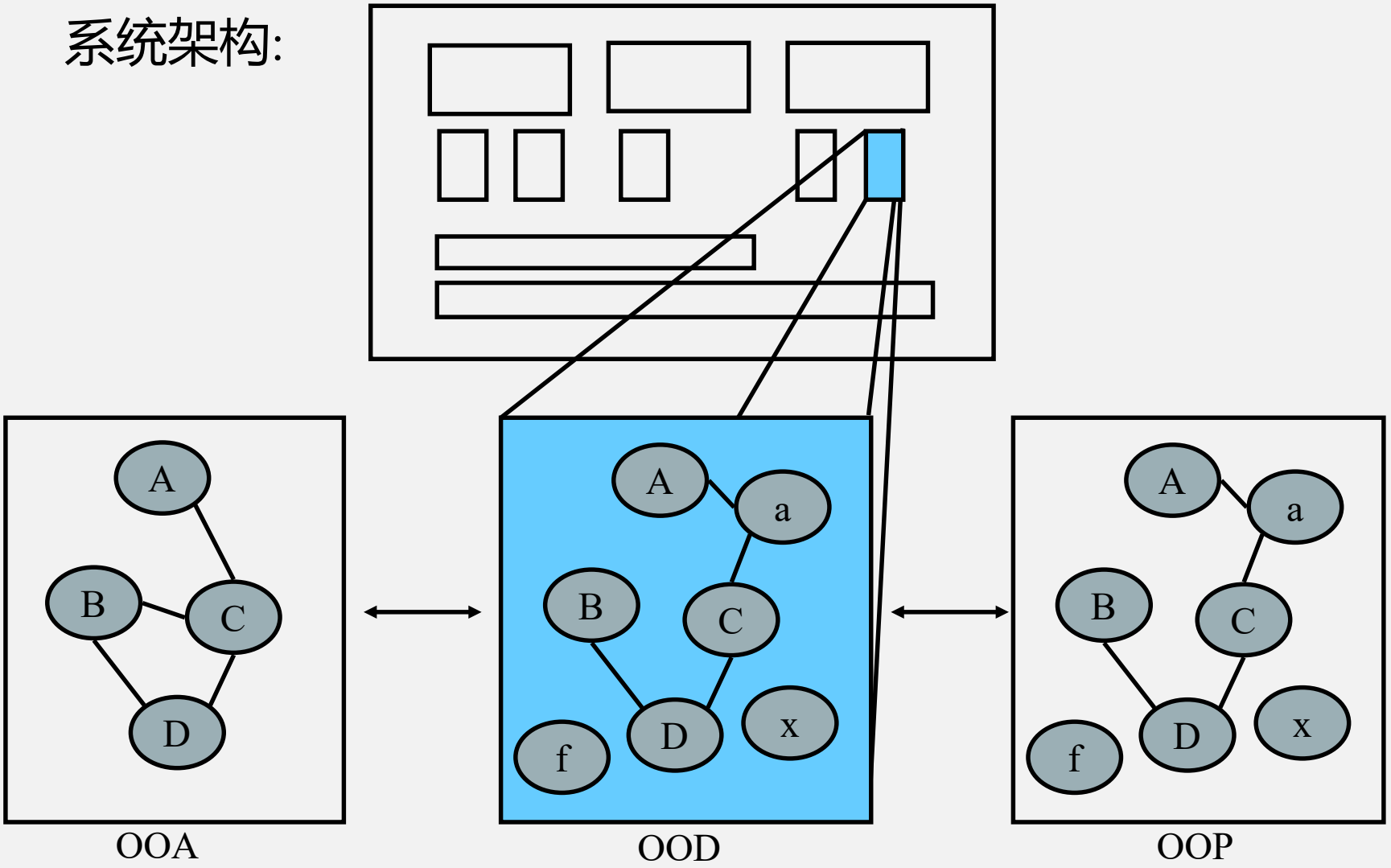
- “需求”定义了
  - 系统需要满足的目标
- “规约”定义了
  - 系统的外部可观察到的行为
- “架构”定义了
  - 系统一级的主要组成部分
  - 各部分的交互方法
  - 使用的技术
- “设计”定义了
  - 如何完成任务
  - 需要写的代码
  - 我们将专门关注OO设计

面向对象软件设计

- 将实现的约束条件应用到面向对象分析（OOA）所产生的概念模型的过程
- 用方法和属性来描述用于构成系统的类
- 添加不明显属于领域的类，比如抽象类和接口
- 描述类是如何构成组件的

OOD 所处环节

系统架构:

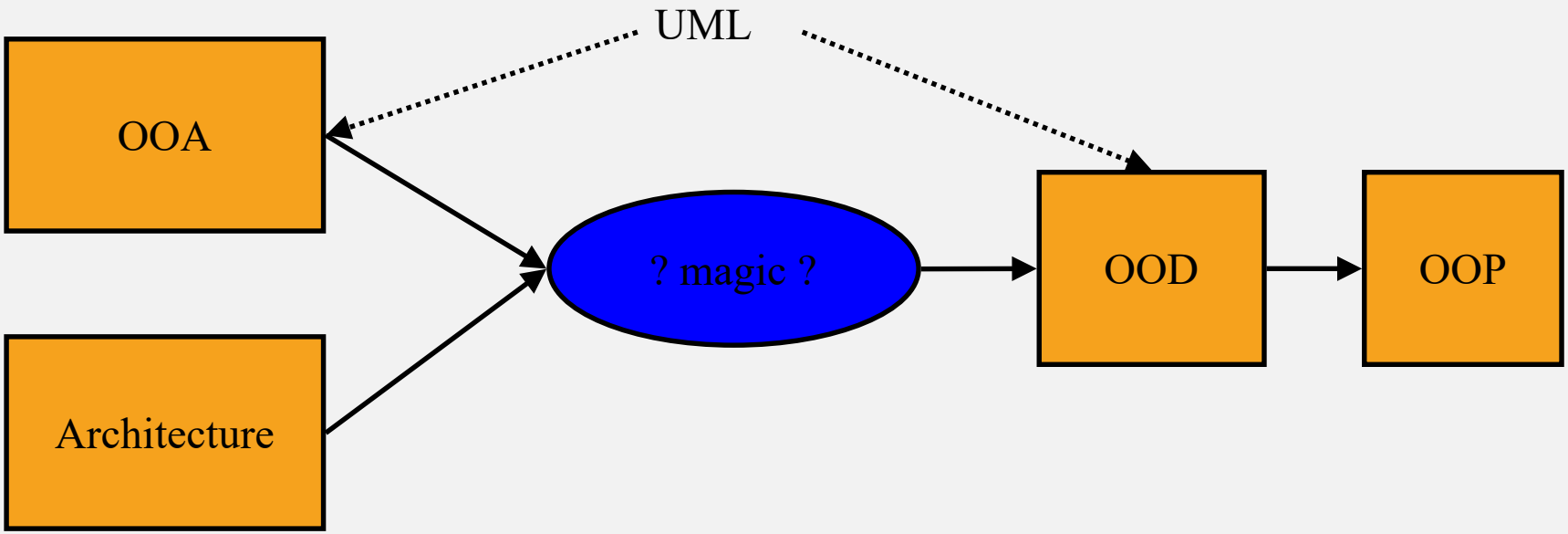


如何发现合适的对象

- OOD的难点在于将一个系统分解成对象
- 许多对象直接来自于
- 分析模型
- 　　或从
- 实现空间（数据库、文件、用户界面、IPC...）
- 同样，还有其他一些类没有这样的对应类
  - 用来使可能过于特殊的设计变得更为通用
    - 例如，使用“策略”模式
      - 如果你认为一个算法很可能会改变
    - 添加类来实现 "策略 "模式

经验至关重要

- 从OOA到OOD，没有循序渐进的简单方法
  - 至少OOA以相当直接的方式给出了问题域组件
  - 对于其他的部分，则需要经验。



经验如何而来

- 经验丰富的设计师看到同样的老问题一次又一次的出现
  - 如何设计我的第五个用户界面的类?
  - 如何第三次设计支持持久化到数据库的类?
  - 如何组织班级进行第五次汇报工作?
  - …
- 每次遇到类似的问题，设计师一般都会从之前行之有效的东西入手
  - 但通常会体会到上一次他们可以做得更好
  - 技术在我们脚下不断变化，所以我们的设计经验很快就被淘汰了
  - 但设计原则以及一些经典的设计经验不会褪色……

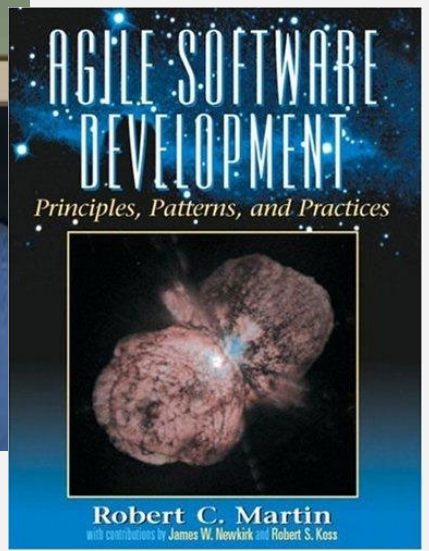
本节内容

- 面向对象设计原则概述
- 单一职责原则
- 开闭原则
- 里氏代换原则
- 依赖倒转原则
- 接口隔离原则
- 合成复用原则
- 迪米特法则

面向对象设计原则概述

- 软件的可维护性和可复用性
  - 知名软件大师Robert C.Martin认为一个可维护性(Maintainability) 较低的软件设计，通常由于如下4个原因造成：
    - 过于僵硬(Rigidity)
    - 过于脆弱(Fragility)
    - 复用率低(Immobility)
    - 黏度过高(Viscosity)

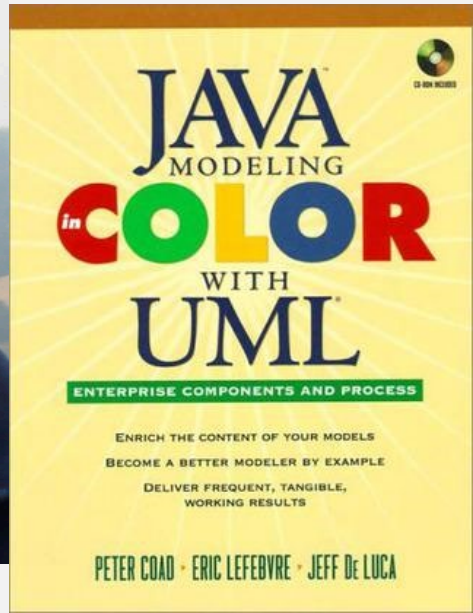
Robert C.Martin



面向对象设计原则概述

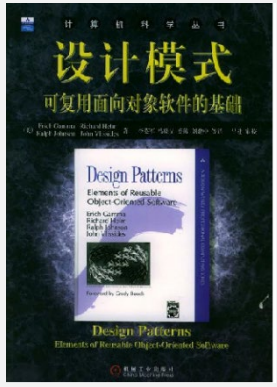
- 软件的可维护性和可复用性
  - 软件工程和建模大师Peter Coad认为，一个好的系统设计应该具备如下三个性质：
    - 可扩展性(Extensibility)
    - 灵活性(Flexibility)
    - 可插入性(Pluggability)

Peter Coad



面向对象设计原则概述

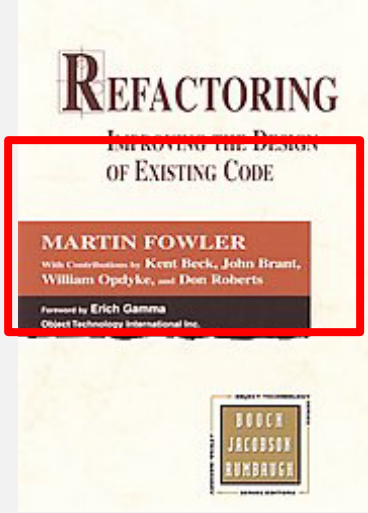
- 软件的可维护性和可复用性
  - 软件的复用(Reuse)或重用拥有众多优点，如可以提高软件的开发效率，提高软件质量，节约开发成本，恰当的复用还可以改善系统的可维护性。
  - 面向对象设计复用的目标在于实现支持可维护性的复用。
  - 在面向对象的设计里面，可维护性复用都是以面向对象设计原则为基础的，这些设计原则首先都是复用的原则，遵循这些设计原则可以有效地提高系统的复用性，同时提高系统的可维护性。



面向对象设计原则概述

- 软件的可维护性和可复用性
  - 面向对象设计原则也是对系统进行合理重构的指南针，重构(Refactoring)是在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性。

Martin Fowler



面向对象设计原则概述

- 面向对象设计原则简介
  - 常用的面向对象设计原则包括7个，这些原则并不是孤立存在的，它们相互依赖，相互补充。

设计原则名称	设计原则简介	重要性
单一职责原则 (Single Responsibility Principle, SRP)	类的职责要单一，不能将太多的职责放在一个类中	★★★★☆
开闭原则 (Open-Closed Principle, OCP)	软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能	★★★★★
里氏代换原则 (Liskov Substitution Principle, LSP)	在软件系统中，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★☆
依赖倒转原则 (Dependency Inversion Principle, DIP)	要针对抽象层编程，而不要针对具体类编程	★★★★★
接口隔离原则 (Interface Segregation Principle, ISP)	使用多个专门的接口来取代一个统一的接口	★★☆☆☆
合成复用原则 (Composite Reuse Principle, CRP)	在系统中应该尽量多使用组合和聚合关联关系，尽量少使用甚至不使用继承关系	★★★★☆
迪米特法则 (Law of Demeter, LoD)	一个软件实体对其他实体的引用越少越好，或者说如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，而是通过引入一个第三者发生间接交互	★★★☆☆

单一职责原则

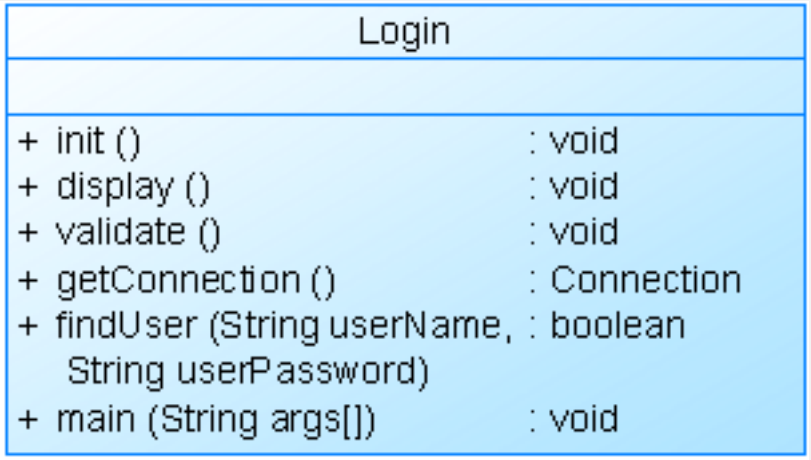
- 单一职责原则定义
  - 单一职责原则(Single Responsibility Principle, SRP)定义如下：
    - 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。
  - 其英文定义为：
    - Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.
  - 另一种定义方式如下：
    - 就一个类而言，应该仅有一个引起它变化的原因。
  - 其英文定义为：
    - There should never be more than one reason for a class to change.

单一职责原则

- 单一职责原则分析
  - 一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小，而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作。
  - 类的职责主要包括两个方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现。
  - 单一职责原则是实现高内聚、低耦合的指导方针，在很多代码重构手法中都能找到它的存在，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

单一职责原则

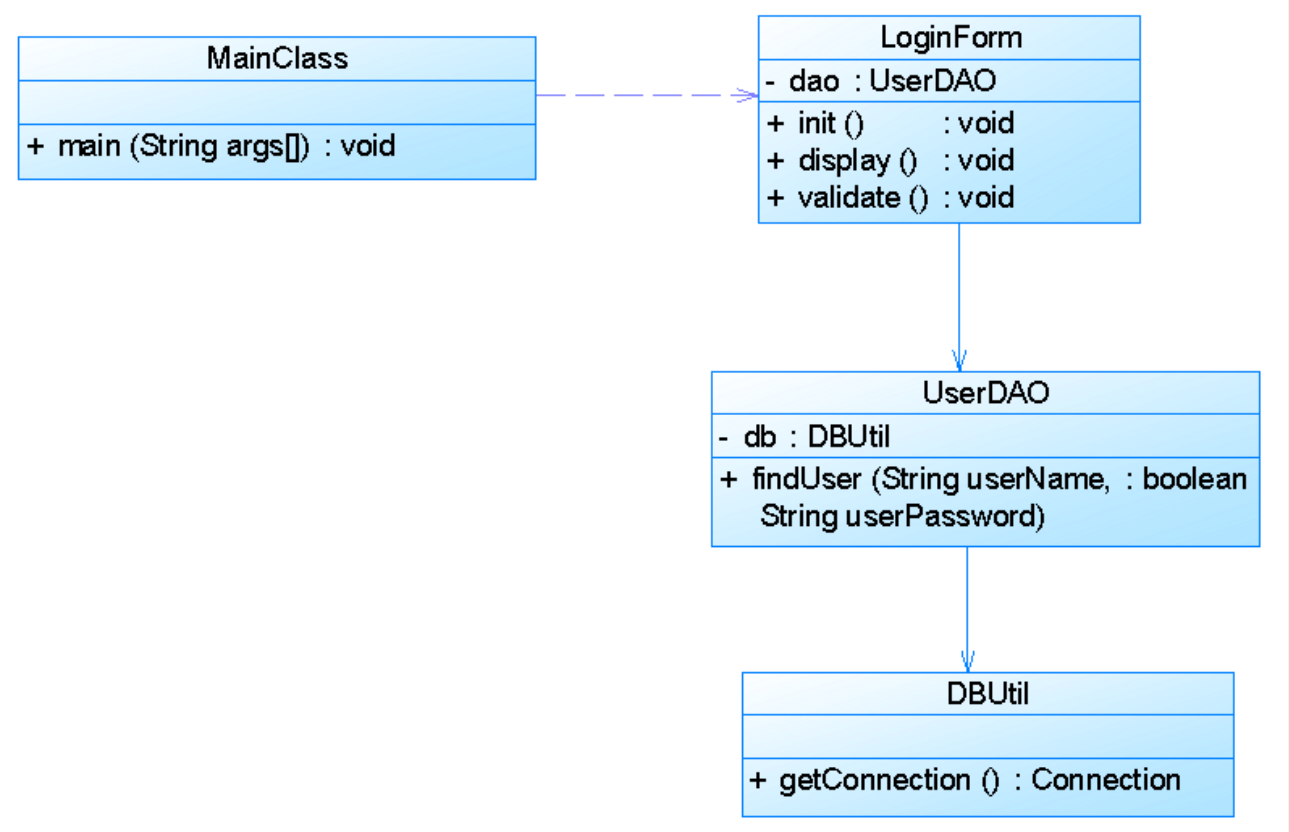
- 单一职责原则实例
  - 实例说明
    - 某基于Java的C/S系统的“登录功能”通过如下登录类(Login)实现：



    - 现使用单一职责原则对其进行重构。

单一职责原则

- 单一职责原则实例
  - 实例解析

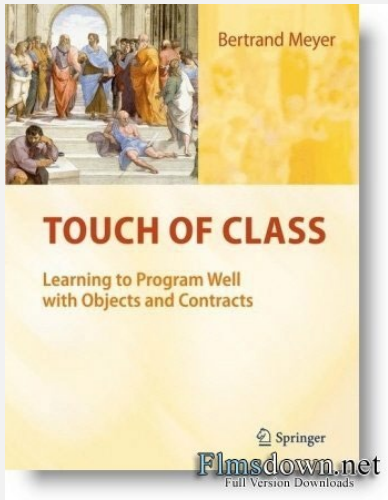


开闭原则

- 开闭原则定义
  - 开闭原则(Open-Closed Principle, OCP)定义如下：
    - 一个软件实体应当对扩展开放，对修改关闭。也就是说在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。
  - 其英文定义为：
    - Software entities should be open for extension, but closed for modification.

开闭原则

- 开闭原则分析
  - 开闭原则由Bertrand Meyer于1988年提出，它是面向对象设计中最重要的原则之一。
  - 在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

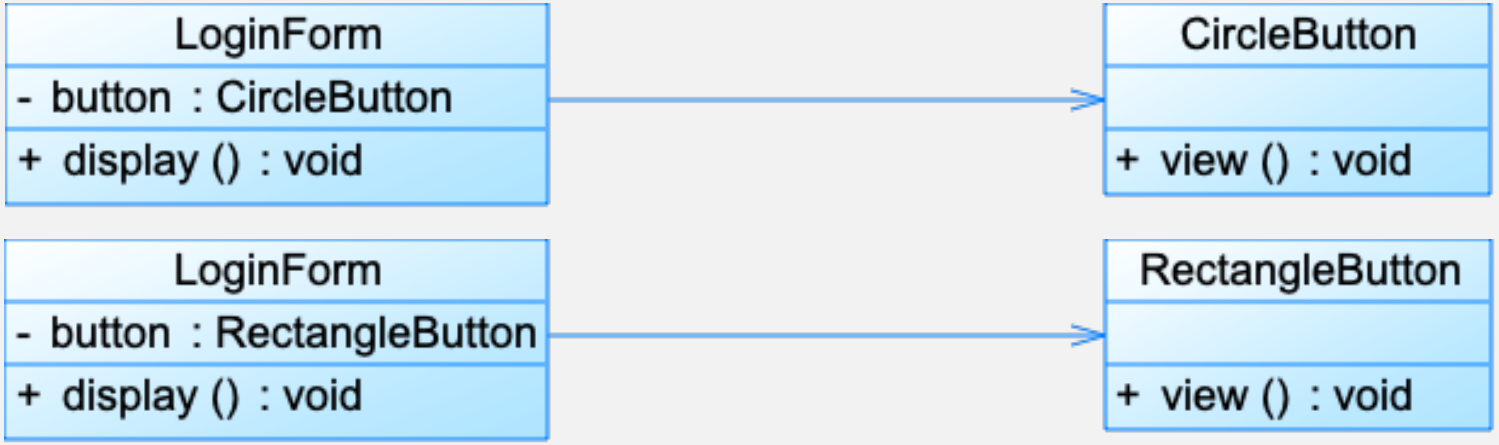


开闭原则

- 开闭原则分析
  - 抽象化是开闭原则的关键。
  - 开闭原则还可以通过一个更加具体的“对可变性封装原则”来描述，对可变性封装原则(Principle of Encapsulation of Variation, EVP)要求找到系统的可变因素并将其封装起来。

开闭原则

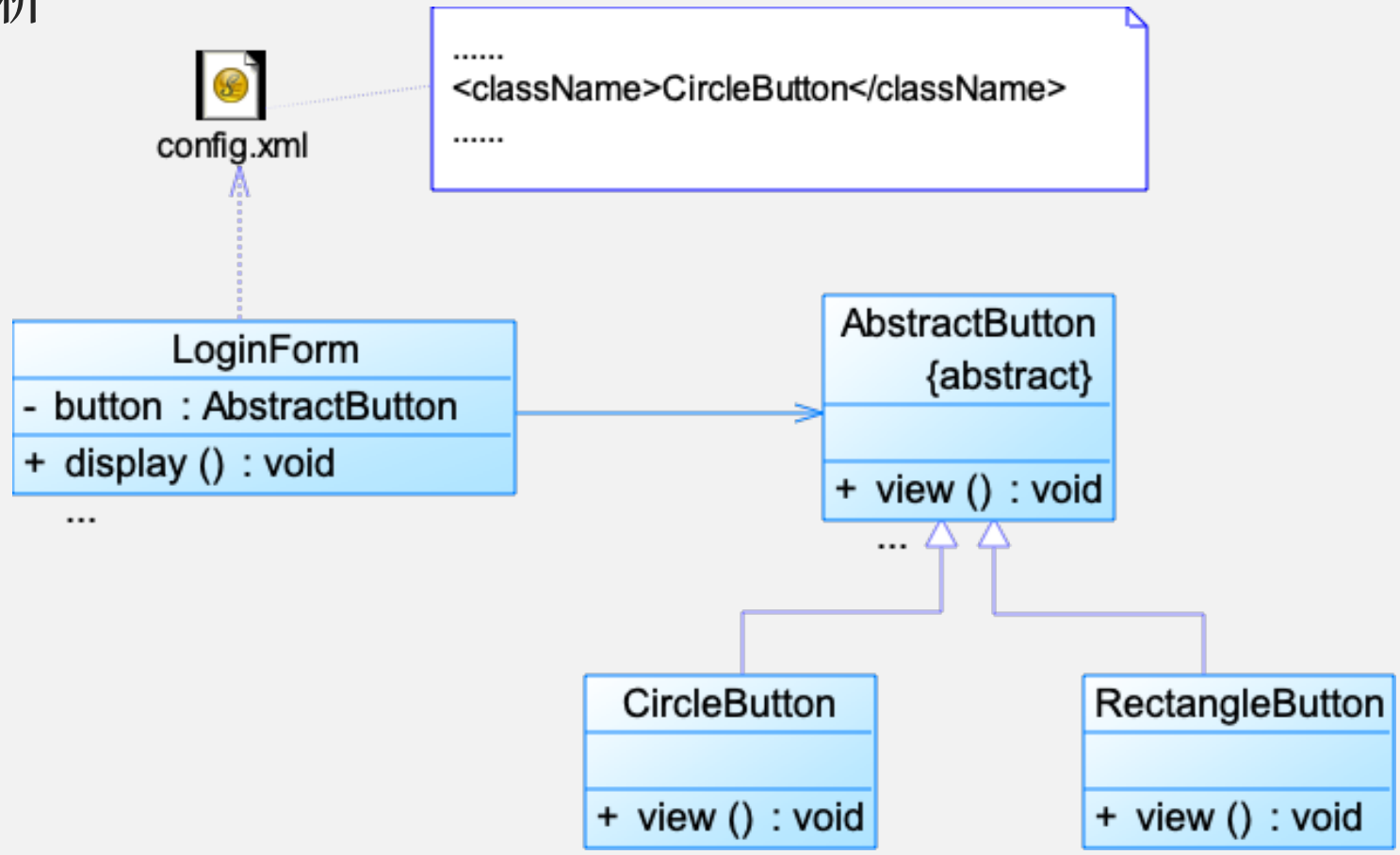
- 开闭原则实例
  - 实例说明
    - 某图形界面系统提供了各种不同形状的按钮，客户端代码可针对这些按钮进行编程，用户可能会改变需求要求使用不同的按钮，原始设计方案如图所示：



    - 现对该系统进行重构，使之满足开闭原则的要求。

开闭原则

- 开闭原则实例
  - 实例解析



里氏代换原则

- 里氏代换原则定义
  - 里氏代换原则(Liskov Substitution Principle, LSP)有两种定义方式，第一种定义方式相对严格，其定义如下：
    - 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o2都代换成o1时，程序P的行为没有变化，那么类型S是类型T的子类型。
  - 其英文定义为：
    - If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
  - 第二种更容易理解的定义方式如下：
    - 所有引用基类（父类）的地方必须能透明地使用其子类的对象。
  - 其英文定义为：
    - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

里氏代换原则

- 里氏代换原则分析
  - 里氏代换原则可以通俗表述为：在软件中如果能够使用基类对象，那么一定能够使用其子类对象。把基类都替换成它的子类，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类的话，那么它不一定能够使用基类。
  - 里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

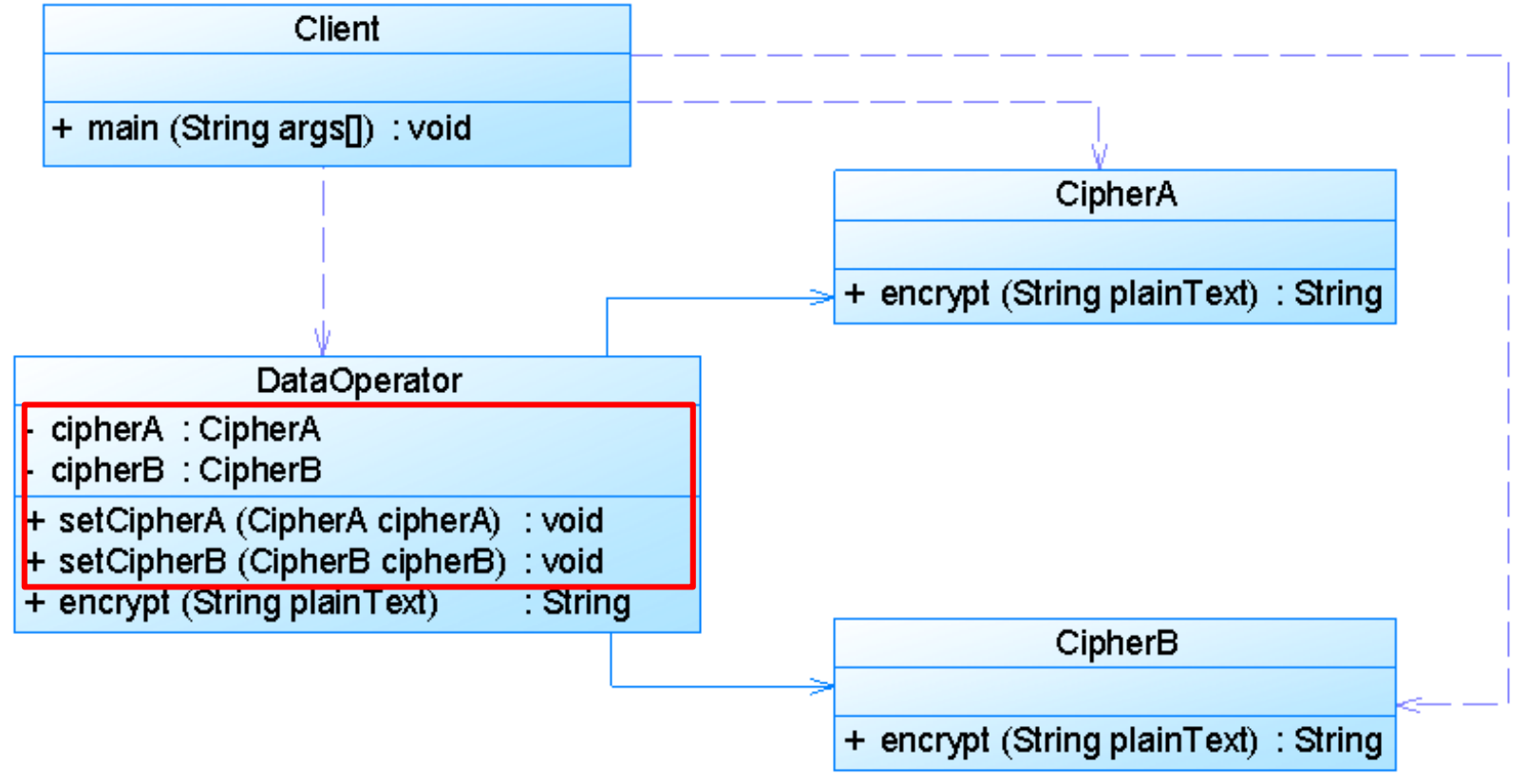
里氏代换原则

- 里氏代换原则由2008年图灵奖得主、美国第一位计算机科学女博士、麻省理工学院教授Barbara Liskov和卡内基.梅隆大学Jeannette Wing教授于1994年提出。其原文如下：Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

芭芭拉·利斯科夫（**Barbara Liskov**），美国计算机科学家，2008年图灵奖得主，2004年约翰.冯诺依曼奖得主，美国工程院院士，美国艺术与科学院院士，美国计算机协会会士。她是美国第一个计算机科学女博士。

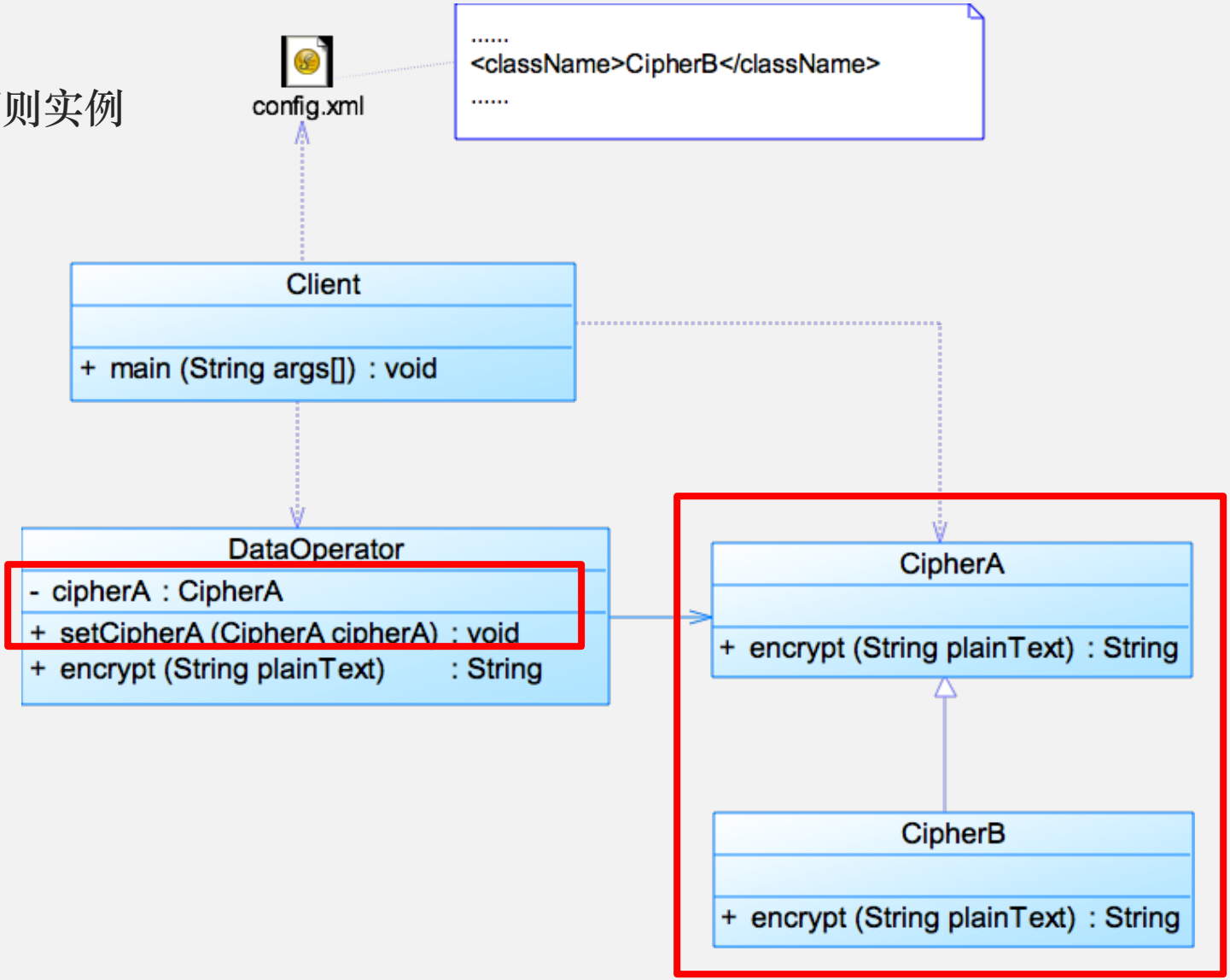
里氏代换原则

- 里氏代换原则实例
  - 实例说明



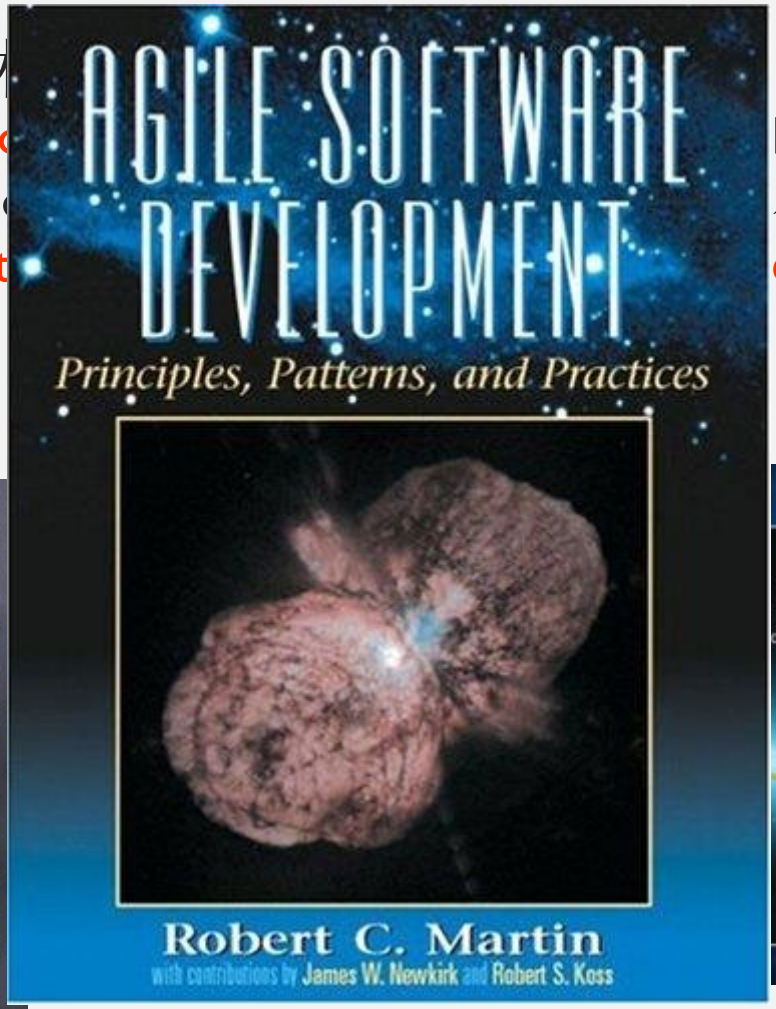
里氏代换原则

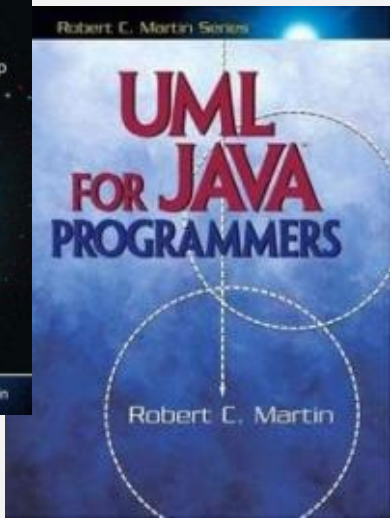
- 里氏代换原则实例
  - 实例解析



依赖倒转原则

- 依赖倒转原则分
  - 依赖倒转原则是R Reporter》所写的专栏Engineering N 在2002年出版的经典著作《Agile Soft erns, and Practices》中。





依赖倒转原则

- 依赖倒转原则定义
  - 依赖倒转原则(Dependence Inversion Principle, DIP)的定义如下：
    - 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。
  - 其英文定义为：
    - High level modules should not depend upon low level modules, both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions.
  - 另一种表述为：
    - 要针对接口编程，不要针对实现编程。
  - 其英文定义为：
    - Program to an interface, not an implementation.

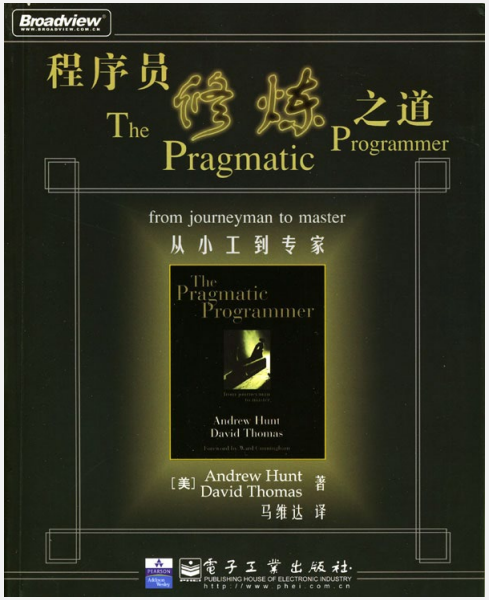
依赖倒转原则

- 依赖倒转原则分析
  - 简单来说，依赖倒转原则就是指：代码要依赖于抽象的类，而不要依赖于具体的类；要针对接口或抽象类编程，而不是针对具体类编程。
  - 实现开闭原则的关键是抽象化，并且从抽象化导出具体化实现，如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要手段。

依赖倒转原则

- 依赖倒转原则分析
  - 依赖倒转原则的常用实现方式之一是在代码中使用抽象类，而将具体类放在配置文件中。
    - “将抽象放进代码，将细节放进元数据”
    - Put Abstractions in Code, Details in Metadata

（《程序员修炼之道：从小工到专家》(The Pragmatic programmer: from journeyman to master)）

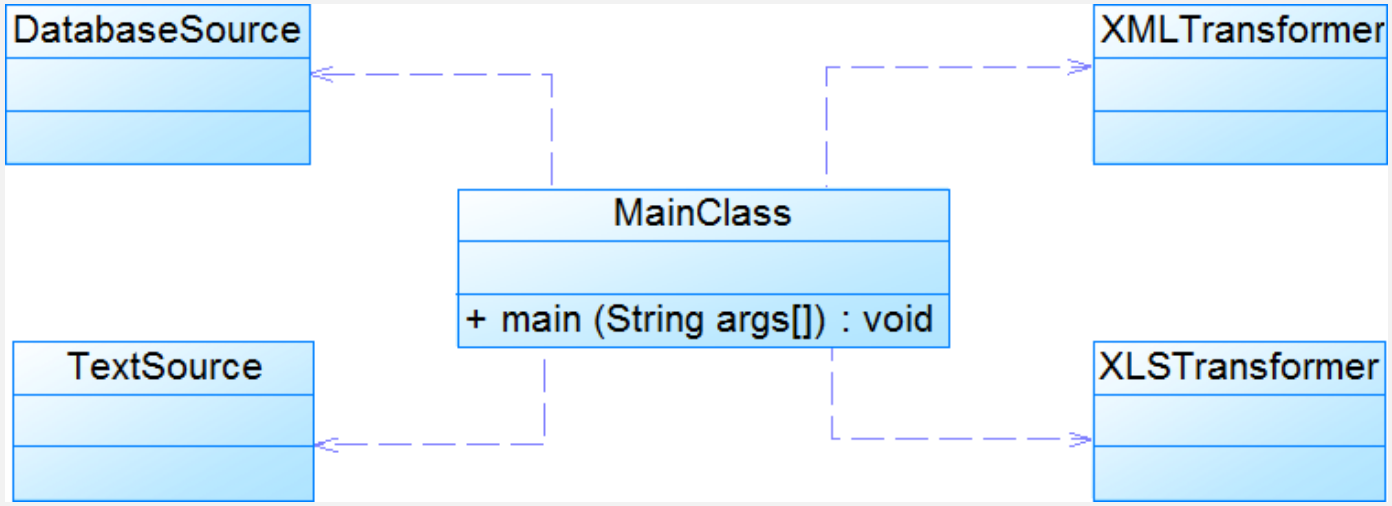


依赖倒转原则

- 依赖倒转原则分析
  - 类之间的耦合
    - 零耦合关系
    - 具体耦合关系
    - 抽象耦合关系
  - 依赖倒转原则要求客户端依赖于抽象耦合，以抽象方式耦合是依赖倒转原则的关键。

依赖倒转原则

- 依赖倒转原则实例
  - 实例说明
    - 某系统提供一个数据转换模块，可以将来自不同数据源的数据转换成多种格式，如可以转换来自数据库的数据(DatabaseSource)、也可以转换来自文本文件的数据(TextSource)，转换后的格式可以是XML文件(XMLTransformer)、也可以是XLS文件(XLSTransformer)等。

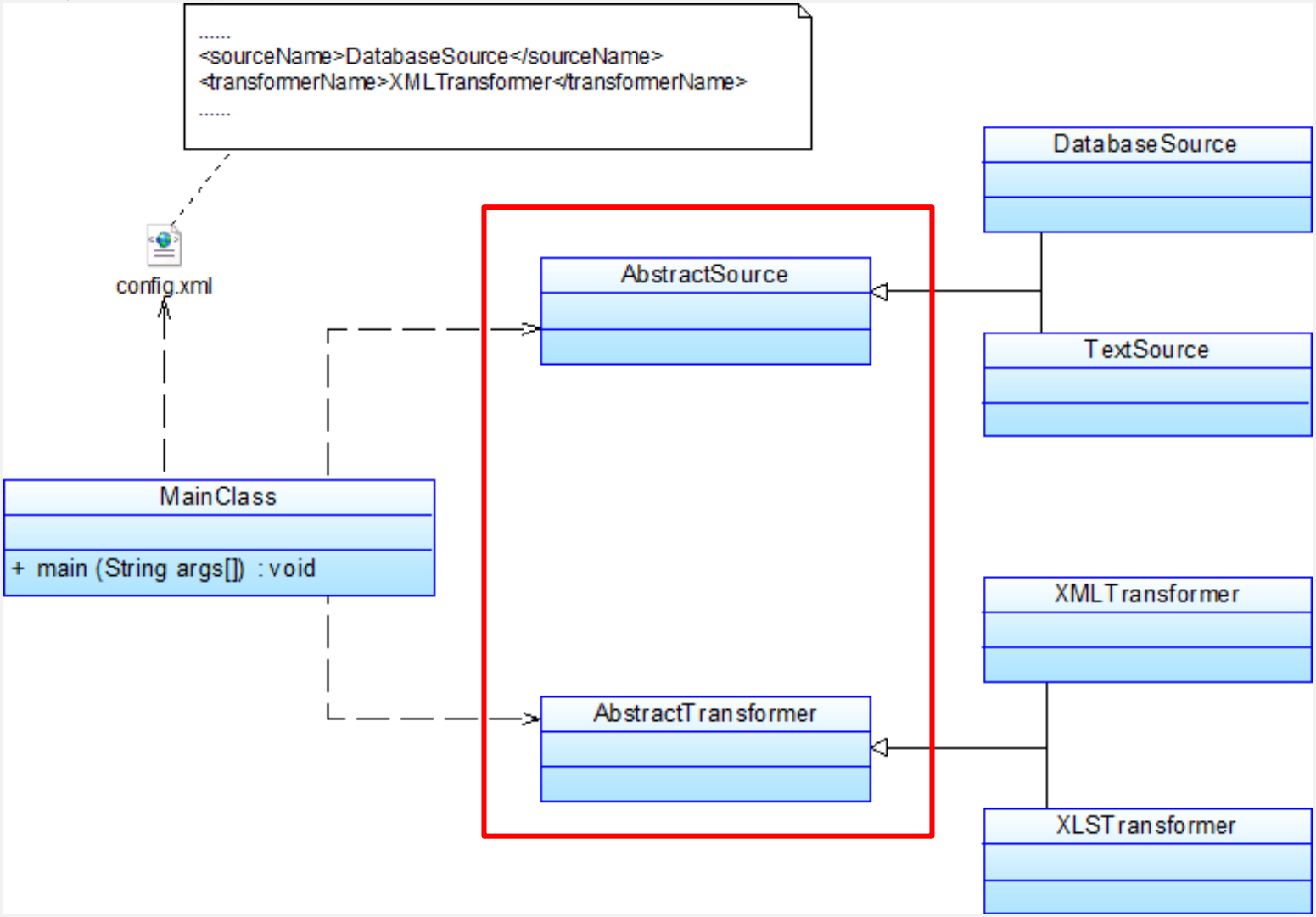


依赖倒转原则

- 依赖倒转原则实例
  - 实例说明
    - 由于需求的变化，该系统可能需要增加新的数据源或者新的文件格式，每增加一个新的类型的数据源或者新的类型的文件格式，客户类MainClass都需要修改源代码，以便使用新的类，但违背了开闭原则。现使用依赖倒转原则对其进行重构。

依赖倒转原则

- 依赖倒转原则实例
  - 实例解析



接口隔离原则

- 接口隔离原则定义
  - 接口隔离原则(Interface Segregation Principle, ISP)的定义如下：
    - 客户端不应该依赖那些它不需要的接口。
  - 其英文定义为：
    - Clients should not be forced to depend upon interfaces that they do not use.
  - 注意，在该定义中的接口指的是所定义的方法。
  - 另一种定义方法如下：
    - 一旦一个接口太大，则需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。
  - 其英文定义为：
    - Once an interface has gotten too 'fat' it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them.

接口隔离原则

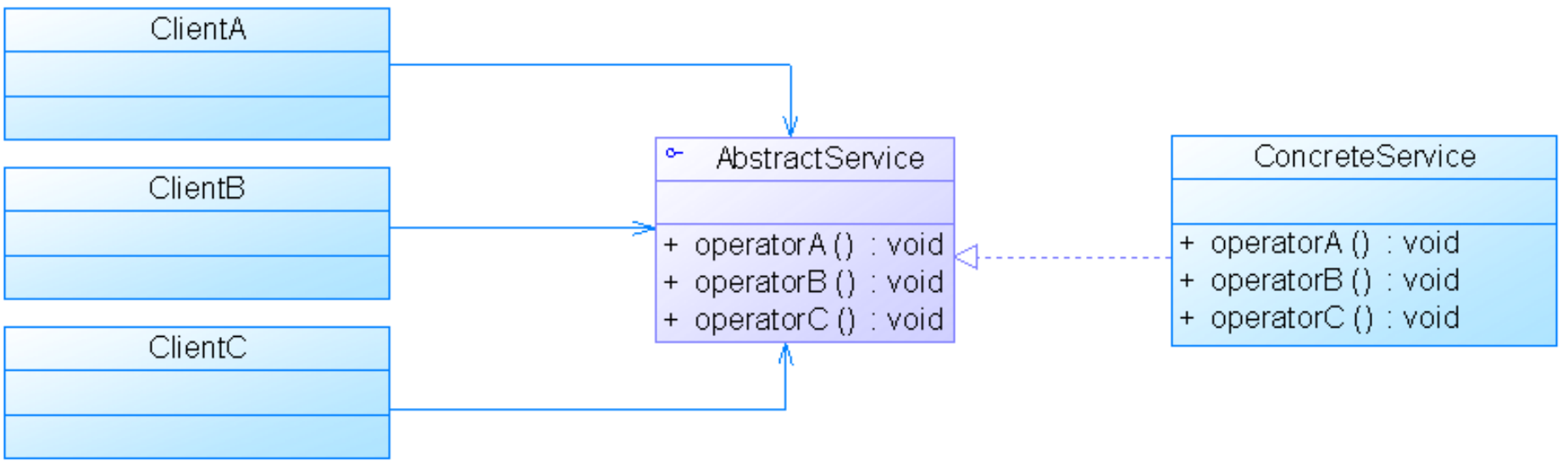
- 接口隔离原则分析
  - 接口隔离原则是指使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色，不多不少，不干不该干的事，该干的事都要干。
    - (1) 一个接口就只代表一个角色，每个角色都有它特定的一个接口，此时这个原则可以叫做“角色隔离原则”。
    - (2) 接口仅仅提供客户端需要的行为，即所需的方法，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

接口隔离原则

- 接口隔离原则分析
  - 使用接口隔离原则拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。
  - 可以在进行系统设计时采用定制服务的方式，即为不同的客户端提供宽窄不同的接口，只提供用户需要的行为，而隐藏用户不需要的行为。

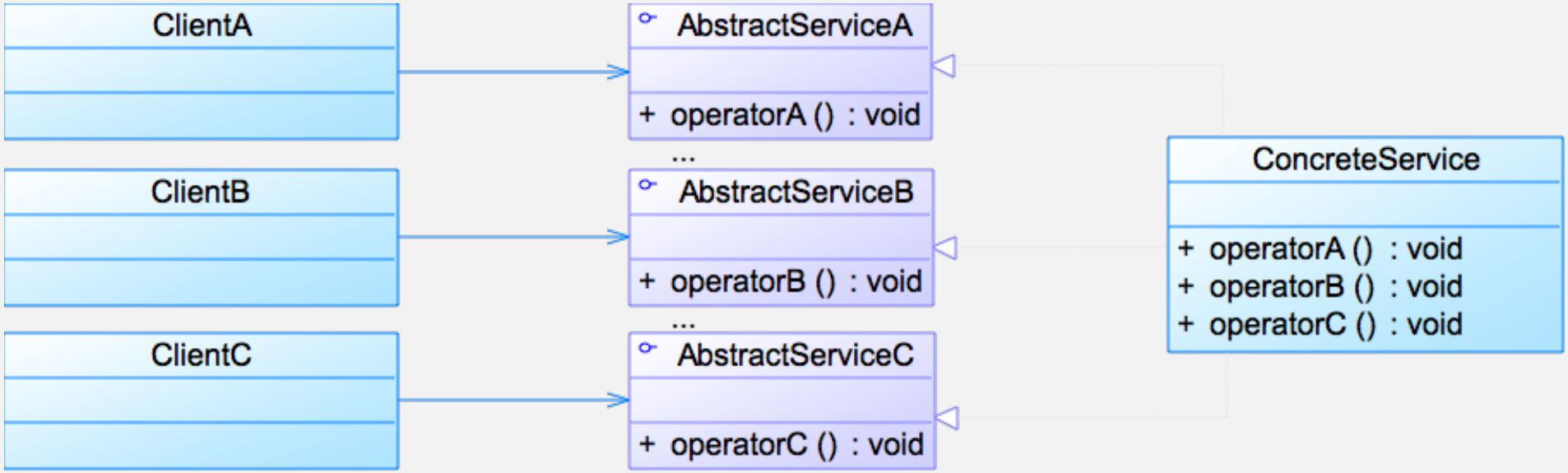
接口隔离原则

- 接口隔离原则实例
  - 实例说明
    - 下图展示了一个拥有多个客户类的系统，在系统中定义了一个巨大的接口（胖接口）AbstractService来服务所有的客户类。可以使用接口隔离原则对其进行重构。



接口隔离原则

- 接口隔离原则实例
  - 实例解析



合成复用原则

- 合成复用原则定义
  - 合成复用原则(Composite Reuse Principle, CRP)又称为组合/聚合复用原则(Composition/ Aggregate Reuse Principle, CARP)，其定义如下：
    - 尽量使用对象组合，而不是继承来达到复用的目的。
  - 其英文定义为：
    - Favor composition of objects over inheritance as a reuse mechanism.

合成复用原则

- 合成复用原则分析
  - 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。

合成复用原则

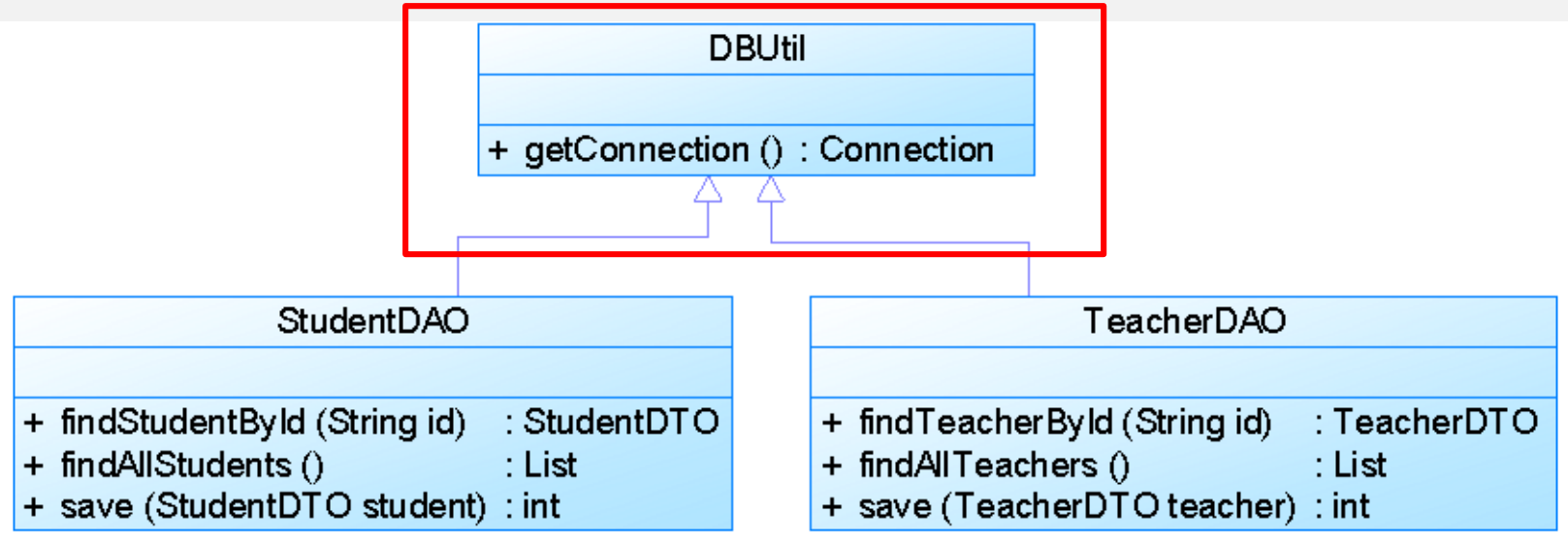
- 合成复用原则分析
  - 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。
    - 继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用 ）
    - 组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用 ）

合成复用原则

- 合成复用原则分析
  - 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

合成复用原则

- 合成复用原则实例
  - 实例说明
    - 某教学管理系统部分数据库访问类设计如图所示：

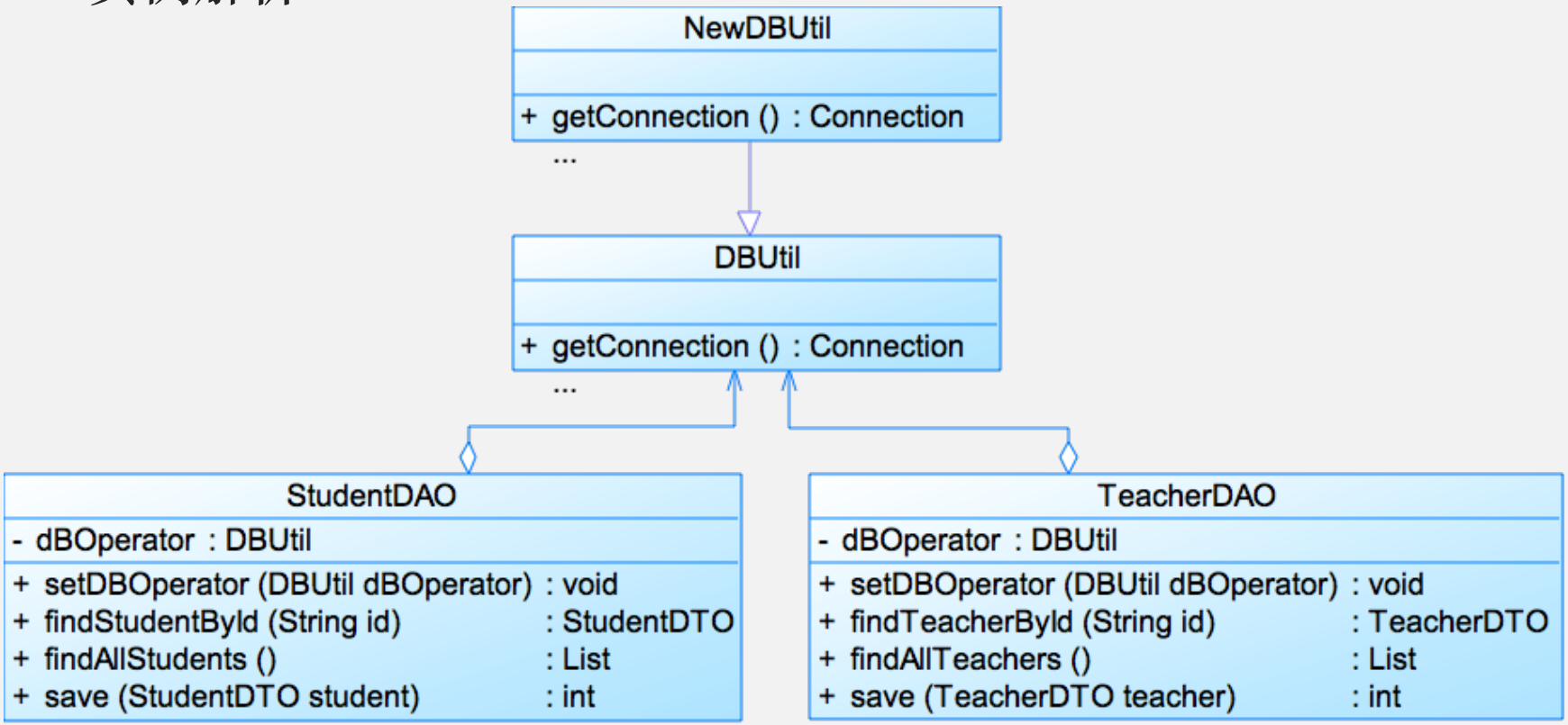


合成复用原则

- 合成复用原则实例
  - 实例说明
    - 如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需要修改DBUtil类源代码。如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类，这将违背开闭原则，系统扩展性较差。
    - 现使用合成复用原则对其进行重构。

合成复用原则

- 合成复用原则实例
  - 实例解析



最小知识原则（迪米特法则）

- 迪米特法则定义
  - 迪米特法则(Law of Demeter, LoD)又称为最少知识原则(Least Knowledge Principle, LKP)，它有多种定义方法，其中几种典型定义如下：
    - (1) 不要和“陌生人”说话。英文定义为：Don't talk to strangers.
    - (2) 只与你的直接朋友通信。英文定义为：Talk only to your immediate friends.
    - (3) 每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。英文定义为：Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.

迪米特法则

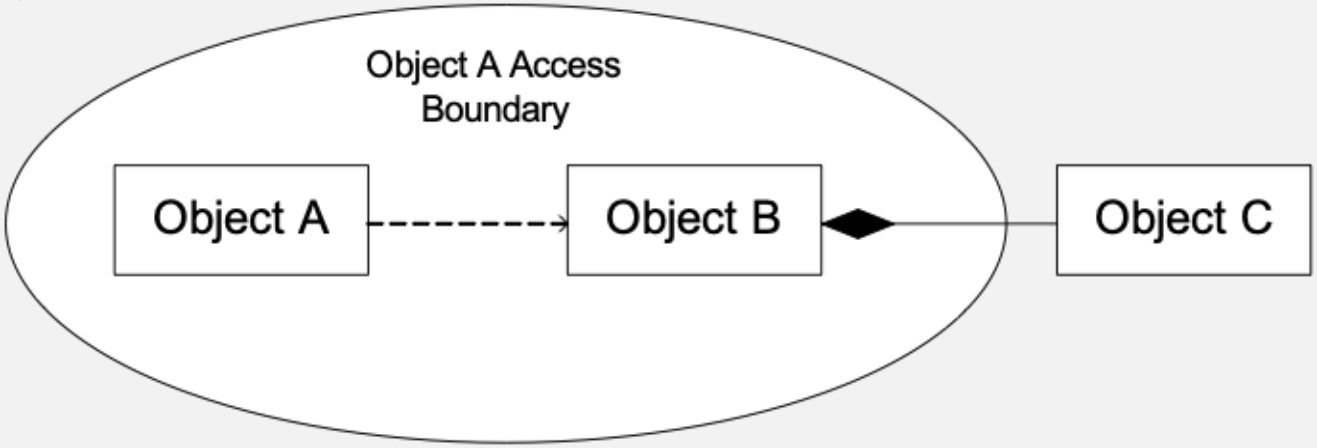
- 迪米特法则分析
  - 迪米特法则来自于1987年秋美国东北大学(Northeastern University)一个名为“Demeter”的研究项目。
  - 简单地说，迪米特法则就是指一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。

迪米特法则

- 迪米特法则分析
  - 在迪米特法则中，对于一个对象，其朋友包括以下几类：
    - (1) 当前对象本身(this)；
    - (2) 以参数形式传入到当前对象方法中的对象；
    - (3) 当前对象的成员对象；
    - (4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；
    - (5) 当前对象所创建的对象。
  - 任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。

迪米特法则

- 迪米特法则分析
  - 迪米特法则可分为狭义法则和广义法则。在狭义的迪米特法则中，如果两个类之间不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。



迪米特法则

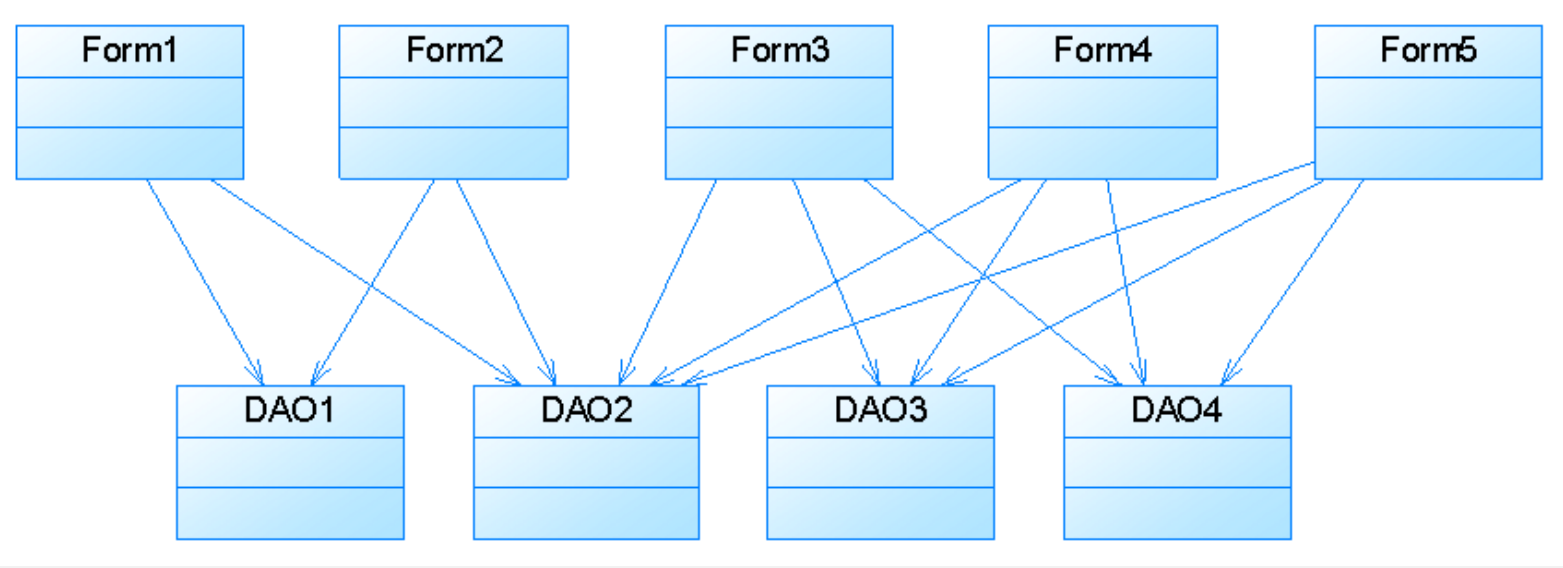
- 迪米特法则分析
  - 狭义的迪米特法则：可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落，它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。
  - 广义的迪米特法则：指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改，同时可以促进软件的复用，由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，而信息隐藏的重要性也就越明显。

迪米特法则

- 迪米特法则分析
  - 迪米特法则的主要用途在于控制信息的过载：
    - 在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；
    - 在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；
    - 在类的设计上，只要有可能，一个类型应当设计成不变类；
    - 在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

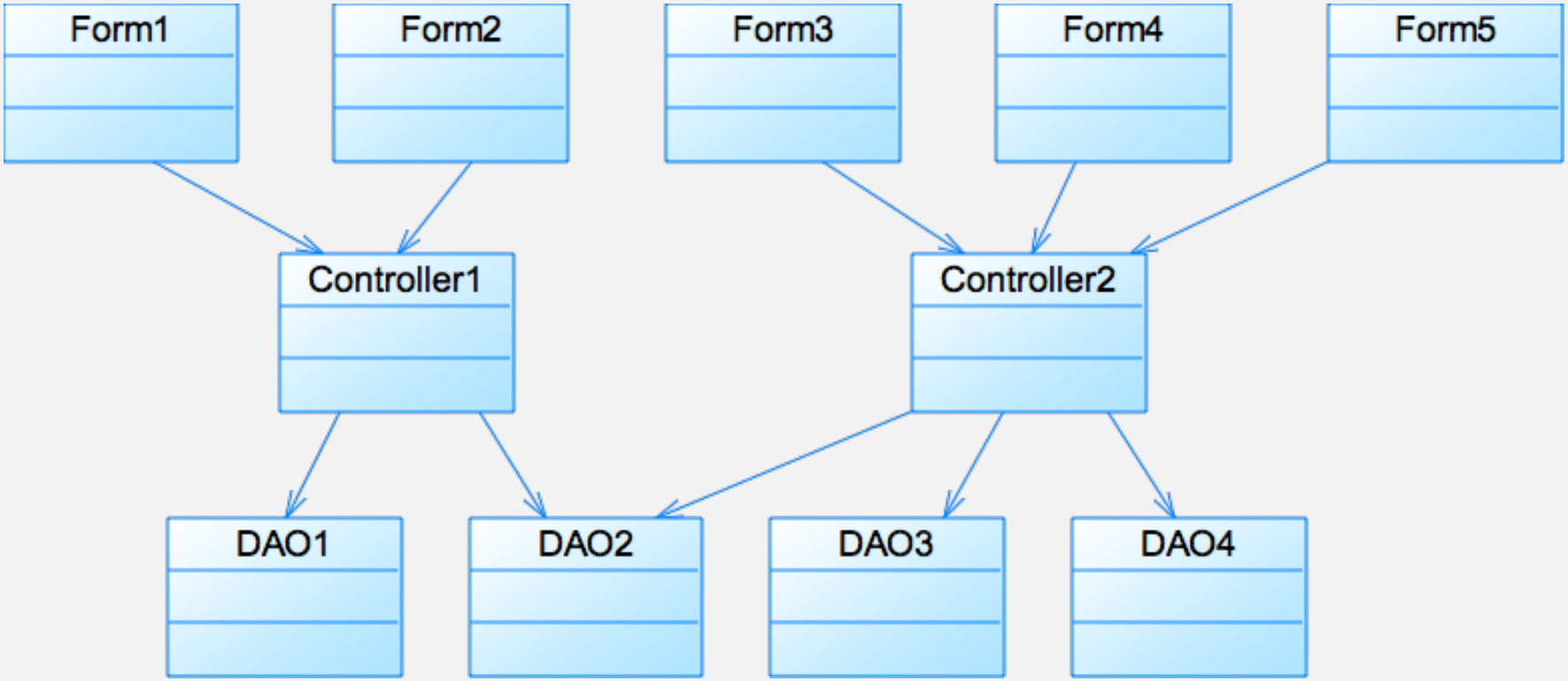
迪米特法则

- 迪米特法则实例
  - 实例说明
    - 某系统界面类(如Form1、Form2等类)与数据访问类(如DAO1、DAO2等类)之间的调用关系较为复杂，如图所示：



迪米特法则

- 迪米特法则实例
  - 实例解析



思考

- 在 JDK 中，java.util.Stack是java.util.Vector类的子类，该设计合理吗？若不合理，请分析解释该设计存在的问题。

小结

- 对于面向对象的软件系统设计来说，在支持可维护性的同时，需要提高系统的可复用性。
- 软件的复用可以提高软件的开发效率，提高软件质量，节约开发成本，恰当的复用还可以改善系统的可维护性。
- 单一职责原则要求在软件系统中，一个类只负责一个功能领域中的相应职责。
- 开闭原则要求一个软件实体应当对扩展开放，对修改关闭，即在不修改源代码的基础上扩展一个系统的行为。
- 里氏代换原则可以通俗表述为在软件中如果能够使用基类对象，那么一定能够使用其子类对象。

小结

- 依赖倒转原则要求抽象不应该依赖于细节，细节应该依赖于抽象；要针对接口编程，不要针对实现编程。
- 接口隔离原则要求客户端不应该依赖那些它不需要的接口，即将一些大的接口细化成一些小的接口供客户端使用。
- 合成复用原则要求复用时尽量使用对象组合，而不使用继承。
- 迪米特法则要求一个软件实体应当尽可能少的与其他实体发生相互作用。

小结

- 目标：开闭原则
- 指导：最小知识原则
- 基础：单一职责原则、可变性封装原则
- 实现：依赖倒转原则、合成复用原则、里氏代换原则、接口隔离原则