CS10102302
软件设计

赵君峤 副教授

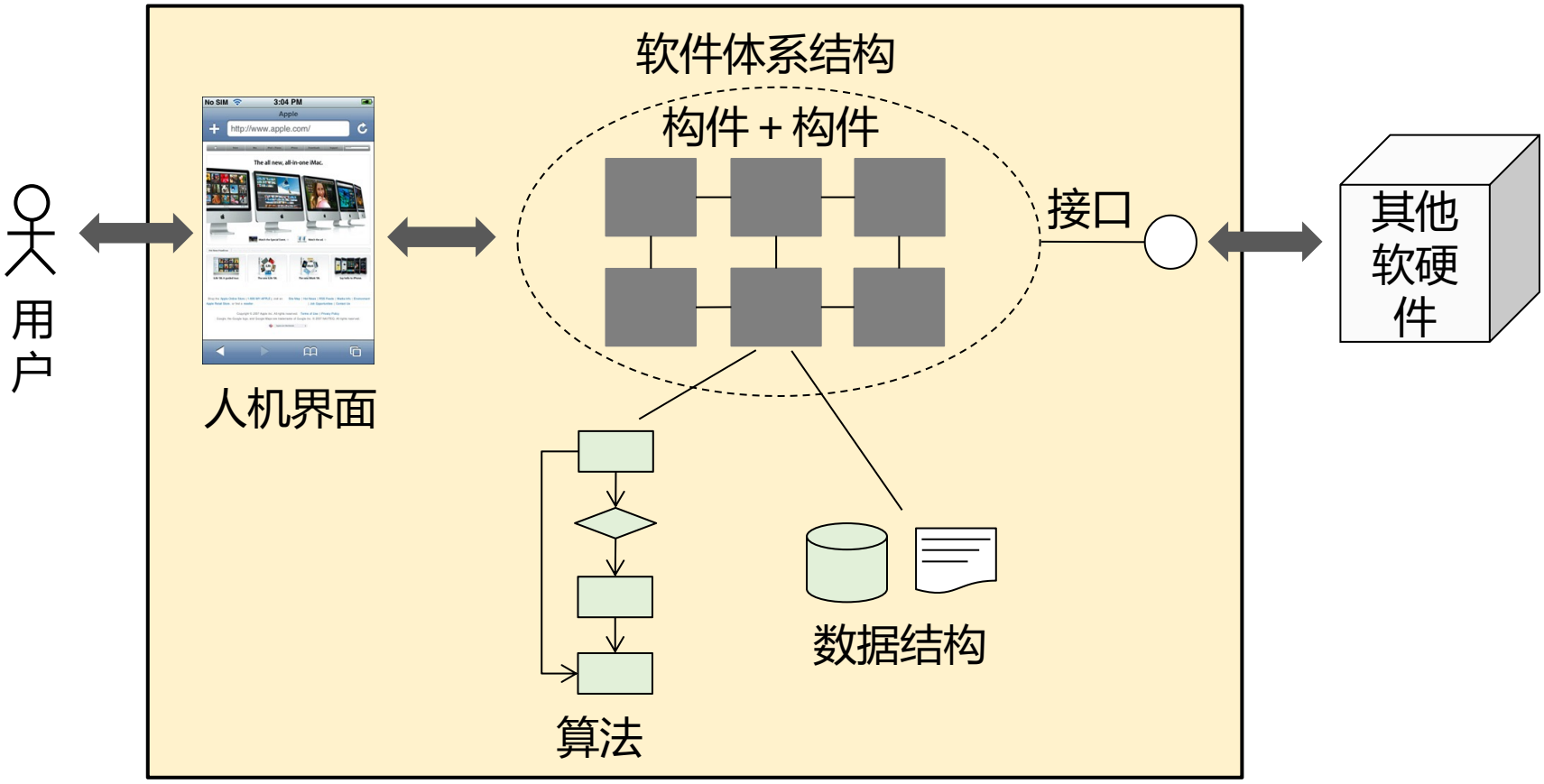
计算机科学与技术系 电子与信息工程学院

同济大学

软件设计

- 站在代码之上
- 抽象的思考
- 不仅仅是软件体系结构

软件设计元素



软件设计的驱动力

- 需求分析
- 非功能需求
  - 质量、性能、安全性等
- 约束
  - 时间、成本、技术积累等

系统设计过程

建立系统总体结构

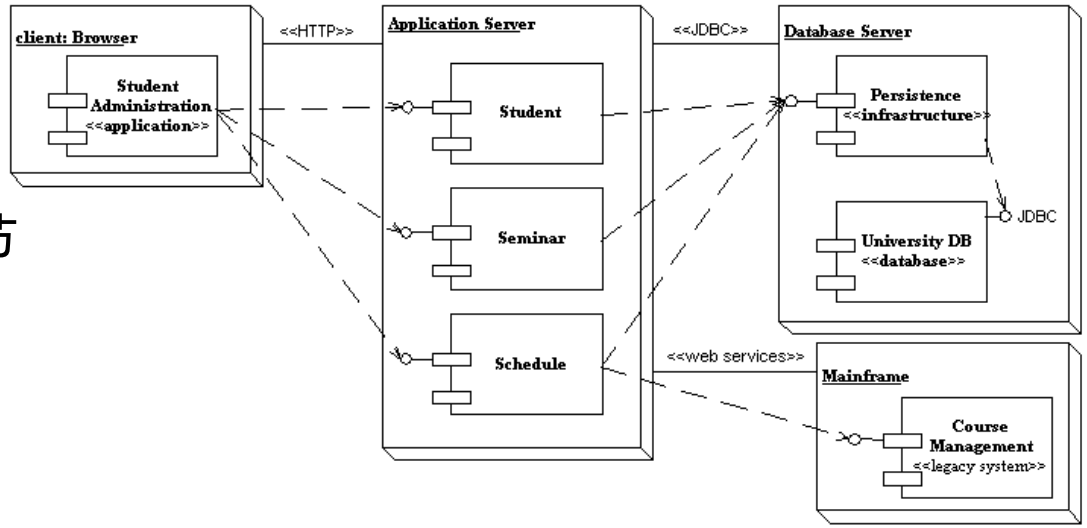
- 定义系统的设计目标
- 划分若干子系统

确定系统的物理部署

- 选择软硬件平台
- 将子系统映射到物理节点

定义设计策略

- 确定数据存储策略
- 确定访问控制策略
- 设计全局控制流



子系统设计过程

开始于体系结构模型

- 类的模型：概念类和框架类
- 总体的状态模型
- 用例模型

引入设计类和设计模式

- 连接概念类和框架类
- 细化设计模型，确保一致和完整

详细设计类

- 定义属性、操作、状态等

软件设计文档（IEEE 1016-1998）

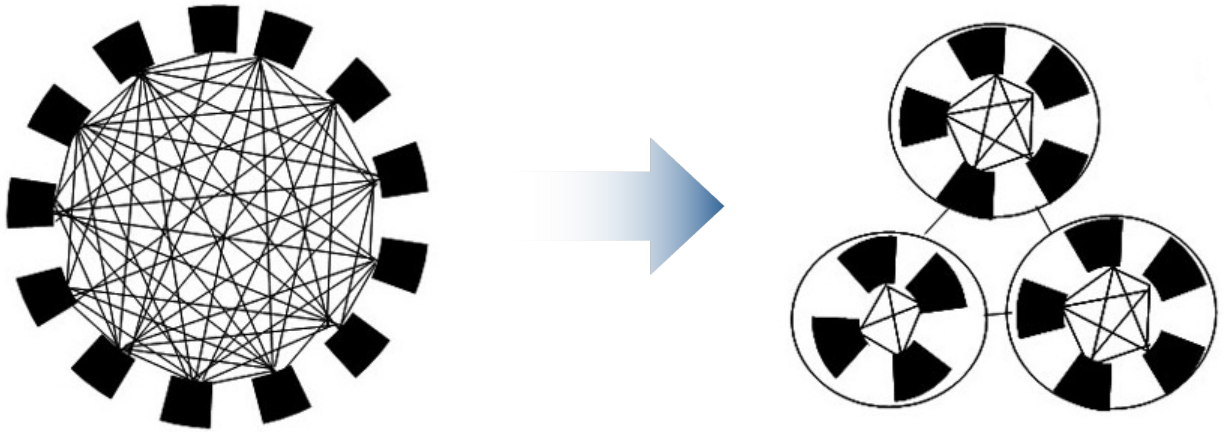
1. 引言
   - 1.1 目的
   - 1.2 范围
   - 1.3 定义和缩写词
2. 参考文献
3. 分解说明
   - 3.1 模块分解
   - 3.2 并发进程
   - 3.3 数据分解
4. 依赖关系说明
   - 4.1 模块间的依赖关系
   - 4.2 进程间的依赖关系
   - 4.3 数据间的依赖关系
5. 接口说明
   - 5.1 模块接口
   - 5.2 进程接口
6. 详细设计
   - 6.1 模块详细设计
   - 6.2 数据详细设计

附录

注：3-5是体系结构设计；6是详细设计。

处理复杂性

对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。



一个平面规划的例子

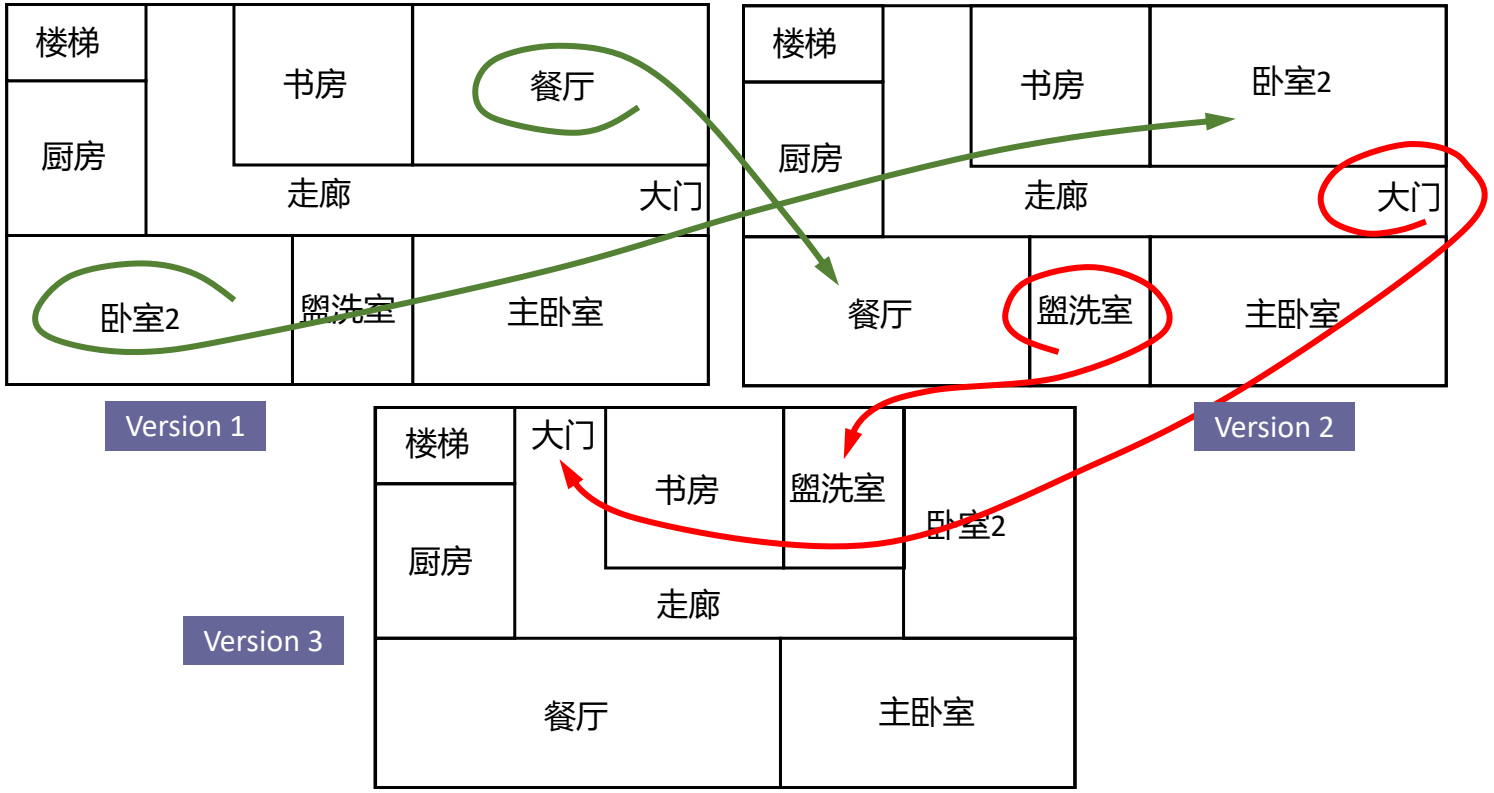
住宅设计的约束条件：

- 应该有2个卧室
- 书房、厨房和客厅/餐厅各1个
- 住户每天行走的路程要求最短
- 卧室的白天采光量要求达到最大

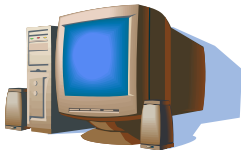
假设：住户的大部分时间集中在客厅/餐厅和主卧室的区域内活动

如何给出满足上述约束的平面规划方案？

一个平面规划的例子


楼梯
厨房
书房
餐厅
走廊
大门
卧室2
盥洗室
主卧室
Version 1
楼梯
厨房
书房
卧室2
走廊
大门
餐厅
盥洗室
主卧室
Version 2
楼梯
厨房
大门
书房
盥洗室
卧室2
走廊
餐厅
主卧室
Version 3

一个平面规划的例子

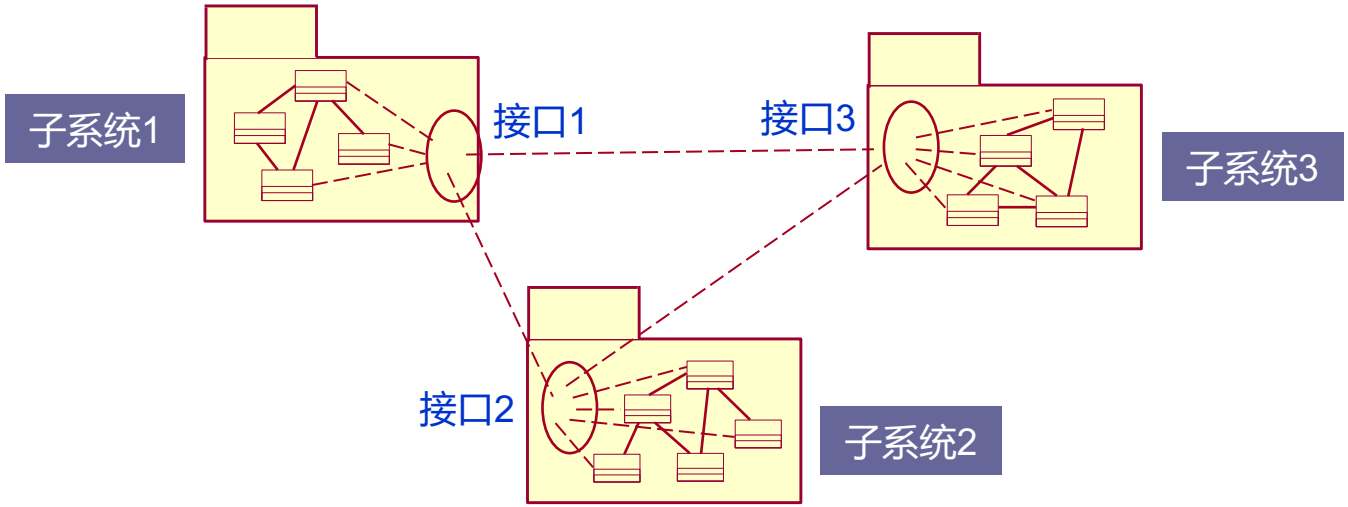


与软件工程概念的映射

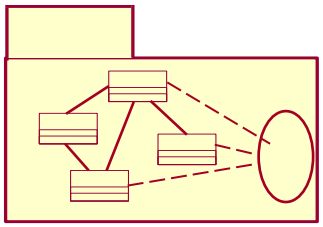
	建筑结构概念	软件工程概念
构件	房间	子系统
接口	门	服务
非功能需求	生活区	响应时间
功能需求	住房	用例
返工代价	移动墙壁	子系统接口的改变

分解

在求解域中，我们将整个系统分解成多个更小的部分，称为子系统。在面向对象系统中，它是由多个表达子求解域的类组成。



子系统与接口

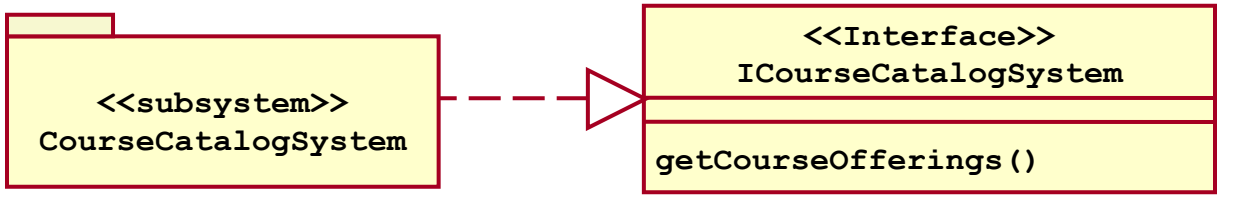


- 子系统的特征是由该子系统提供给其他子系统的服务来刻画的，一个服务是一组分担公共目的的相关操作。
- 子系统提供给其他子系统的一组操作称为子系统接口，包括操作名、操作参数、类型及其返回值。

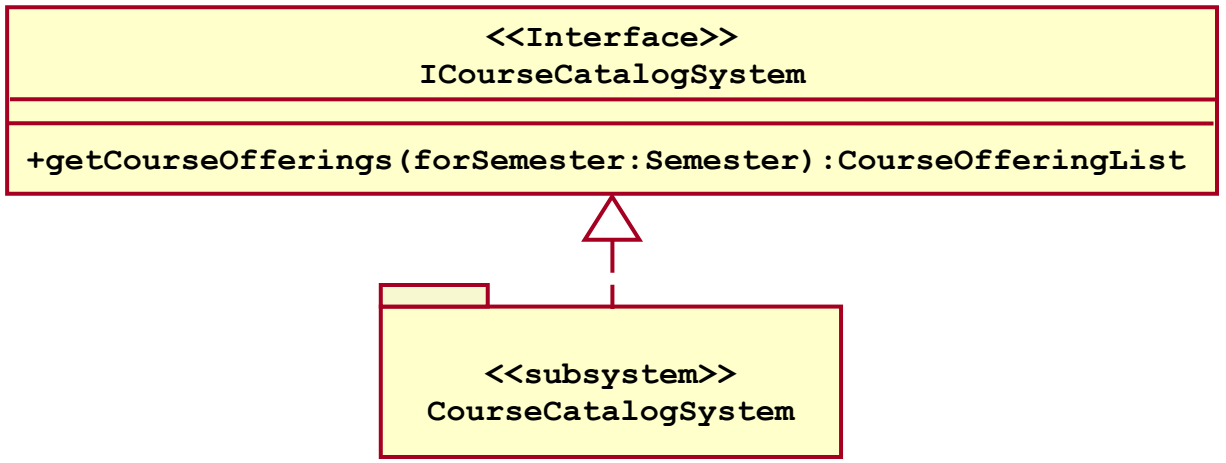
- 系统设计关注于定义子系统提供的服务，即列出所有操作以及这些操作的参数和操作的高层行为。
- 构件设计关注于应用程序接口（API），细化并扩展子系统接口，也包括各个操作参数的类型和返回值。

子系统与接口

系统设计阶段
子系统服务



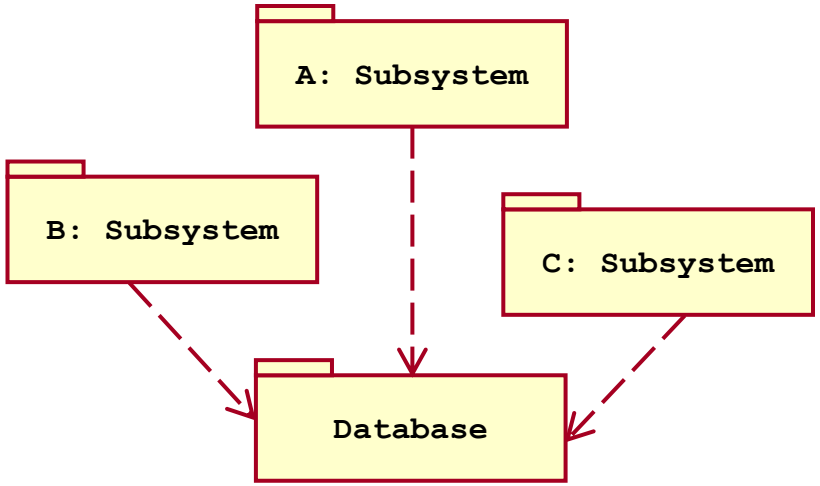
构件设计阶段
子系统接口



系统分解原则

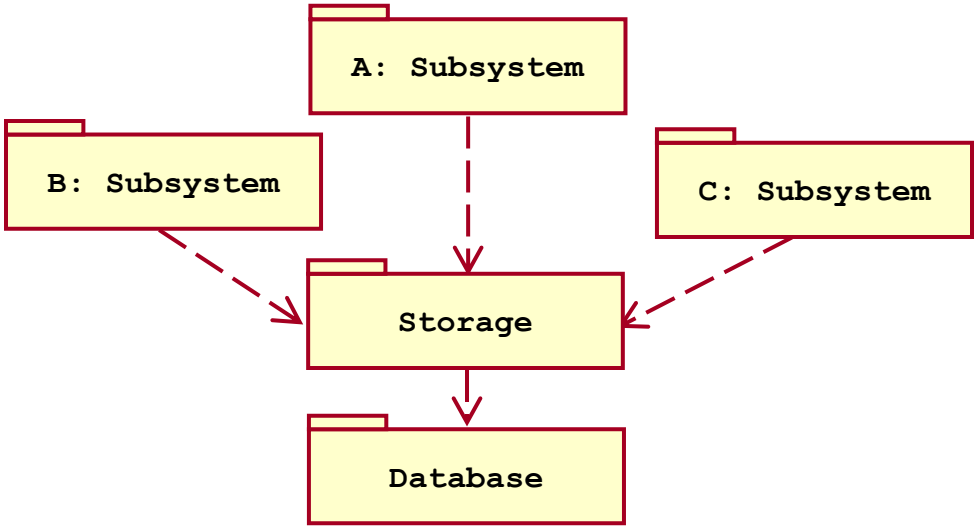
举例：在一个软件系统中，有三个子系统A、B、C都要访问一个关系数据库。在下面的设计方案中，三个子系统直接访问数据库子系统。

这个方案有什么问题？



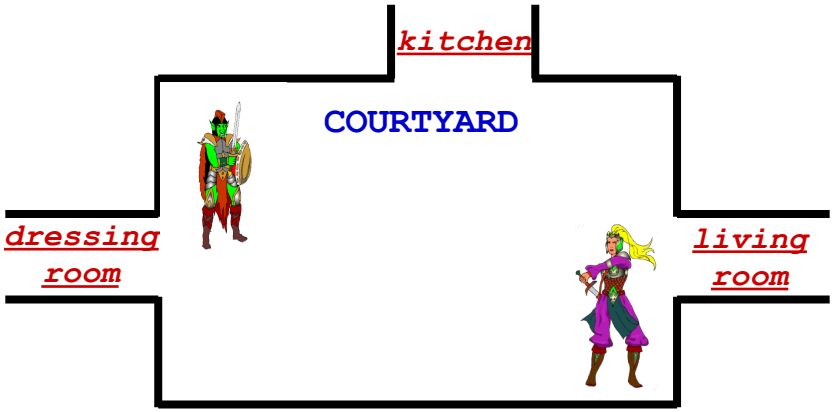
系统分解原则

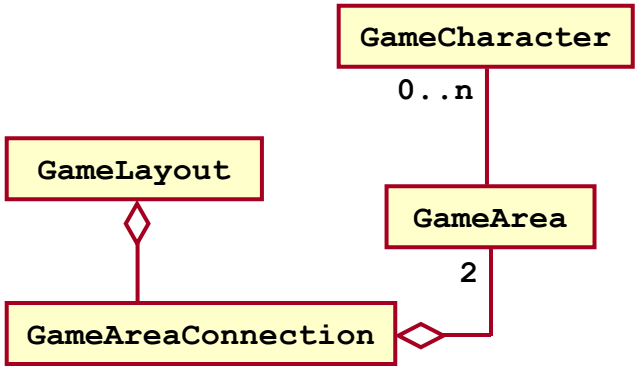
修改：在子系统和数据库之间增加一个存储子系统Storage，从而屏蔽了底层数据库的变化对上层子系统的影响。



系统分解原则

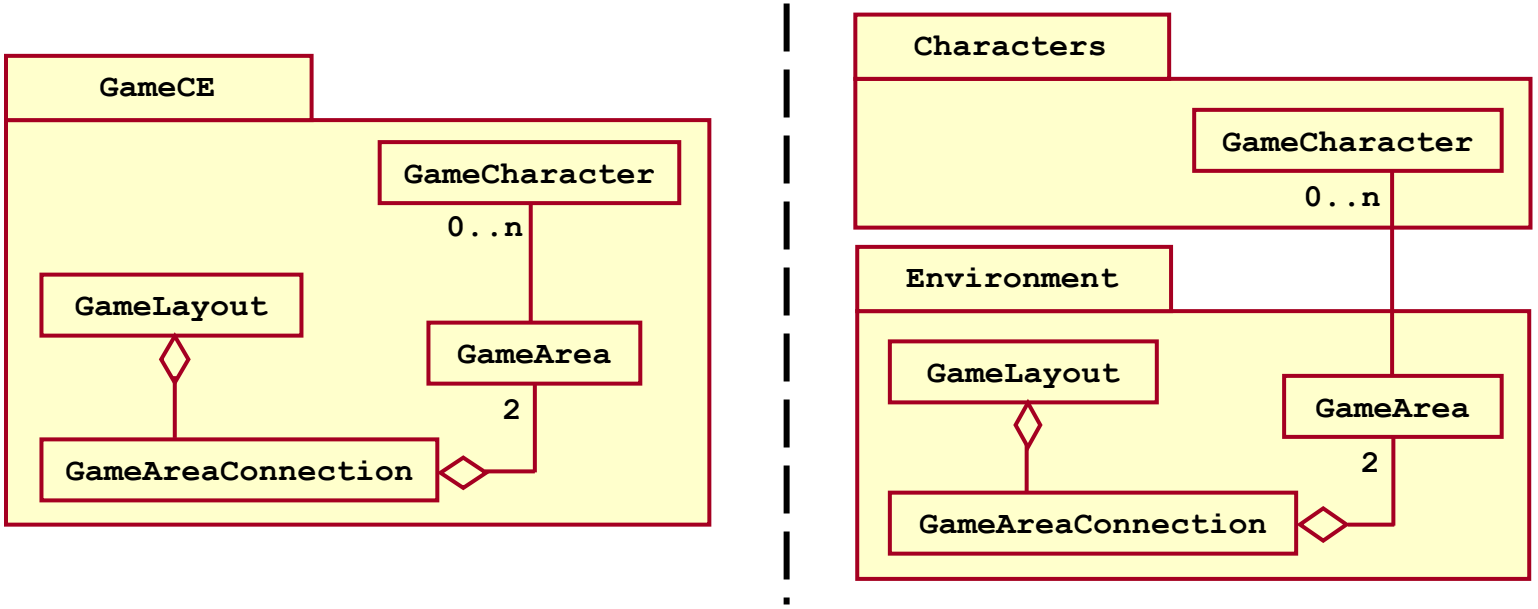
举例：某游戏软件的部分类图如下，其中GameCharacter代表人物，GameLayout代表布局，GameArea 代表一个区域，GameAreaConnection 代表两个区域的连接。





系统分解原则

你认为以下哪个分解方案更好？为什么？



系统分解原则

系统分解的目标：高内聚、低耦合

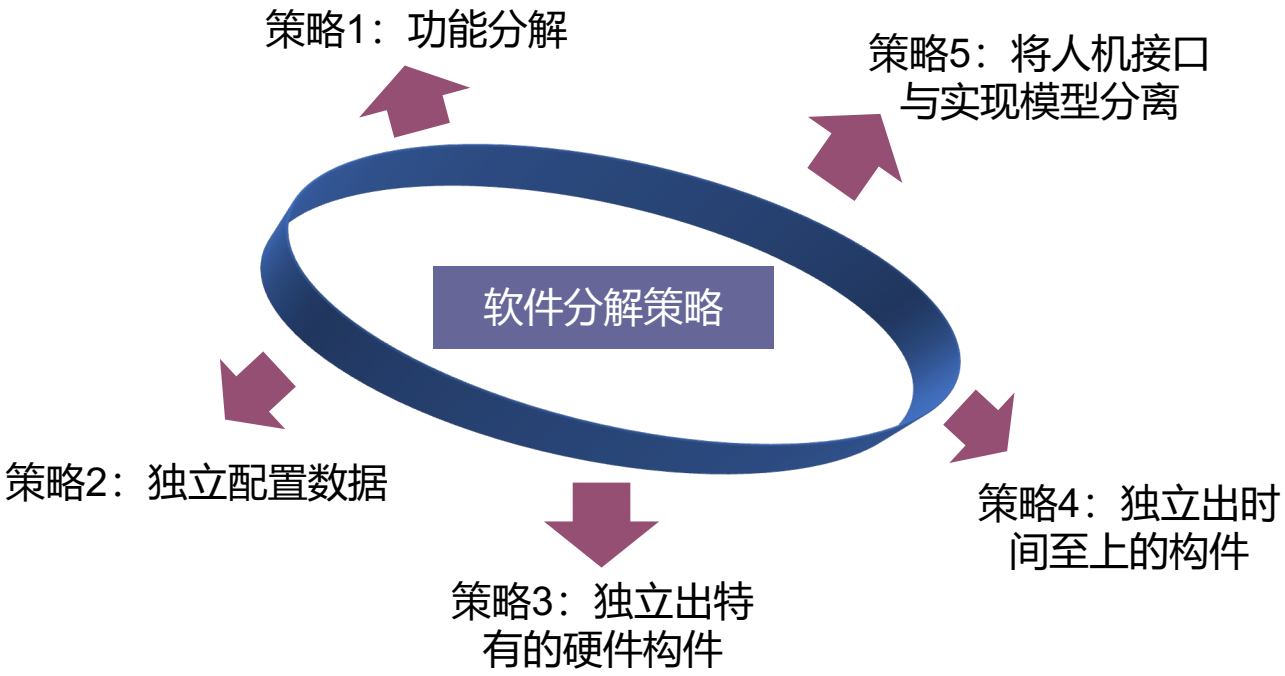
耦合性是两个子系统之间依赖关系的强度。

•如果两个子系统是松散耦合的，二者相互独立，那么当其中一个发生变化时对另一个产生的影响就很小；如果两个子系统是紧密耦合的，其中一个发生变化就可能对另一个产生较大影响。

内聚性是子系统内部的依赖程度。

•如果某个子系统含有许多彼此相关的对象，并且它们执行类似的任务，那么它的内聚性比较高；如果某个子系统含有许多彼此不相关的对象，它的内聚性就比较低。

分解策略


策略1：功能分解
策略5：将人机接口与实现模型分离
软件分解策略
策略2：独立配置数据
策略4：独立出时间至上的构件
策略3：独立出特有的硬件构件

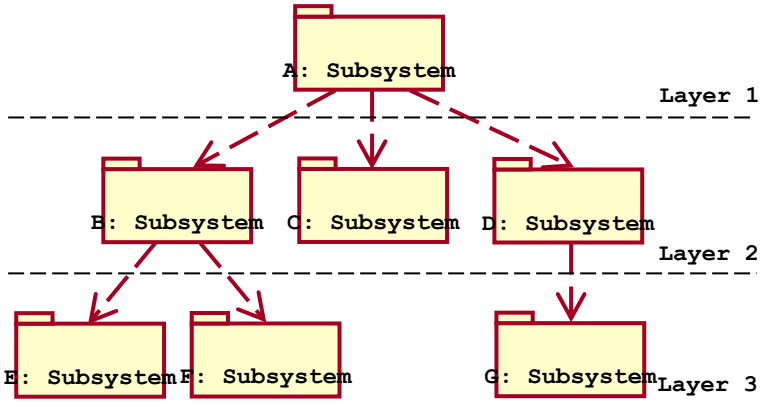
分层和划分

分层（Layering）

- 每一层可以访问下层，不能访问上层
- 封闭式结构：每一层只能访问与其相邻的下一层
- 开放式结构：每一层还可以访问下面更低的层次
- 层次数目不应超过 7±2 层

划分（Partitioning）

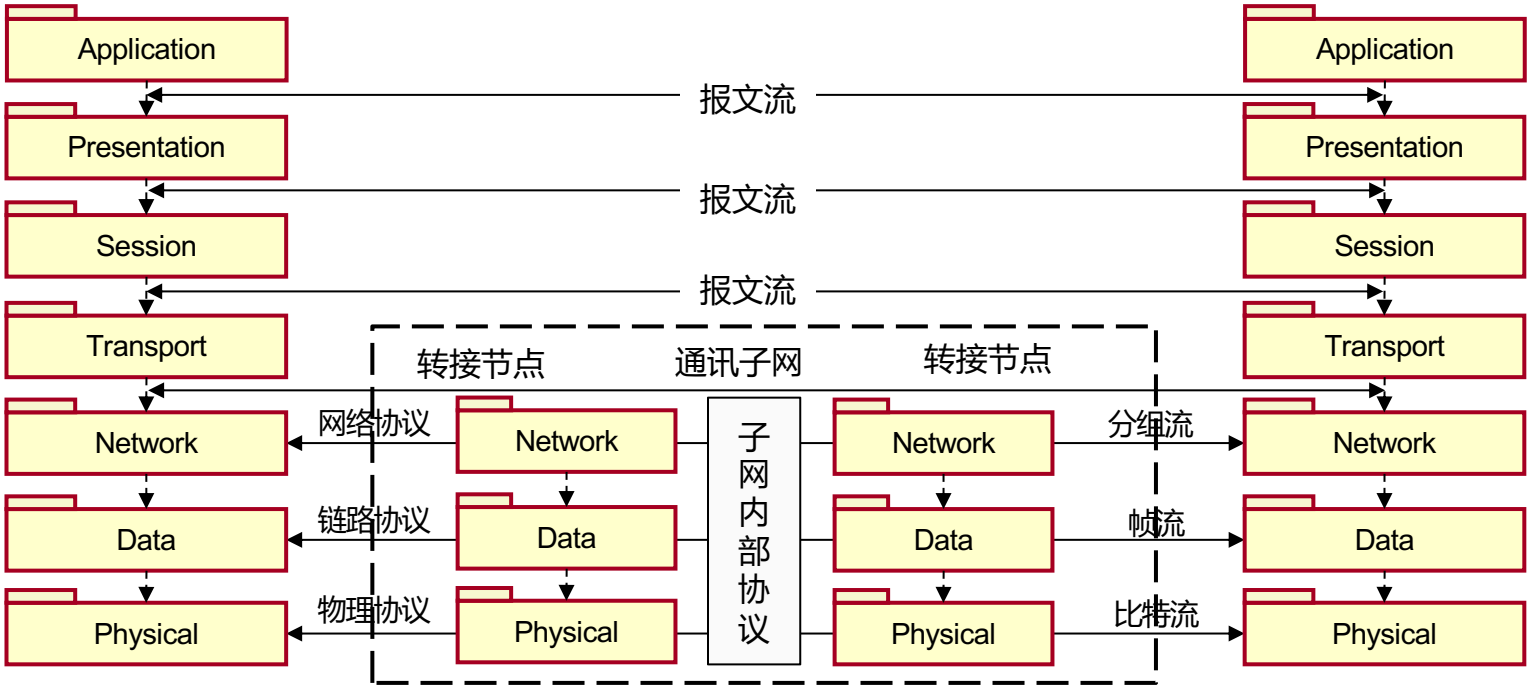
- 系统被分解成相互对等的若干子系统
- 每个子系统之间依赖较少，可以独立运行



注意：子系统增加了处理开销，过度分层或划分会增加复杂性。

分层和划分

示例：网络分层模型



系统设计

系统设计是将分析模型转换到系统设计模型，即定义系统的设计目标，将系统划分成若干子系统，建立整个系统的体系结构，并选择合适的系统设计策略。

确定系统设计目标
确定子系统
定义设计策略
编写系统设计文档
评审系统设计

系统设计目标

性能准则：

- 响应时间：系统响应用户请求的时间
- 吞吐量：在一个固定时间内系统完成的任务量
- 存储量：系统运行需要的存储空间

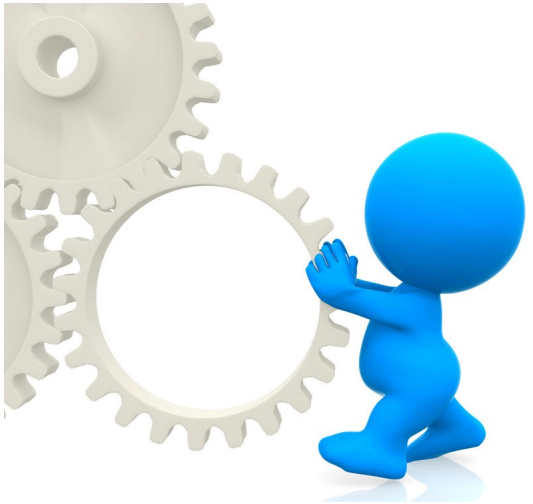
可靠性准则：

- 健壮性：系统承受用户无效输入的能力
- 可靠性：指定操作与所观察行为之间的差别
- 可用性：系统用于完成正常任务的时间
- 容错性：在错误条件下系统的运行能力
- 安全性：系统抵御恶意攻击的能力
- 预防性：在出现错误和故障时系统避免威胁人类生命的能力

系统设计目标

维护准则：

- 可扩展性：增加系统功能或新类的难易程度
- 可修改性：更改系统功能的难易程度
- 适应性：将系统应用到不同应用域的难易程度
- 可移植性：系统移植到不同平台的难易程度
- 可读性：通过阅读代码理解系统的难易程度
- 需求可追踪性：将代码映射到特定需求的难易程度



最终用户准则：

- 效用：系统对用户工作的支持程度
- 易用性：用户使用系统的难易程度

系统设计目标

成本准则：

•开发成本：开发初始系统的成本
•部署成本：安装系统和培训用户的成本
•升级成本：从原有系统导出数据的成本
•维护成本：修复错误和增强系统的成本
•管理成本：对系统进行管理的成本

说明：

•设计目标定义了系统应该重点考虑的质量要求

•性能、可靠性和最终用户准则通常可以从非功能需求或应用领域中推断出来，维护和成本准则需要由用户和开发人员识别。

举例：权衡设计目标

空间与速度：

- 如果响应时间或吞吐量不满足需求，怎么办？
  高速缓存、更多冗余等
- 如果软件不满足存储空间的限制，怎么办？
  以速度为代价压缩数据

交付时间与功能：

•如果开发进度发生落后，可能采取什么策略？

① 按时交付系统，但减少所交付的功能。

② 推迟交付时间，保证最终交付全部功能。

•合同软件通常更强调功能性，成品软件更侧重于交付时间。

举例：权衡设计目标

交付时间与质量：

- 如果测试落后于进度，可能采取什么策略？

① 按时交付系统，但交付的软件带有已知错误，可能随后提供一个补丁程序进行修复。

② 推迟交付时间，努力修改错误，保证交付软件的质量。

交付时间与人员：

- 如果开发进度发生落后，可能采取什么策略？

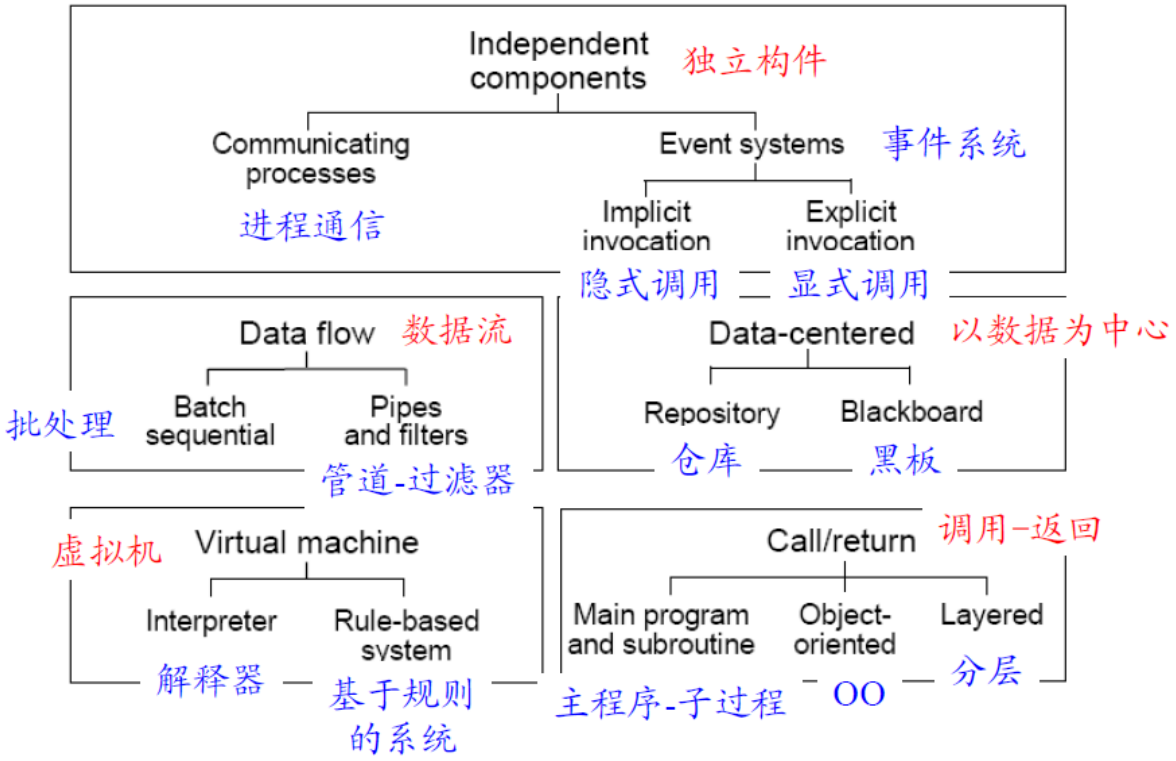
① 在项目早期，可以考虑增加开发人员。

② 尽量激励开发人员加班加点工作。

回顾：软件体系结构风格

体系结构风格
(Architectural Styles)

描述特定系统组织方式的惯用范例，强调软件系统中通用的组织结构



定义设计策略

- 数据文件：数据文件是由操作系统提供的存储形式，应用系统将数据按字节顺序存储，并定义如何以及何时检索数据。
- 关系数据库：在关系数据库中，数据是以表的形式存储在预先定义好的成为Schema 的类型中。
- 面向对象数据库：与关系数据库不同的是，面向对象数据库将对象和关系作为数据一起存储。
- NoSQL数据库：一种不同于关系型数据库的数据库管理系统，强调Key-Value Stores和文档数据库的优点。

定义设计策略

确定访问控制策略：

- 哪些对象在参与者中共享？
- 如何对参与者进行访问控制？
- 系统如何识别参与者的身份？
- 如何对系统中选定的数据进行加密？

识别用户身份：

- 用户名 / 密码：一个用户对应一个用户名和密码的组合，系统在存储和传输密码之前对其进行加密。
- 智能卡：配合密码同时使用。
- 生物特征：指纹、虹膜等。

定义设计策略

设计全局控制流：

- 控制流是系统中动作的先后次序。
- 控制流问题需要在设计阶段考虑，其决策取决于操作者或随时间推移所产生的外部事件。

控制流机制：

- 过程驱动：在需要来自参与者的数据时，操作等待输入。
- 事件驱动：主循环等待外部事件，在外部事件到达时，系统根据与事件相关的信息将其分配给适当的对象。
- 线程：系统创建任意数目的线程，每个线程对应不同的事件。如果某个线程需要额外的数据，就等待参与者的输入。

定义设计策略

识别边界条件：

- 系统何时启动、初始化、关闭?
- 如何处理主要故障（如软件错误、断电、断网等）？

边界用例：

- 系统管理：对于不在普通用例中创建或销毁的对象，增加一个由系统管理员调用的用例进行管理。
- 启动与关闭：启动、关闭和配置构件。
- 异常处理：通过对需求获取中识别的一般用例进行扩展而得到，需要考虑用户错误、硬件故障、软件故障等因素。

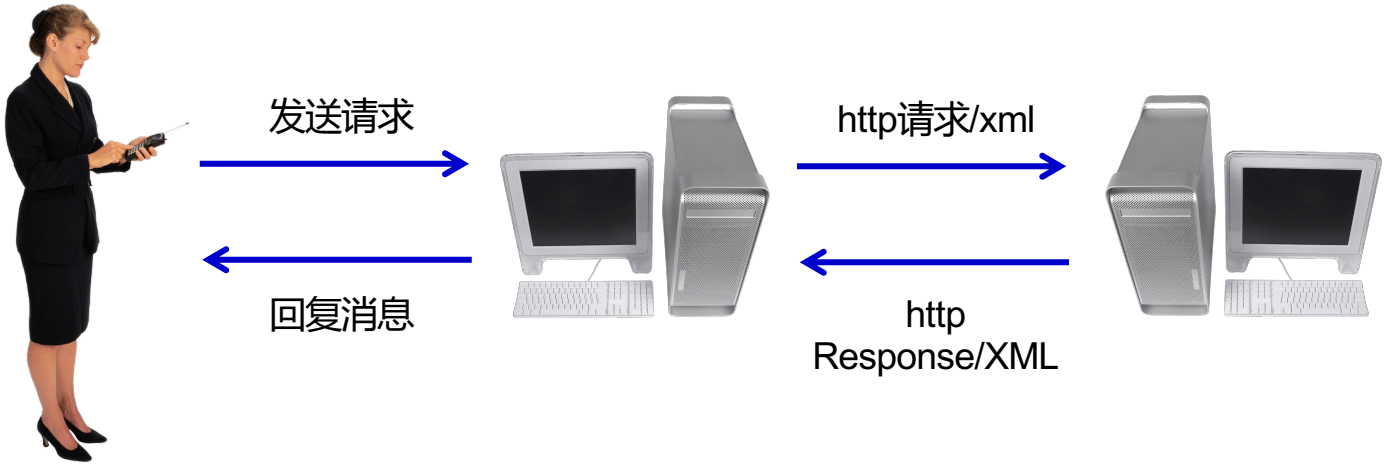
案例：HIT成绩助手

HIT成绩助手是由哈尔滨工业大学学生开发的一个微信公众号，它通过微信渠道提供各类校园信息服务，可以快捷方便地查询成绩、课程表、考试倒计时、饭卡对账单、图书搜索和借阅等。

目前由于学校教务处干涉的原因，该账号已经暂停使用。

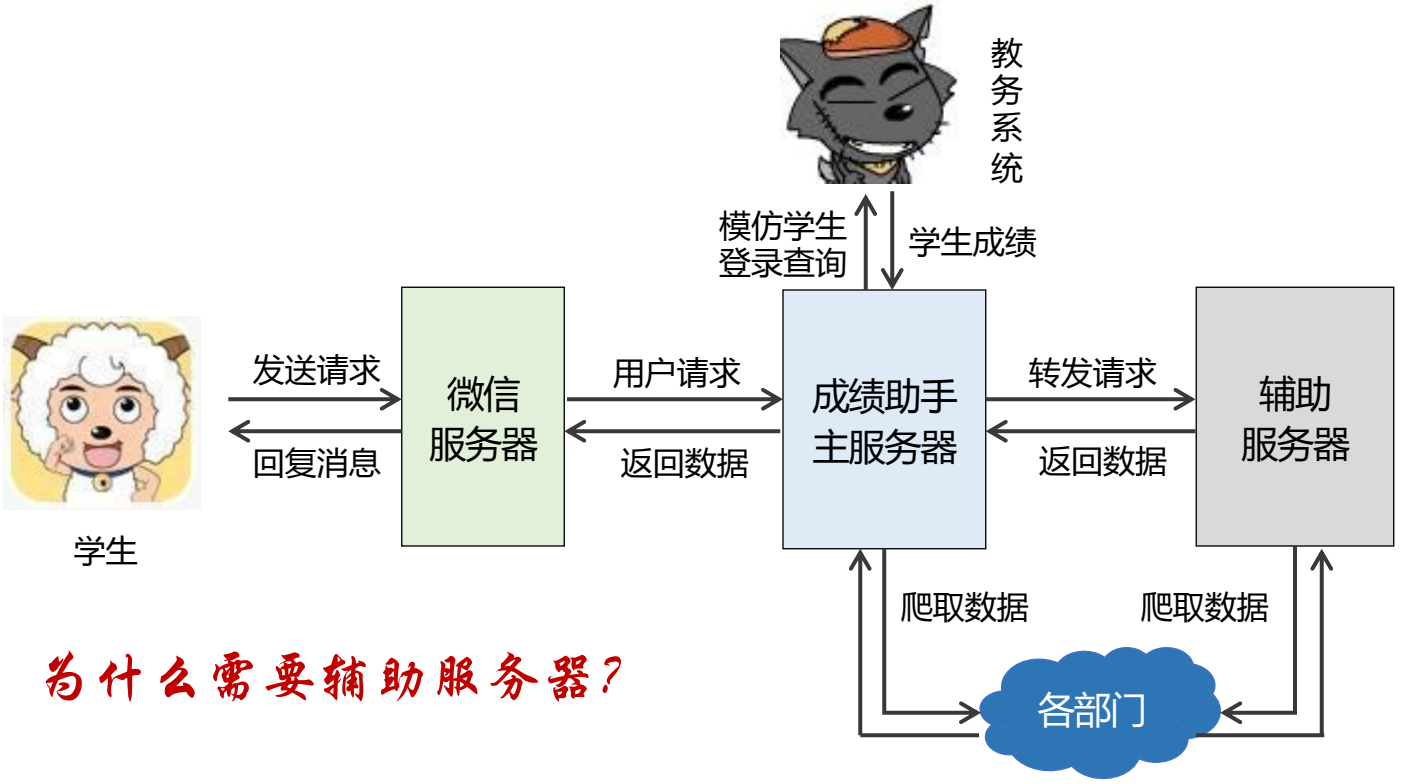
案例：HIT成绩助手

微信平台的开发模式原理



当用户向公众号发送信息时，这个信息会发到微信服务器进行处理，之后微信再将响应的信息发到开发者服务器；开发者要在5秒内响应并发回给微信服务器，最后由微信服务器发送信息返回给用户。

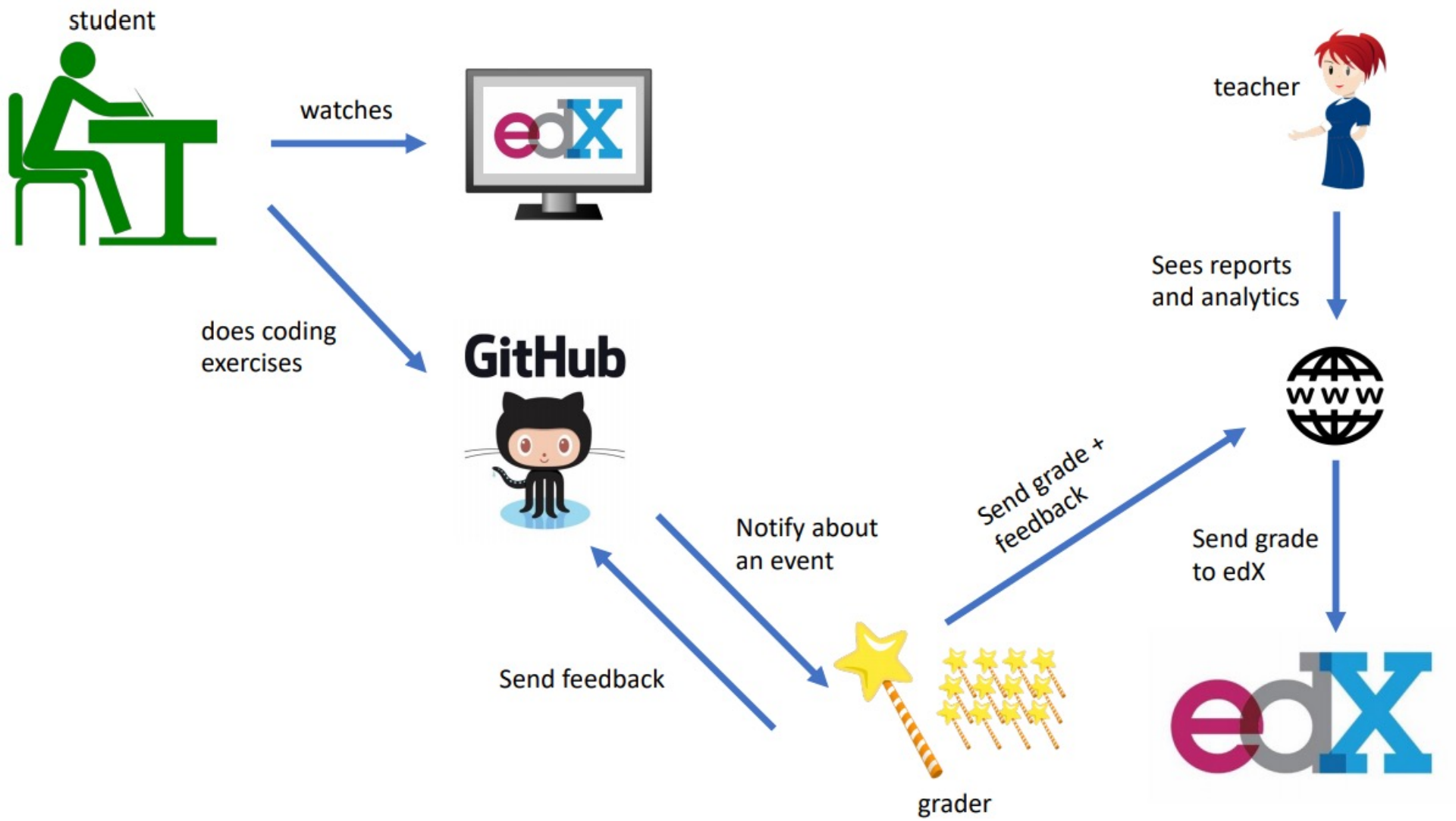
案例：HIT成绩助手

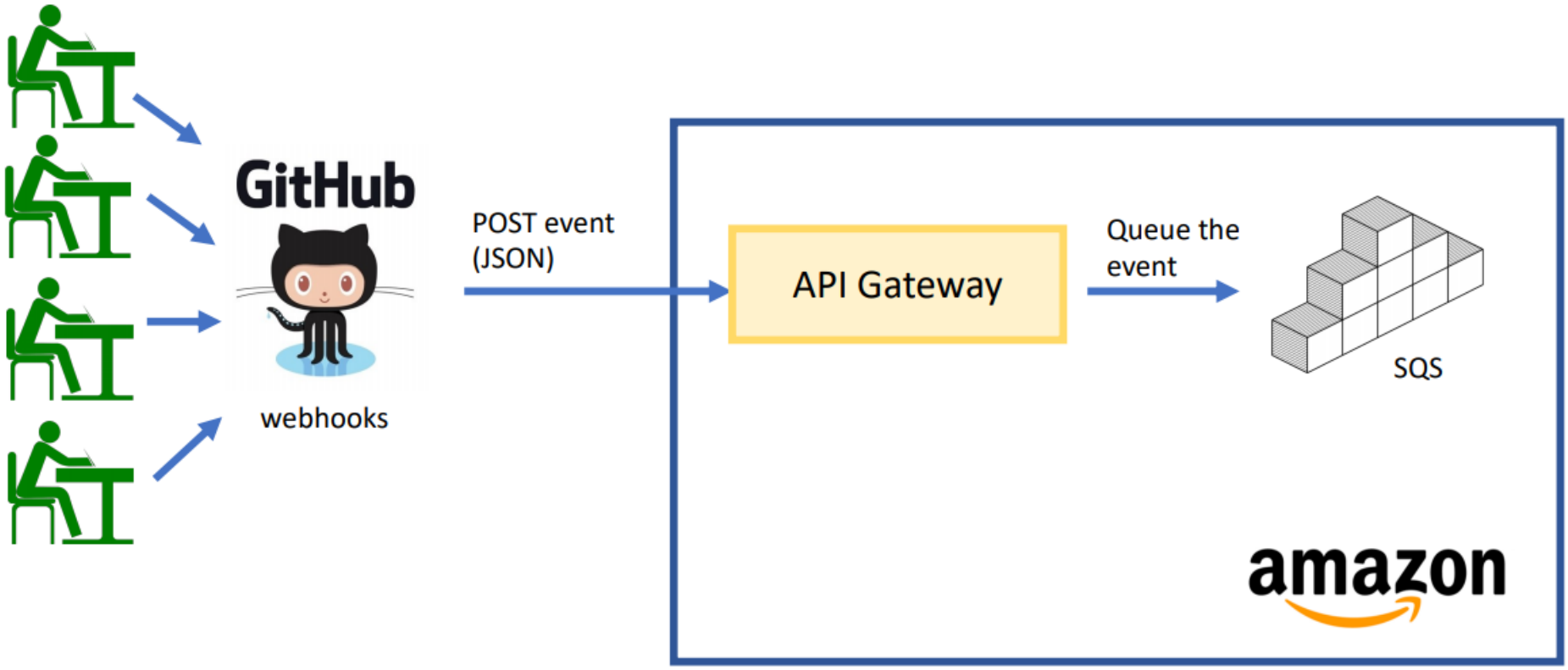


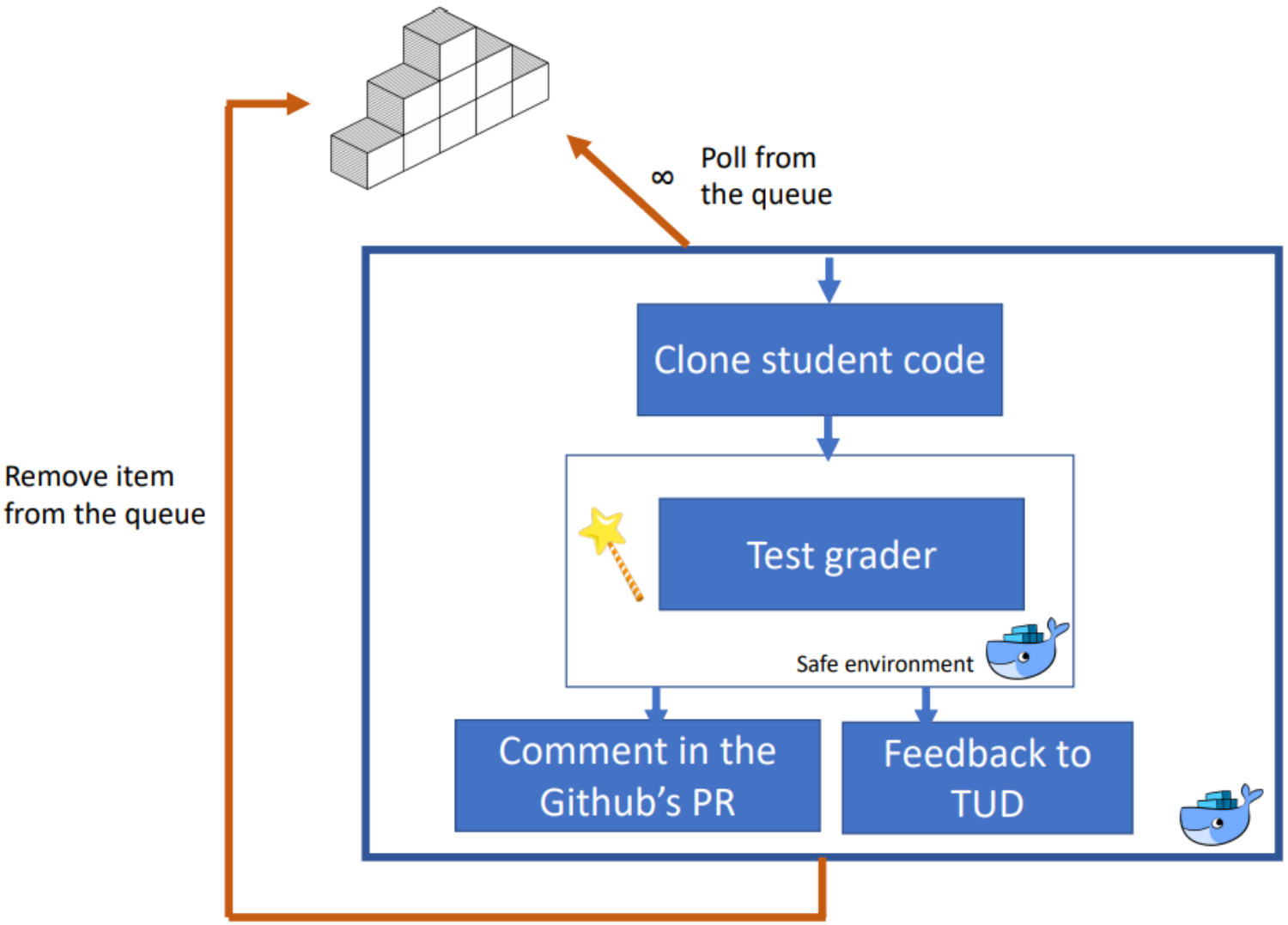
为什么需要辅助服务器？

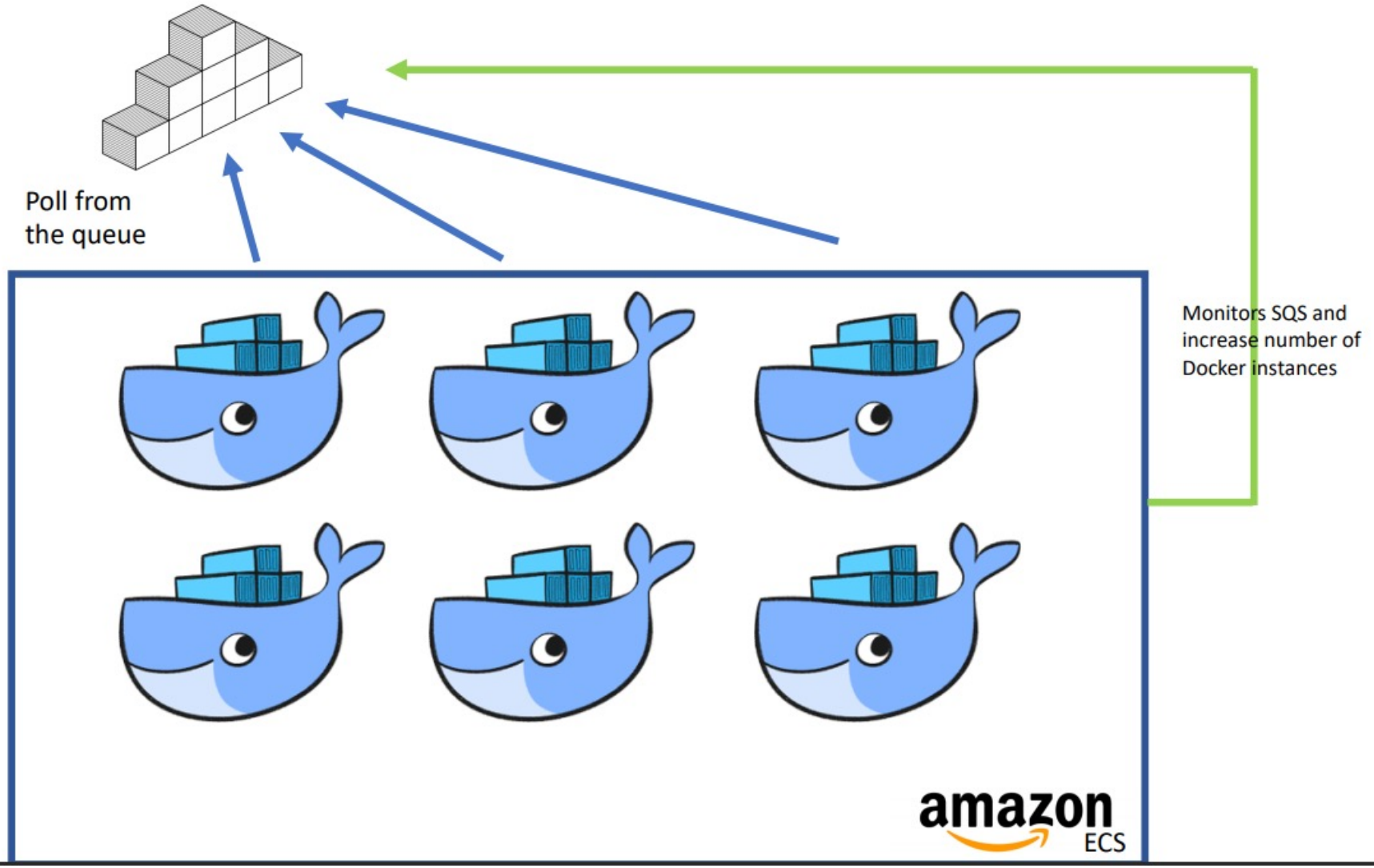
案例: TUDelft edX Grading

- Students will watch videos and see exercise descriptions at edX.
- Students will have different coding exercises to solve. Each exercise has a different way to be automatically graded / feedback.
  - We should provide students with feedback about what they did wrong in the exercise.
  - We should send the results (grade) of a graded assignment back to edX.
- Students will use GitHub/GitLab to store their code.
- Final grades + feedback should also be stored at TU Delft.
- We expect 10-20k students.
  - Remember that students tend to submit assignments near the deadline.


student
watches
edX
teacher
does coding
exercises
GitHub
Sees reports
and analytics
www
Notify about
an event
Send grade +
feedback
Send grade
to edX
Send feedback
grader
edX


GitHub
webhooks
POST event
(JSON)
API Gateway
Queue the
event
SQS
amazon


∞
Poll from
the queue
Remove item
from the queue
Clone student code
Test grader
Safe environment
Comment in the
Github's PR
Feedback to
TUD

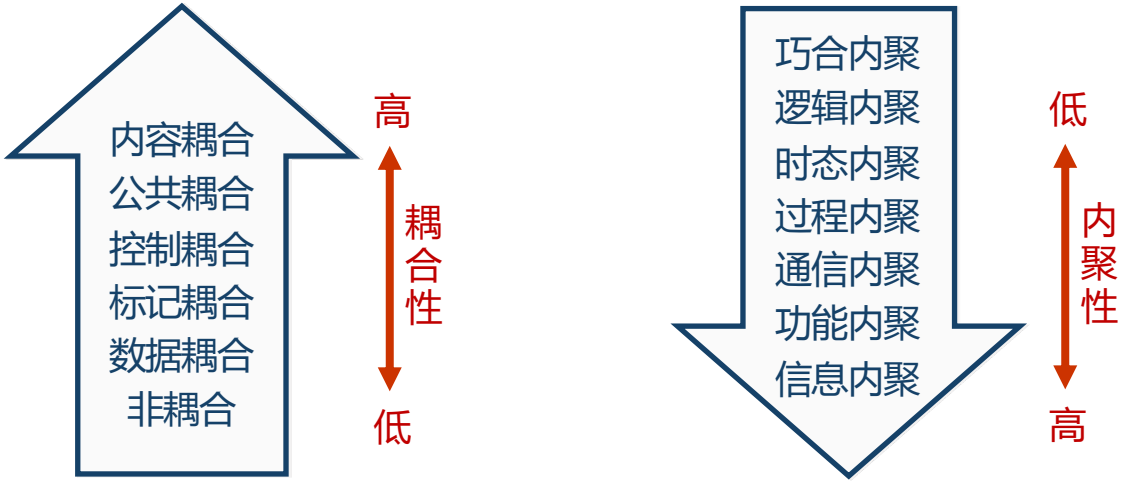

Poll from
the queue
Monitors SQS and
increase number of
Docker instances
amazon
ECS

软件设计原则

设计原则是系统分解和模块设计的基本标准，应用这些原则可以使代码更加灵活、易于维护和扩展。

软件设计原则

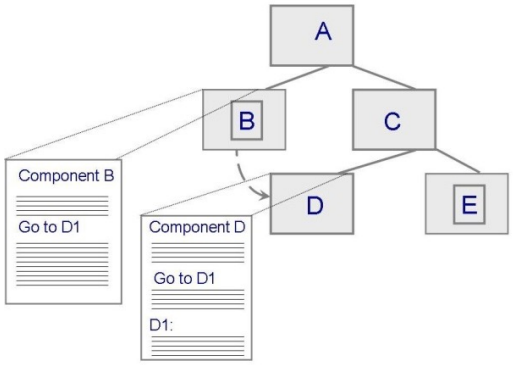
模块化（也称关注点分离）是将系统中各不相关的部分进行分离的原则，以便于各部分能够独立研究。

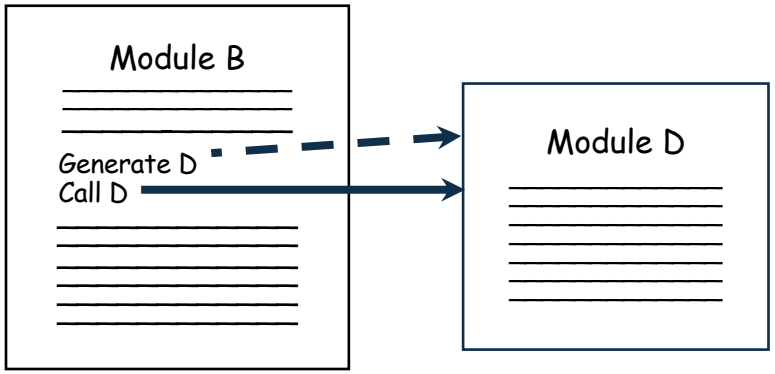


软件设计原则

内容耦合：

- 一个模块修改了另一个模块的内部数据项
- 一个模块修改了另一个模块的代码
- 一个模块内的分支转移到另一个模块中





软件设计原则

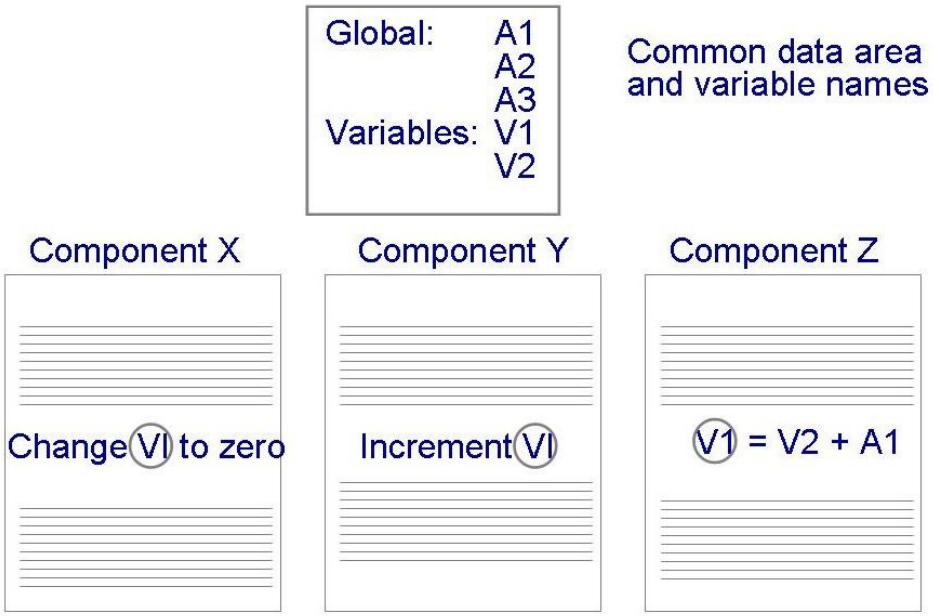
```
public class Line
{
    private point start, end;
    ...
    public point getStart() { return start; }
    public point getEnd() { return end; }
}

public class Arch
{
    private Line baseline;
    ...
    void slant(int newY)
    {
        point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getX(), newY);
    }
}
```

内容耦合

软件设计原则

公共耦合：所有模块通过从公共数据存储区访问数据而产生耦合。



软件设计原则

控制耦合：某个模块通过传递参数或返回代码来控制另一个模块的活动。

```
public routineX(String command)
{
    if (command.equals("drawCircle")
    {
        drawCircle();
    }
    else
    {
        drawRectangle();
    }
}
```

标记耦合：使用一个复杂的数据结构从一个模块向另一个模块传送参数，并且传送的是该结构本身。

数据耦合：两个模块传递参数时如果只是数据值，而不是结构数据。

软件设计原则

标记耦合

```
public class Emailer
{
    public void sendEmail(Employee e, String txt) {...}
    ...
}
```

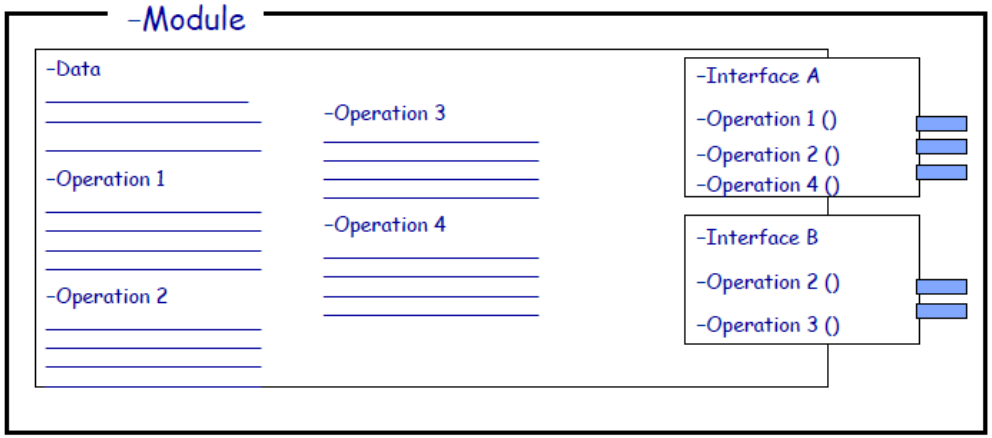
数据耦合

```
public class Emailer
{
    public void sendEmail(String name, String email, String txt) {...}
    ...
}
```

如果方法只带一个参数，那么数据耦合是松散的，选择数据耦合比标记耦合更好。如果用许多简单类型的参数（数据耦合）代替一个复杂的参数（标记耦合），总的耦合会高。

软件设计原则

接口定义了一个软件单元的服务，并对外封装了单元的设计和实现的细节。



要求：应将软件单元接口设计得尽可能简单，并将单元对于环境的假设和要求降至最低。

软件设计原则

```
public class Emailer
{
    public void sendEmail(Employee e, String txt) {...}
    ...
}
```

```
public interface Addressee
{
    public abstract String getName();
    public abstract String getEmail();
}
public class Employee implements Addressee {...}
public class Emailer
{
    public void sendEmail(Addressee e, String txt) {...}
    ...
}
```

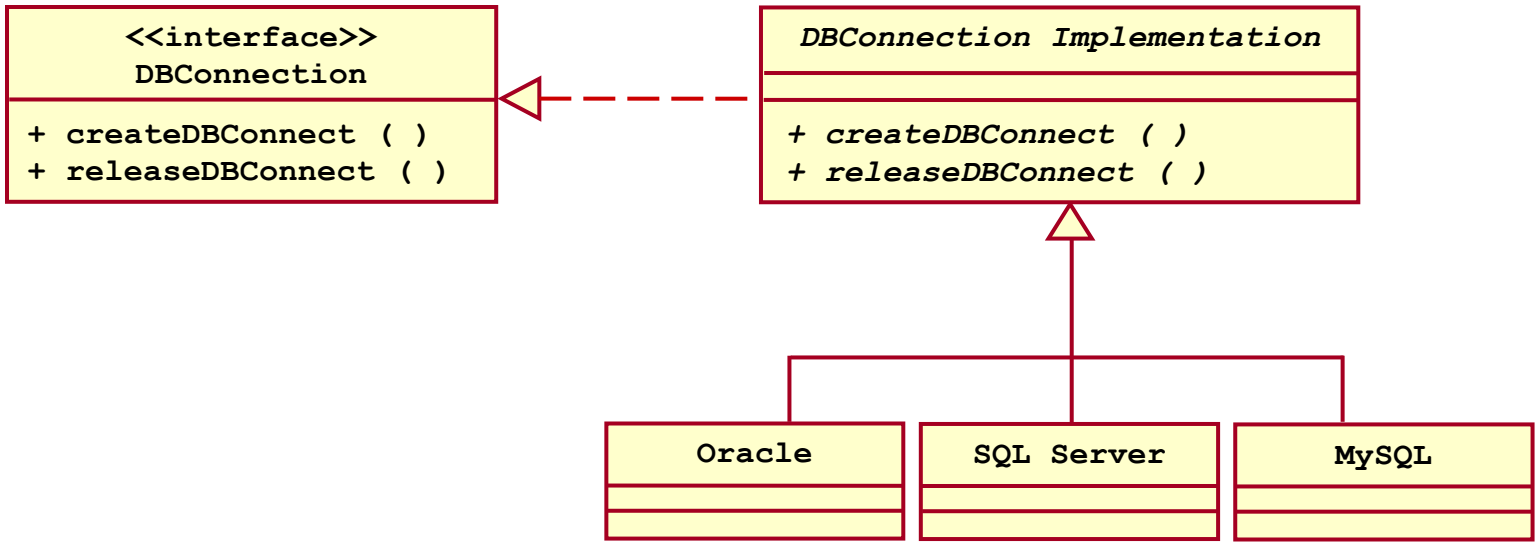
软件设计原则

信息隐藏以系统分解为特征，每个软件单元对其他所有单元都隐藏自己的设计决策，各个单元的性质通过其外部可见的接口来描述。

- 假设Employee类定义公司员工的基本信息，PaySlip类负责计算和打印员工每月的工资单。那么，PaySlip对象应该通过什么方法获得Employee对象的工资水平?
- 实现方法A时，实际上只需要类B的两个属性，但却把整个类B定义成了方法A的参数。
- 大多数情况下类的方法并不会把成员变量作为自身的参数，而是会直接使用成员变量。

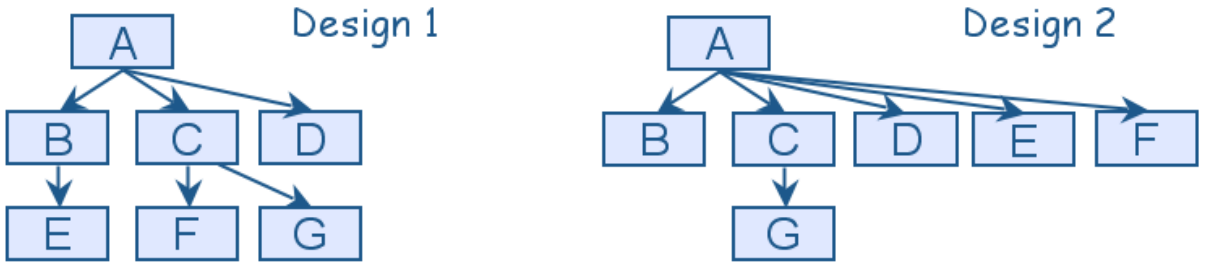
软件设计原则

抽象是一种忽略一些细节来关注其他细节的模型或表示。



软件设计原则

增量式开发：在确定软件单元及其接口之后，可以根据单元之间的依赖关系进行增量式的开发。从满足增量开发的考虑，创建一个具有**高扇入和低扇出**的软件单元。



高扇出意味着单元的职责过多，也许可以进一步分解为更小更简单的单元。在上图中，设计1比设计2更合理。

e.g.

低扇入和低扇出 Standalone class

高扇入和低扇出 Low-level class

低扇入和高扇出 High level class

高扇入和高扇出 Problematic!

软件设计原则

复用（Reuse）是利用某些已开发的、对建立新系统有用的软件元素来生成新的软件系统。复用的好处在于提高生产效率，提高软件质量，改善软件系统的可维护性。

- 类库：强调在类级别的代码复用，如MFC、JDK等
- 框架：强调构件级的设计复用，如CORBA、J2EE、.NET等
- 设计模式：通过为对象协作提供思想和范例来强调方法的复用，它们是目前所知的良好开发实践。

面向对象设计原则

> 一个好的系统设计应该具有如下性质：
>
> 可扩展性、灵活性、可插入性。
>
> —— Peter Code，1999

- 可扩展性：容易添加新的功能
- 灵活性：代码修改能够平稳地发生
- 可插入性：容易将一个类抽出去，同时将另一个具有同样接口的类加进来

面向对象设计原则

- 单一职责原则
- 开放封闭原则
- Liskov替换原则
- 接口分离原则
- 依赖倒置原则

单一职责原则

单一职责原则（Single Responsibility Principle，SRP）

> ***There should never be more than one reason for a class to change.***
>
> **R. Martin, 1996**

说明：

- 让一个类有且只有一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。
- 该原则可以看作是高内聚、低耦合在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

单一职责原则

举例：下面的接口设计有问题吗？

```
interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(Char c);
    public char recv();
}
```

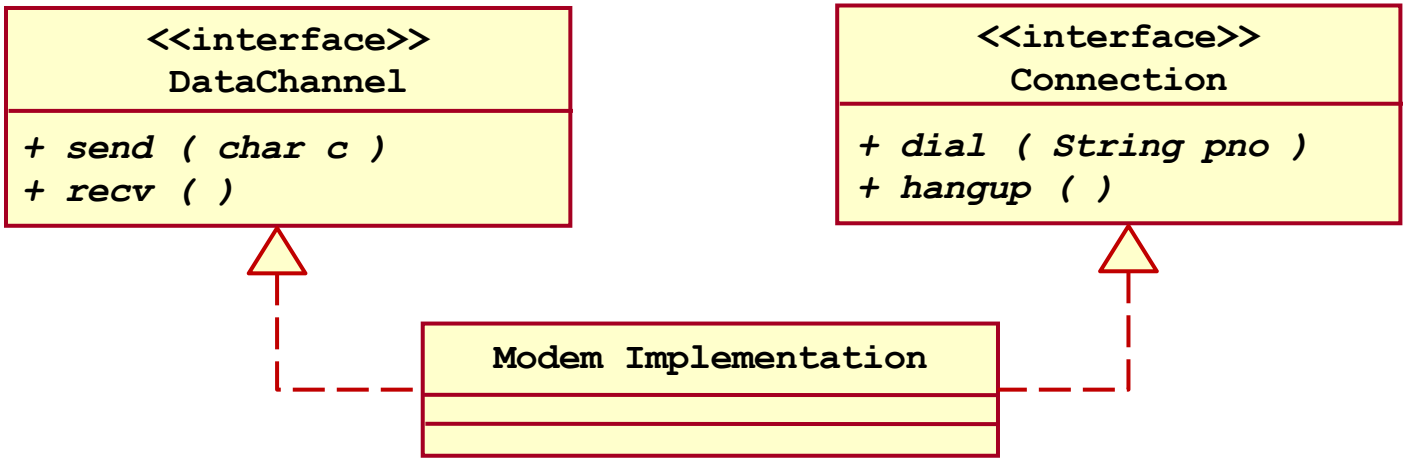
接口职责：

- 连接管理：dial 和 hangup 进行调制解调器的连接处理
- 数据通信：send 和 recv 进行数据通信

单一职责原则

Modem接口是否应该分离?

- 依赖于应用程序变化的方式，它是否影响连接函数的特征?
- 数据通信是经常变化的；连接管理是相对稳定的



单一职责原则

总结：

- 职责的划分：一个类的一个职责是引起该类变化的一个原因
- 单一职责原则分离了变与不变
  - ✓ 对象的细粒度：便于复用
  - ✓ 对象的单一性：利于稳定
- 不要创建功能齐全的类

单一职责原则是一个最简单的原则，也是最难正确应用的一个原则，因为我们会自然地将职责结合在一起。

Example

```
class AgeCalculator {
    private val currentYear = Calendar.getInstance().get(Calendar.YEAR)

    fun calculate(birthYear: Int): String {
        return (currentYear - birthYear).toString()
    }

    fun isValid(birthYear: Int): Boolean {
        return currentYear > birthYear
    }
}
```

Correction

```
class AgeCalculator {
    fun calculate(birthYear: Int): String {
        val currentYear = Calendar.getInstance().get(Calendar.YEAR)
        return (currentYear - birthYear).toString()
    }
}
class AgeValidator {
    fun isValid(birthYear: Int): Boolean {
        val currentYear = Calendar.getInstance().get(Calendar.YEAR)
        return currentYear > birthYear
    }
}
```

开放封闭原则

开放封闭原则（Open-Closed Principle，OCP）

> ***Software entities should be open for extension, but closed for modification.***
>
> **B. Meyer, 1988 / quoted by R. Martin, 1996**

- Be open for extension

  模块的行为是可扩展的，即可以改变模块的功能。

- Be closed for modification

  对模块行为进行扩展时，不需要改动源代码或二进制代码。

- 问题：怎样可以在不改动源码的情况下改变模块行为？

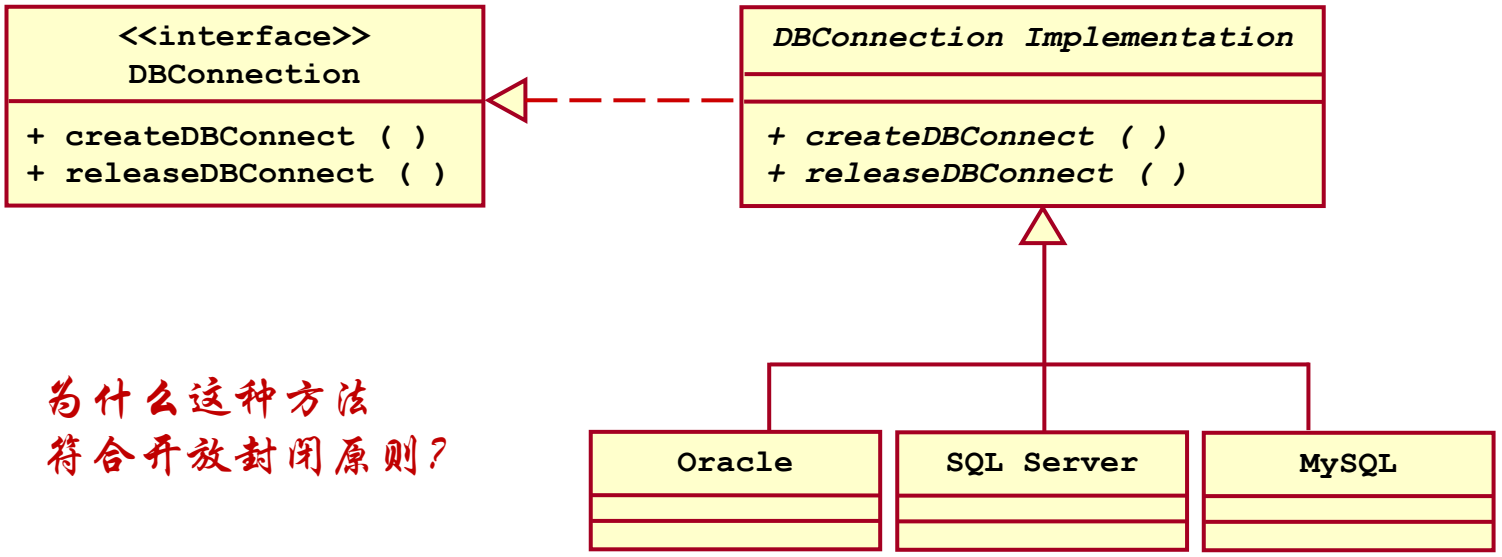
开放封闭原则

举例：如要增加新的数据库类型（如MySQL）怎么处理?

```
bool createDBConnect(...)
{
    ...
    switch (dbType)
        {
         case ORACLE:
            建立 Oracle 连接;
                break;
         case SQLSERVER:
            建立 SQL Server 连接;
                break;
    }
    ...
}
```

开放封闭原则

数据库连接的另一种修改方法：



为什么这种方法
符合开放封闭原则？

开放封闭原则

开放封闭原则：在设计一个软件模块（类、方法）时，应该在不修改原有代码（修改封闭）的基础上扩展其功能（扩张开放）。

实现方式

- 将那些不变的部分加以抽象成不变的接口，这些不变的接口可以应对未来的扩展；
- 接口的最小功能设计：原有的接口要么可以应对未来的扩展，不足的部分可以通过定义新的接口来实现；
- 模块之间的调用通过抽象接口来实现，即使实现层发生变化也无需修改调用方的代码。

开放封闭原则

关键是抽象

- 创建一个抽象基类，描述一组任意个可能行为，而具体的行为由派生类实现。
- 软件实体依赖于一个固定的抽象基类，这样它对于更改可以是关闭的，通过从这个抽象基类派生，也可以扩展该实体的行为。

程序不可能是100%封闭的

- 无论模块多么封闭，都会存在一些无法封闭的变化。
- 设计人员应该策略地对待这个问题，即先预测出最有可能发生变化的部分，再构造抽象来隔离这些变化。

开放封闭原则

- **开放封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术的巨大好处，即可维护性、可扩展性、可复用性和灵活性。**

- **几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到开放封闭原则的要求。**

- 开发人员**应该仅对程序中呈现出频繁变化的那些部分做出抽象**，然而切忌对应用程序中的每个部分都刻意地进行抽象，拒绝不成熟的抽象和抽象本身一样重要。

Example

- enum class ShoesBrand {
- NIKE, ADIDAS
- }
- class ShoesFactory {
- fun getShoesPrice(shoesBrand: ShoesBrand): Int {
- return which (shoesBrand) {
- case: ShoesBrand.NIKE -> 200
- case: ShoesBrand.ADIDAS -> 150
- }
- }
- }
- ShoesFactory shoes
  shoes.getShoesPrice(ShoesBrand.NIKE).toString()

Correction

- abstract class ShoesBrand {
- abstract fun getShoesPrice(): Int
- }
- class NikeShoes : ShoesBrand() {
- override fun getShoesPrice() = 200
- }
- class AdidasShoes : ShoesBrand() {
- override fun getShoesPrice() = 150
- }
- ShoesBrand shoes = NikeShoes()
- shoes.getShoesPrice().toString()

Liskov替换原则

Liskov替换原则（Liskov Substitution Principle，LSP）

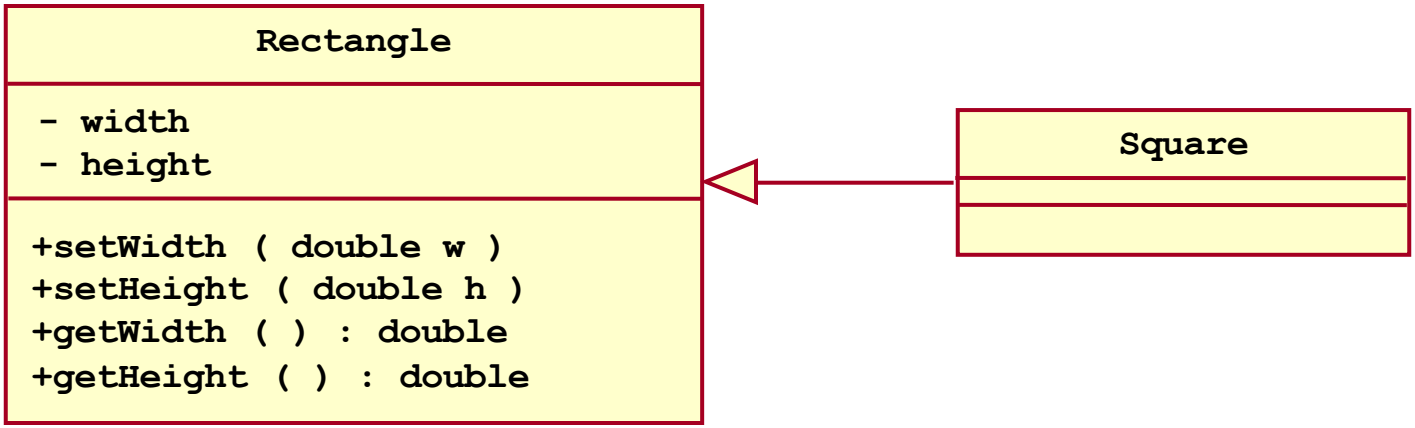
> ***Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.***
>
> **B. Liskov, 1987**

> ***Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.***
>
> **R. Martin, 1996**

理解：子类型必须能够完全替换其父类型。

Liskov替换原则

举例：正方形应该从矩形继承吗？



你是否编写过类似的程序？有什么问题吗？

Liskov替换原则

覆写 Square 类成员函数

```
void Square::setWidth(double w)
{
    Rectangle::setWidth(w);
    Rectangle::setHeight(w);
}

void Square::setHeight(double h)
{
    Rectangle::setWidth(h);
    Rectangle::setHeight(h);
}
```

考虑下面的测试函数：

```
void g(Rectangle& r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert(r.Area() == 20);
}
```

为什么不对？　　原因：从行为方式来看，Square 不是 Rectangle。

Liskov替换原则

契约式设计（Design by Contract）

- 契约式设计把类及其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。
- 契约是通过每个方法声明的前置条件（preconditions）和后置条件（postconditions）来指定的。
- 一个方法得以执行，其前置条件必须为真；执行完毕后，该方法应保证后置条件为真。
- 规则：**在重新声明派生类函数时，只能使用相等或者更弱的前置条件来替换原始前置条件，只能使用相等或者更强的后置条件来替换原始后置条件。**

Liskov替换原则

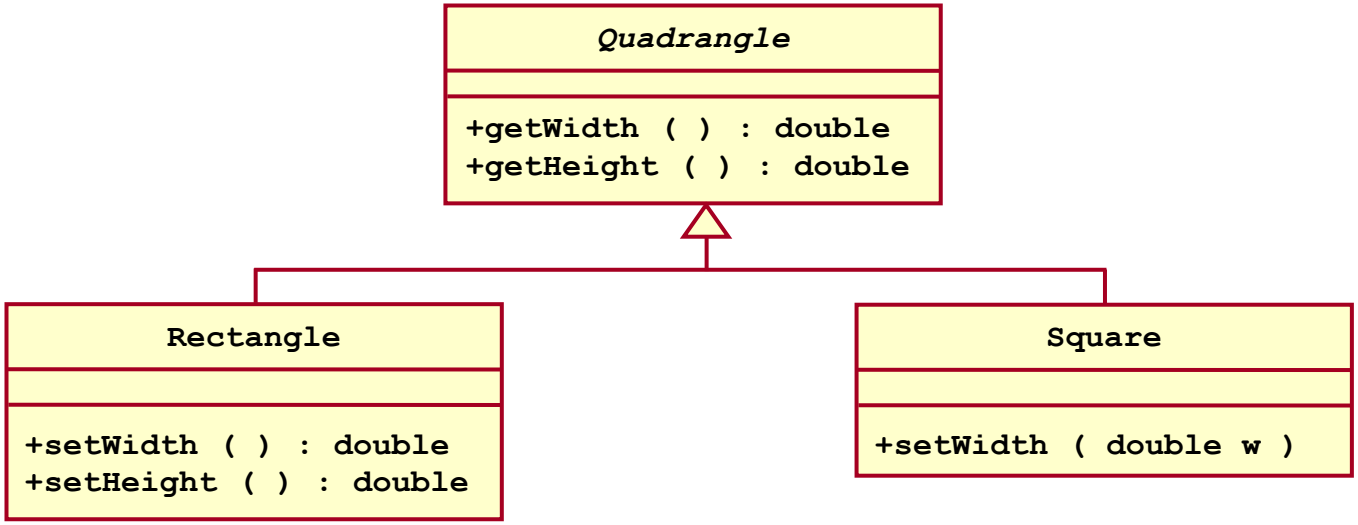
Square符合契约式设计规则吗?

- `Rectangle::setWidth(double w)`的后置条件
  `assert((width == w) && (height == oldHeight))`
- `Square::setWidth(double w)`的后置条件
  `assert((width == w) && (height == w))`
- 由此可见，`Square::setWidth(double w)`的后置条件不能遵从`Rectangle::setWidth(double w)`后置条件的所有约束，故前者的后置条件比后者的弱。

如何解决以上问题?

Liskov替换原则

重构方法：创建一个新的抽象类C，将其作为两个具体类的超类，将A、B的共同行为移动到C中来解决问题。



Liskov替换原则

- LSP可以保证系统或子系统有良好的扩展性。只有子类能够完全替换父类，才能保证系统或子系统在运行期内识别子类接口，因而使得系统或子系统具有良好的扩展性。

- LSP有利于实现契约式编程。契约式编程有利于系统的分析和设计，即定义好系统的接口，然后在编码时实现这些接口即可。在父类里定义好子类需要实现的功能，而子类只要实现这些功能即可。

- LSP是保证OCP的重要原则。它们在实现方法上有个共同点，使用中间接口层来达到类对象的低耦合，即抽象耦合。

接口分离原则

接口分离原则（Interface Segregation Principle，ISP）

> ***Clients should not be forced to depend upon interfaces that they do not use.***
>
> **R. Martin, 1996**

- 胖类会导致其客户程序之间产生不正常且有害的耦合关系。因此，客户程序应仅依赖于它们实际调用的方法。
- 将胖类的接口分解为多个特定于客户程序的接口，每个特定的接口仅声明其特定客户程序调用的那些函数，胖类则继承所有特定于客户程序的接口并实现之。

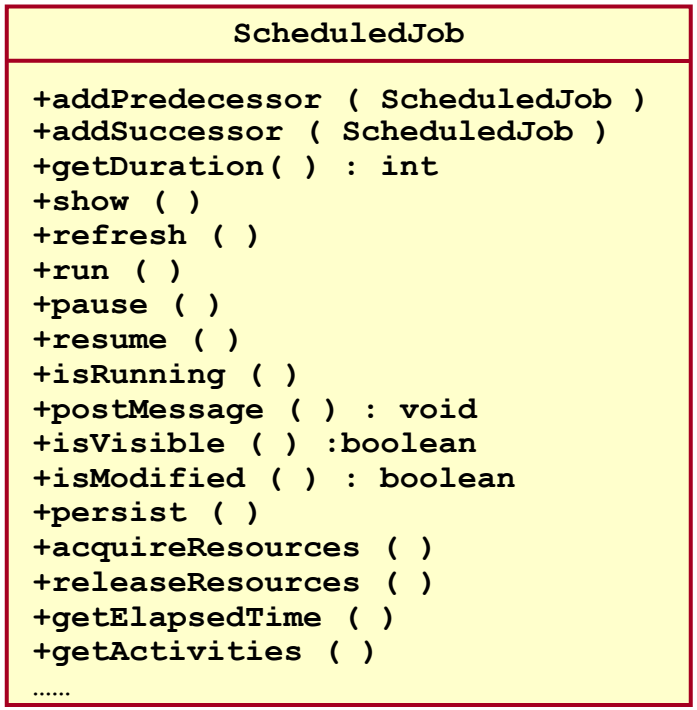
接口分离原则

举例：一个庞大的类

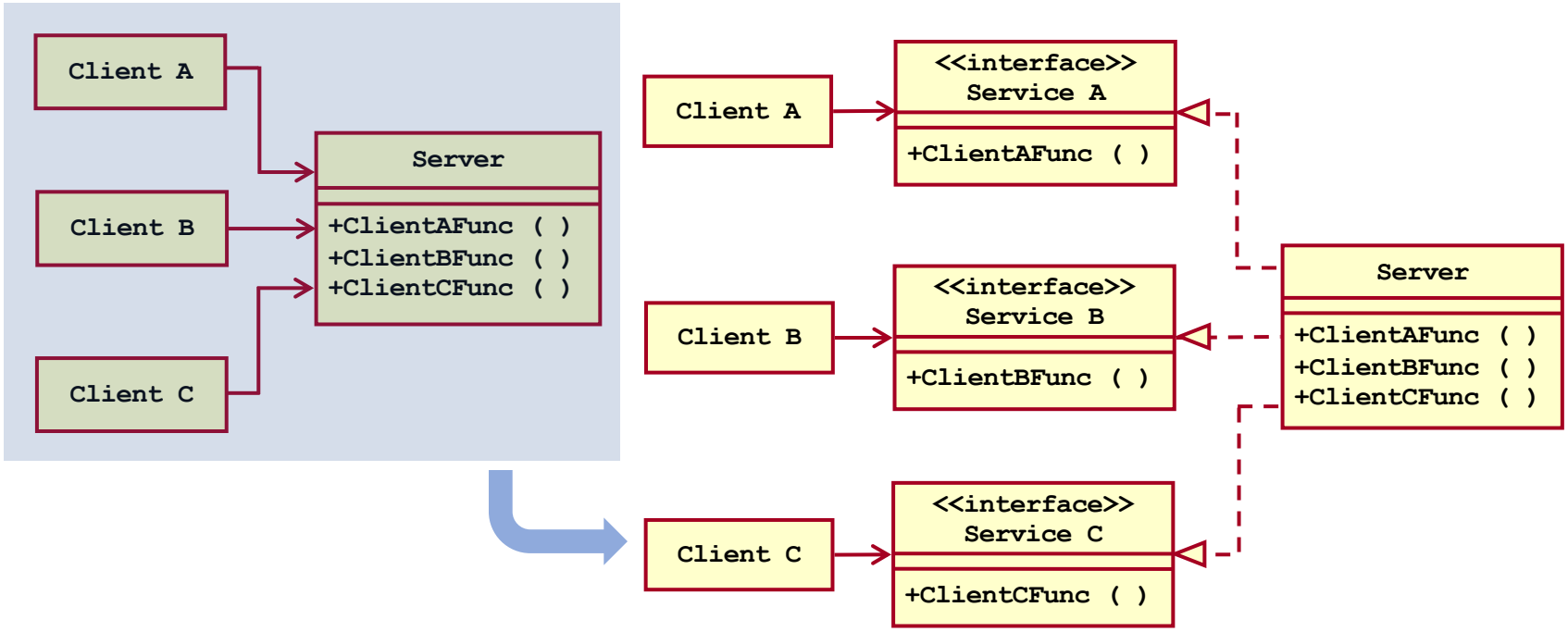
存在问题：

- 弄清楚需要修改什么以及修改会影响到什么是很困难的
- 可能会遇到多个开发人员同时进行不同修改的情况，这样会导致任务调度冲突
- 测试如此庞大的类是非常痛苦的

如何处理这种庞大的类?



接口分离原则


Client A
Client B
Client C
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )
Client A
<<interface>>
Service A
+ClientAFunc ( )
Client B
<<interface>>
Service B
+ClientBFunc ( )
Client C
<<interface>>
Service C
+ClientCFunc ( )
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )

依赖倒置原则

常见的设计问题

- 很难添加新功能，因为每一处改动都会影响其他很多部分。
- 当对某一处进行修改，系统中看似无关的其他部分出现了问题。
- 很难在别的应用程序中重用某个模块，因为不能将它从现有的应用程序中独立提取出来。

什么原因?

- 耦合关系：“高层模块”过分依赖于“低层模块”

一个良好的设计应该是系统的每一部分都是可替换的。

依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）

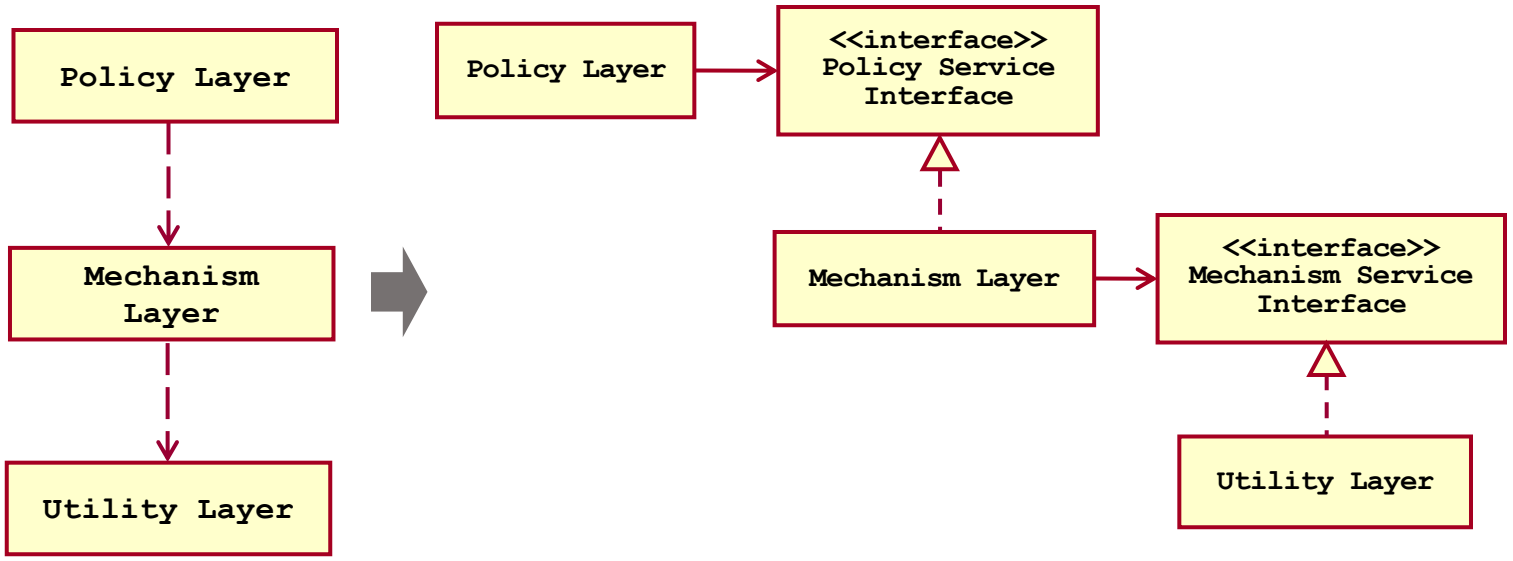
> ***I. High-level modules should not depend on low-level modules. Both should depend on abstractions.***
> ***II. Abstractions should not depend on details. Details should depend on abstractions.***
>
> **R. Martin, 1996**

依赖于抽象

- 任何变量都不应该持有一个指向具体类的指针或引用
- 任何类都不应该从具体类派生
- 任何方法都不应该覆写它的任何基类中已经实现的方法

依赖倒置原则

层次化：应避免较高层次直接使用较低层次。



依赖倒置原则

总结：

- 面向接口编程，不要针对实现编程
- 抽象不应该依赖于细节，细节应该依赖于抽象
- 高层模块和低层模块以及客户端模块和服务模块等都应该依赖于接口，而不是具体实现
- 从问题的具体细节中分离出抽象，以抽象方式对类进行耦合

依赖倒置原则有利于高层模块的复用，其正确应用对于创建可重用的框架是必须的。

Reference

- 清华大学国家级精品课程 《软件工程》主讲人 刘强 副教授 刘璘 副教授
  - https://www.icourses.cn/sCourse/course_3016.html
  - https://www.xuetangx.com/course/THU08091000367/5883555?channel=learn_title
- Maurício Aniche, An Introduction to Software Architecture and Design, TUDelft, 2019
- https://medium.com/tunaiku-tech/s-o-l-i-d-principles-simplified-explanation-example-f7268ca75758

谢谢大家！

THANKS