Lecture 15. Object-Oriented Design-Refine Architecture
面向对象的设计(构件设计)

Object Oriented Modeling Technology
面向对象建模技术

Professor: Yushan Sun

Fall 2022

内容安排

- 关于构件设计
- 用例设计
- 类的设计理论
- 旅游申请系统类的设计实例
- 数据库设计简介

关于构件设计

关于构件（component, 组件）设计

- 任务
  - 基于“架构分析”和“用例分析”的框架，利用“架构设计”提供的素材，在不同的局部，将分析结果用“设计元素”加以“替换”和“实现”
- 主要活动
  - 实现需求场景，称为用例设计，即利用用力分析的结果，细化或修改用例分析顺序图；
  - 实现子系统接口(子系统设计，本门课程不讲)
  - 明确类的实现细节，称为类的设计

Back

用例设计

用例设计

- 用例设计(Use-Case Design)目标
  - 利用交互图(顺序图)改进用例实现
  - 改进对设计类的操作需求
  - 改进对子系统和它们的接口的操作需求
- 输入
  - 用例实现(分析阶段)
  - 设计元素
- 输出
  - 用例实现(设计阶段)

用例分析与用例设计

- 用例分析与用例设计的差别表现在类的职责和操作的差别
  - 用例分析阶段定义类的初步职责（粗糙一些）
  - 用例设计阶段则需要定义具体的操作（C++函数，Java方法）来实现这些职责
    - 发送到设计类的消息，对应该类的操作
    - 发送到子系统的消息，对应其接口的操作
- 分析中主要的业务职责集中在控制类中，因此设计的重点就是控制类职责的实现
  - 臃肿的控制器（Bloated Controllers）

用例设计过程

- 将设计应用于用例
  1. 引入设计元素和设计机制，改进交互图(顺序图)，描述设计对象间的交互
  2. 针对复杂的交互图，引入子系统封装交互，简化交互图
  3. 细化用例实现的事件流，为消息添加与实现相关的细节

改进交互图：职责分配

- 利用设计元素，进行类的职责分配，完成用例实现的交互图
  - 利用设计元素取代分析类
  - 引入架构机制，调整和完善交互图
- 需要遵循相关的设计原则和模式
  - 职责分配模式：GRASP模式
  - 面向对象设计原则：LSP、OCP、SRP、ISP和DIP
  - 引入适用的设计模式

臃肿的控制器

- 臃肿的控制器：低内聚、缺乏重点并且处理过多的职责区；即违背面向对象设计的相关原则：
  - 高内聚、低耦合
  - SRP
- 解决方案
  - 加入更多的控制器（更多的分层）
  - 将部分职责委托给其它对象

用例设计-改进用例实现步骤

- 确定参与用例流的每个对象
  - 用设计元素(设计类)取代分析类
- 在交互图中描绘每一个参与对象
  - 遵循相应的设计原则和模式，利用交互图完成职责分配过程
- 递增地并入可适用的构架机制
  - 引入所需的设计机制(设计模式)，调整和完善交互图

Back

类的设计理论

类设计

- 类设计(Class Design)目标
  - 确保类可为用例实现提供必需的操作
  - 确保提供足够的信息可明确无误地实现
  - 处理和类相关的非功能需求(例如，可扩展性)
- 输入
  - 用例实现(设计)
- 输出
  - 设计类

设计类剖析

- **分析阶段**：只要尽量捕获系统需要的行为，而完全不必考虑如何去实现这些行为
- **设计阶段**：则必须准确地说明类是如何履行它们的职责
  - **属性**：需有完整的属性集合，包括详细说明名称、类型、可视性
  - **操作**：将分析类指定的**职责(粗糙，笼统)**转化成一个或多个**操作（精确）**的完整集合
- 设计类是已经完成了规格说明并且达到能够**被实现程度的类**

类设计的主要内容

- 1. 创建初始设计类
- 2. 定义操作
- 3. 定义属性
- 4. 定义关系

1. 创建初始设计类

- 创建初始设计类，需要考虑
  - 类构造型
    - 边界类
    - 控制类
    - 实体类
  - 可适用的设计模式（在局部软件的设计中使用设计模式，属于另外一门课程）
  - 构架机制
    - 持久性(即持久数据存储)
    - 分布性

边界类的设计策略

- **用户界面（UI）**
  - **使用什么用户界面开发工具？（很多）**
  - **哪些界面可以用开发工具直接创建？**
  - **用户图形界面比较容易创建**

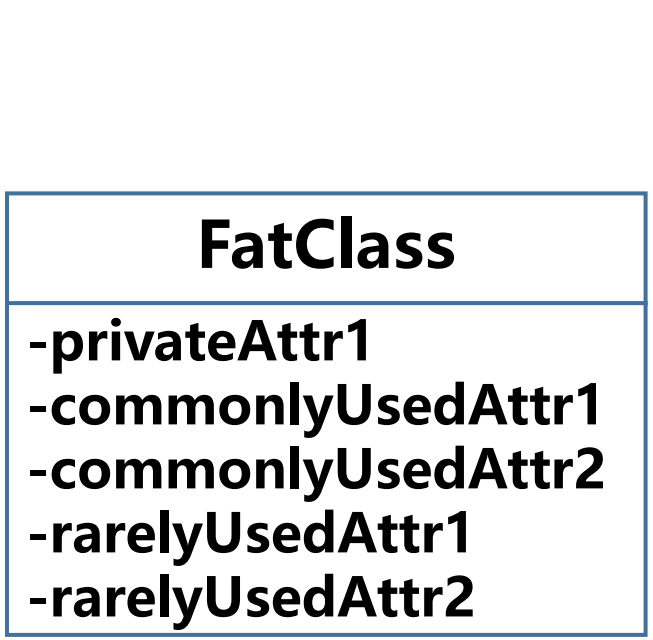
- **外部系统接口(阿里云服务，微信支付，支付宝，百度地图，等)**
  - **通常建模为子系统；在子系统代码中调用外部接口。需要事先申请接口，有的需要付费，有的免费。**
  - **简单程序的情况，也可创建正常的类；在类的代码中，直接调用外部接口。**

实体类的设计策略

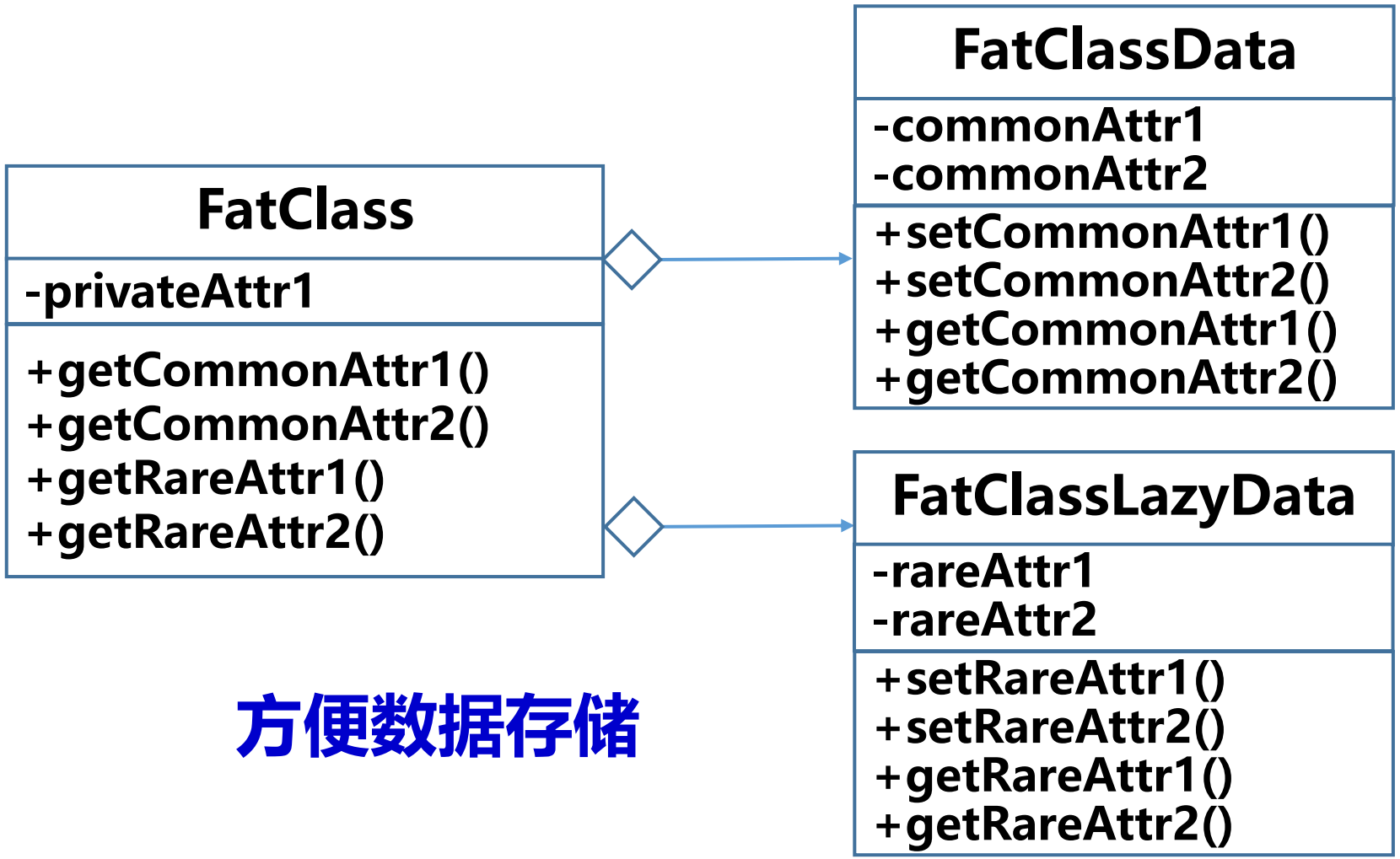
- 实体对象通常是
  - 被动的(不调用别的对象，而是被别的对象调用)和
  - 持久性的（所包含的数据需要存储）

- 性能需求可能要对实体类进行重构，例如考虑到可扩展性，可以考虑使用多态，建立抽象接口类与实现类
- 持久性（关于数据存储）构架机制涉及到实体类

例1：如果一个实体类包含太多数据，考虑到存储的需要，有可能将一个类拆分为两个类



假如，一个类有97个条数据

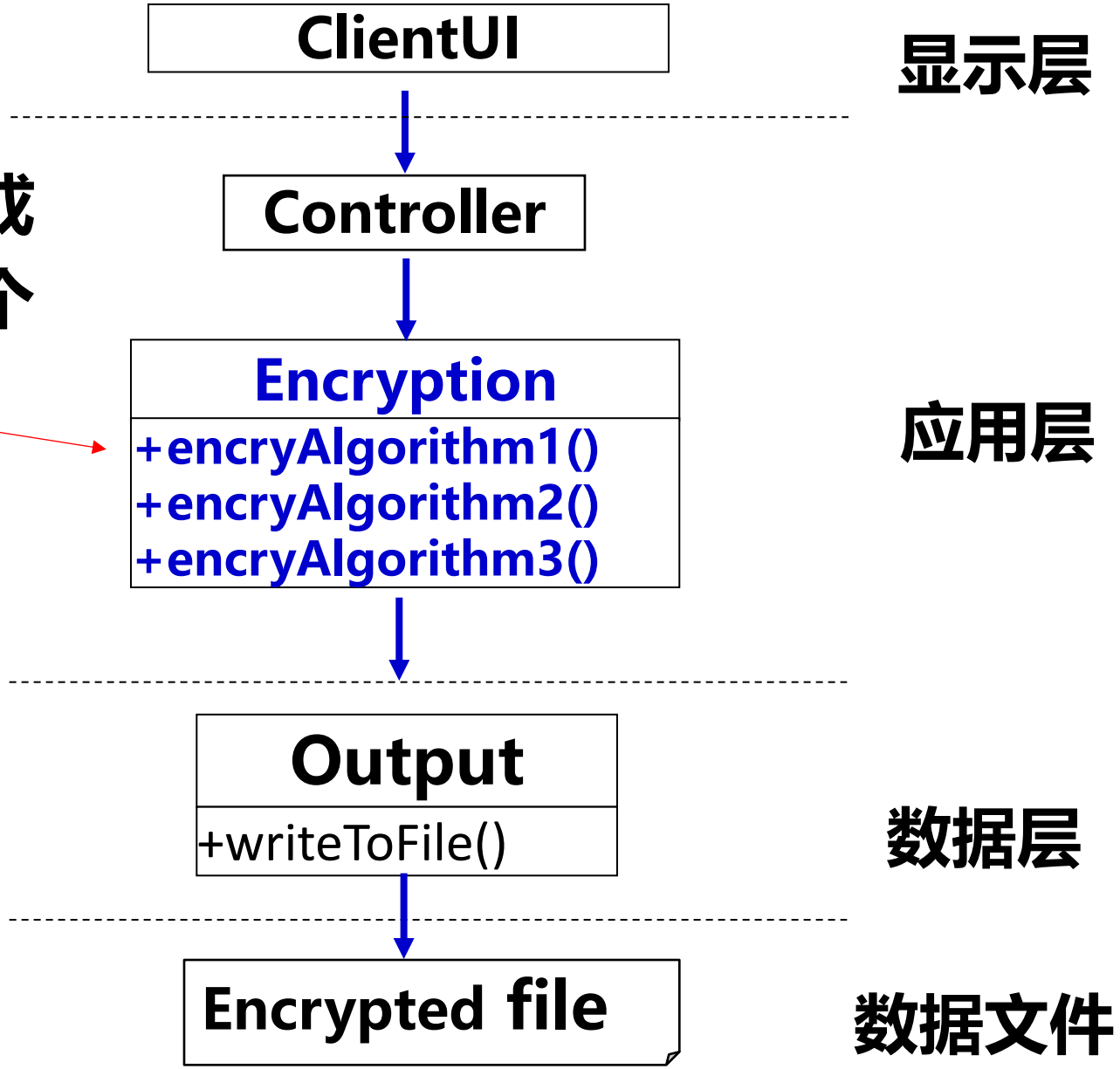


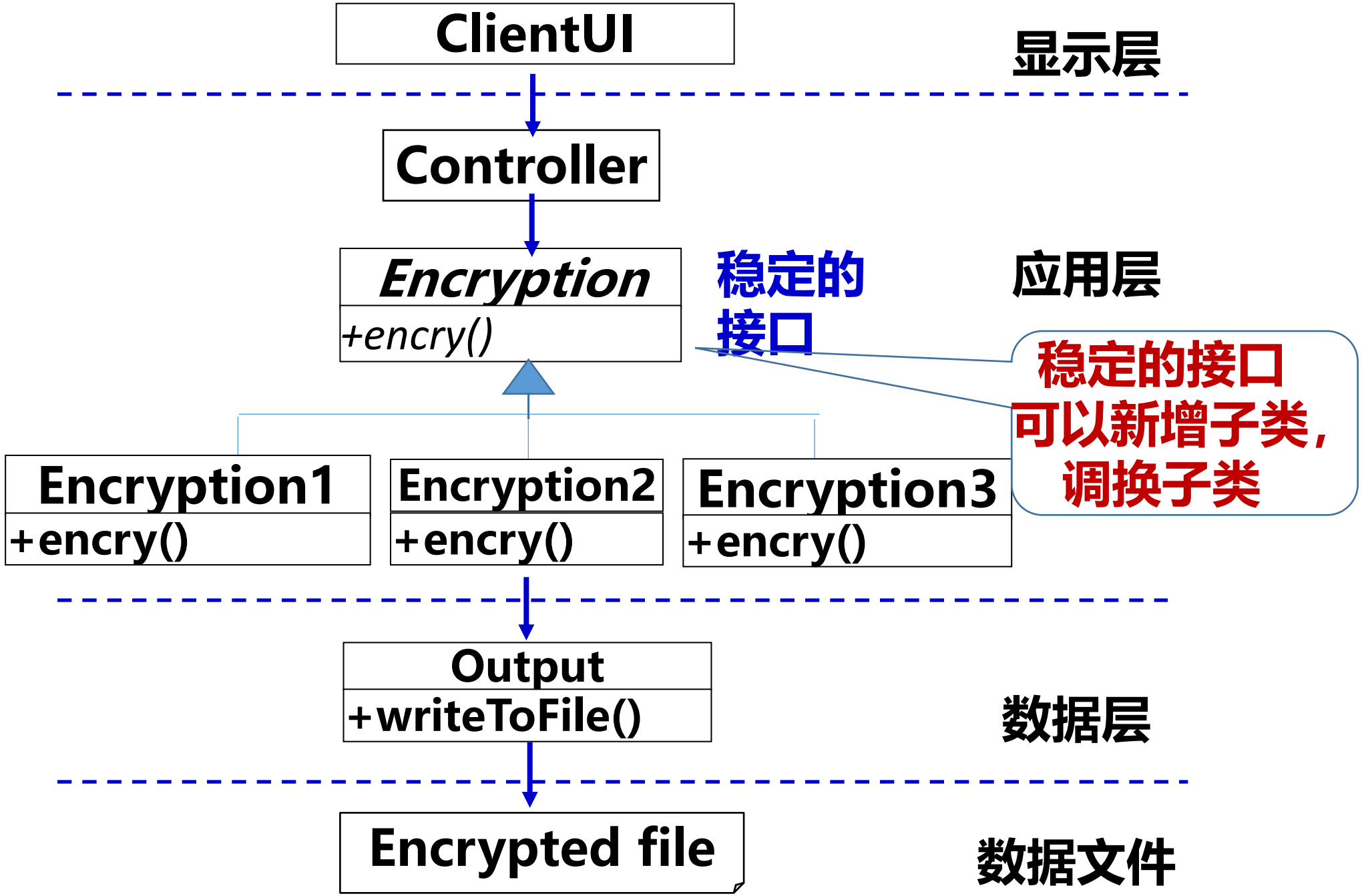
方便数据存储

例2：找出不稳定点，建立稳定接口。

- 确定能预测到的（类型）变化或不稳定点；分配责任以建立一个稳定的接口。

- **解决方案：**按照受保护的变化模式与多态模式，为Encryption类建立一个稳定的接口，设计将下一个PPT




ClientUI
显示层
Controller
Encryption
+encry()
稳定的
接口
应用层
稳定的接口
可以新增子类，
调换子类
Encryption1
+encry()
Encryption2
+encry()
Encryption3
+encry()
Output
+writeToFile()
数据层
Encrypted file
数据文件

控制类的设计策略

- 如何处理控制类
  - 是否真正地需要它们?
  - 它们应当被分开吗?
- 下列情况下，分析阶段的控制类可能变为真正的设计类
  - 封装非常重要的控制流行为
  - 封装的行为很可能变化
  - 必须跨越多个进程或处理器进行分布
  - 封装的行为要求一些事务管理

调整控制类

- 调整控制类的基本策略
  - **提供公共控制类**：多个用例若有同样活动的控制类，将其整合起来，把相同部分作为一个新的控制类
  - **分解复杂控制类**：若用例的控制流程过于复杂，则可以考虑根据不同的控制业务分解成多个控制类

2. 定义操作

- 操作是类的行为特征，描述了该类对于特定请求做出应答的规范
  - 同一个类的每个操作都具有唯一签名，通过描述操作的签名完成类操作的定义
  - UML中的四种可见性
    - 公有（+）、私有（-）、保护（#）和包（~）

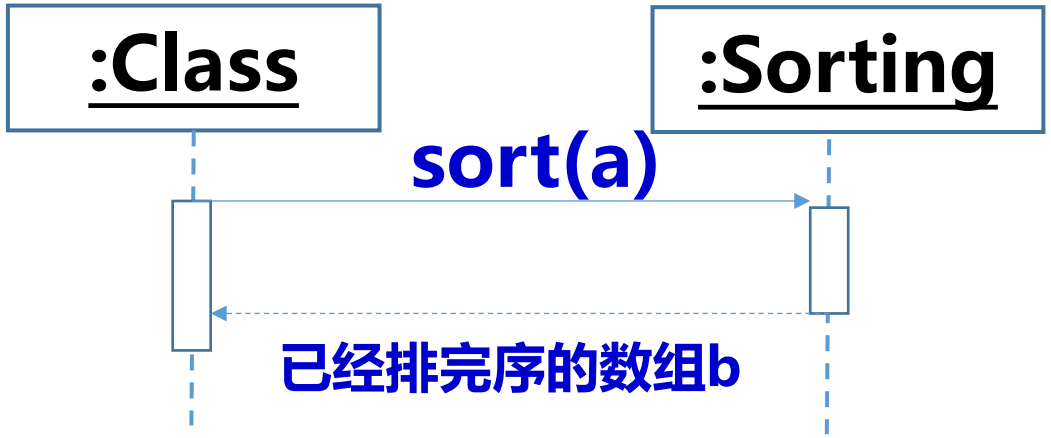
```
public int[] sort(int[] a) {
        // code
}
```

```
private int[] sort(int[] a) {
        // code
}
```

怎样确认操作?

- 交互图中的消息成为类中的方法
- 如果消息的箭头指向类A，则类A就具有消息所代表的方法

例3 消息➔方法的例子



- 其它独立功能的实施
  - ➢ 自身的管理功能(get, set方法；构造函数、析构函数等)
  - ➢ 类复制的需要(测试类是否相等，创建类副本等)
  - ➢ 其它操作机制的需要(垃圾收集、测试等)

例4：定义类的操作

Application
+setTourGroup(g: TourGroup): void
+setPeronLiable(a: Adult): void
+getChargeInfo(): String
+calcFee(): float
+calcDeposit(): float
+updateAppInfo(): void

设置旅游团，通过参数传入TourGroup对象

设置责任人，通过参数传入Adult对象

获得支付信息，返回支付信息

计算费用，返回费用

计算定金，返回定金

更新申请信息

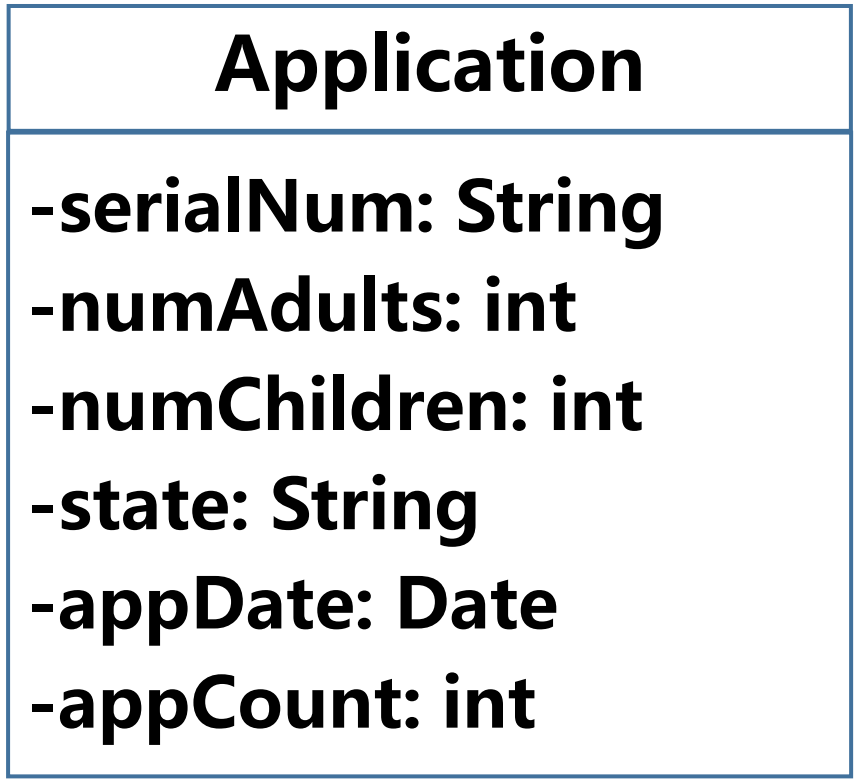
关于操作(方法)应该考虑的内容

- 详细说明操作实现的细节
- 可采用UML活动图对方法进行建模
- 考虑的内容：
  - 特殊算法（由专家给出，程序员编写代码实现）
  - 要使用到其它对象和操作（调用，由顺序图给出）
  - 属性和参数如何实现和使用 （如果某个类的属性没有任何代码使用过，那么这个属性有点可疑）
  - 关系如何实现和使用 （如果是聚合关系，则可以在聚合类中明确）

3. 定义属性

- 指定名字、类型、可见性和可选的缺省值
  - 可见性 属性名 类型
  - 类型应当是编程语言支持的数据类型(例如，Java 的Date类)
- 发现属性(attributes)
  - 检查类自身需要维护的所有信息，如果缺失(例如id)，补上
  - 检查方法和状态，从而发现新的属性

例5. 定义属性



申请序列号

成年人人数

儿童人数

状态（是否满员）

申请日期

申请总人数

4. 定义关系

依赖关系

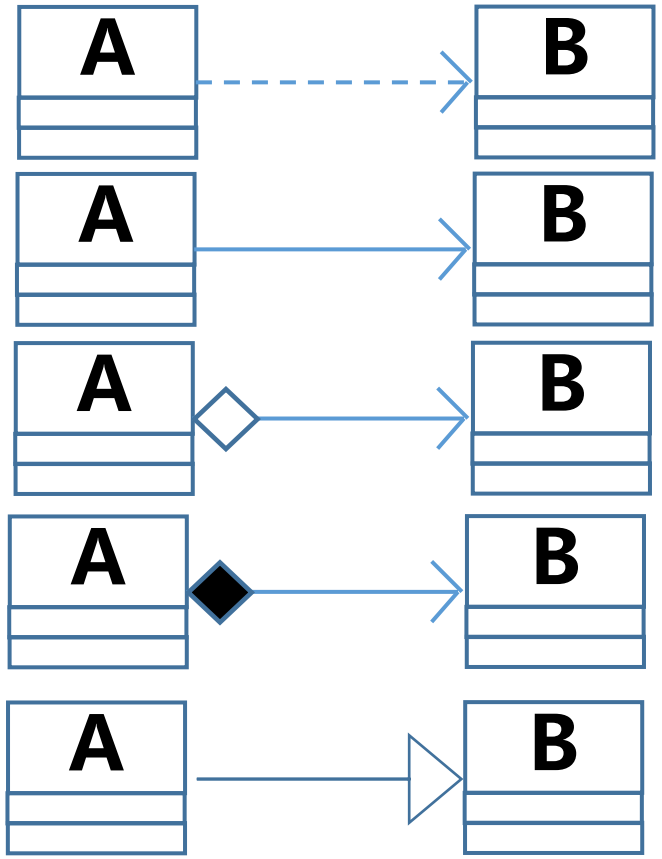
关联关系

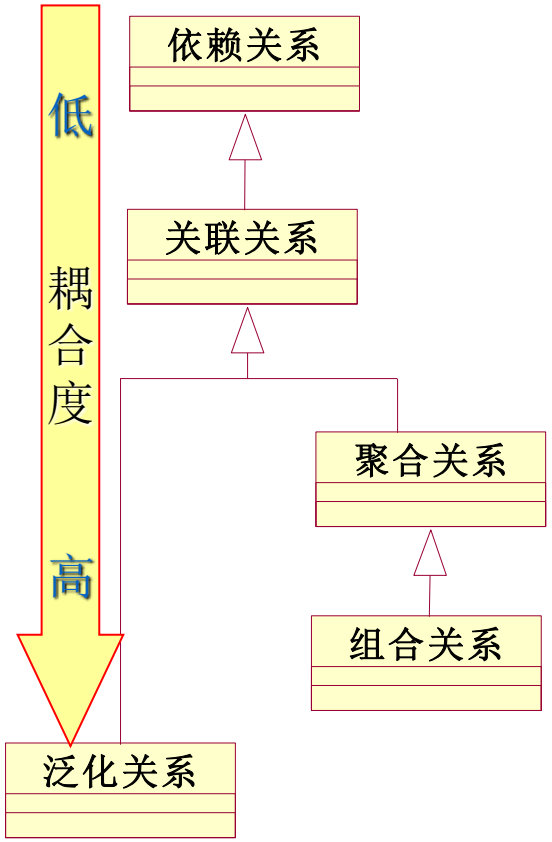
聚合关系

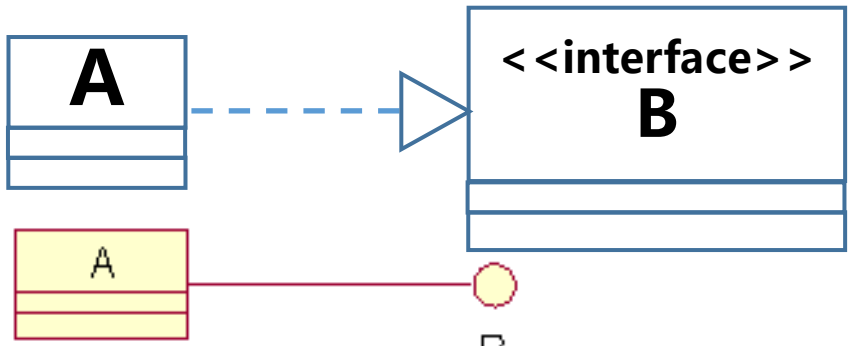
组合关系

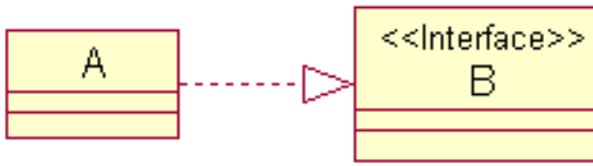
泛化关系

实现关系
(类A实现接口B)





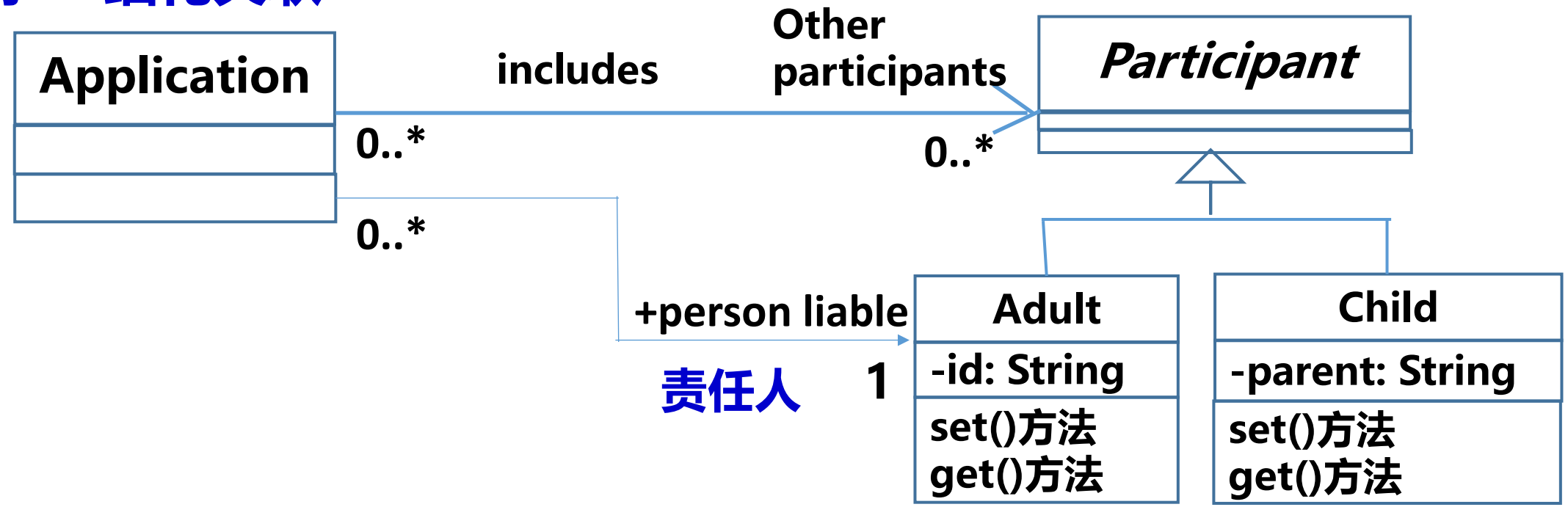




关联关系的表示方法

- 在分析阶段给出的粗略的关联关系，在设计时需要细化：
  - 关联名称：动词短语
  - 关联的多重性表达式：*，1..*，1-40，5，3, 5, 8，...
  - 导航性、端点名称

例6. 细化关联



关联的导航性

- 关联的方向性(导航)是指对象间链接的方向(也可理解为消息发送的方向)
  - **在分析阶段**，如果没有考虑方向性；则此时默认为双方向的关联(互相知道对方对象，可以互相发送消息)
  - **在设计阶段**，在有可能的情况下将关联关系改为单方向的关联
    - 好的设计目标是最小化类间耦合
    - 双向关联难以实现，需要消耗成本、内存等

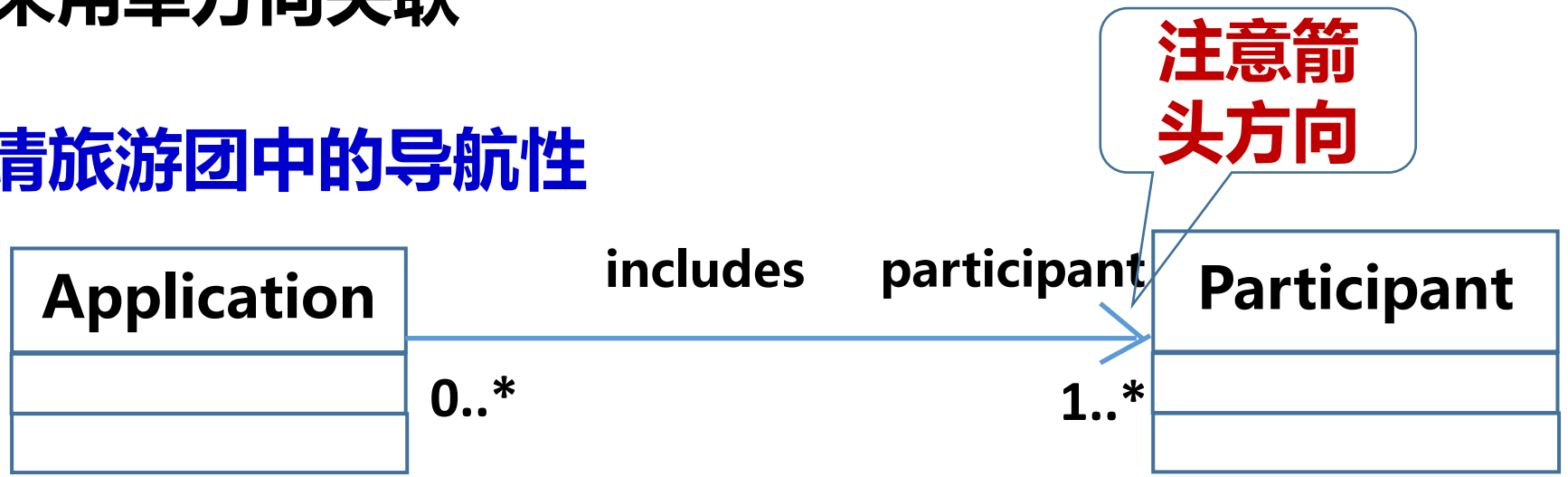
单方向关联的设计

- 设计考虑：类A与类B关联时
  - 类A的对象是否需要知道类B的对象(即类A的对象是否向类B的对象发送消息)
  - 类B的对象是否需要知道类A的对象(即类B的对象是否向类A的对象发送消息)
- 设计规则：
  - 通过分析顺序图，如果只向一个方向发送消息，则定义为单方向的关联(方向与消息的发送方向一致)
  - 如果双向发送消息时，则需要进一步的考虑

双方向关联的设计

- 如果双向发送消息时，可能的方案：
  - 方案1：采用双方向的关联(有的实体类之间确实应该有双向关联)
  - 方案2：改变原有的消息发送顺序，从而将消息改成单方向的发送，从而采用单方向关联

例7：申请旅游团中的导航性



提示编程人员：Application对象可以调用Participant对象；但是Participant对象不能调用Application对象。

泛化关系的设计

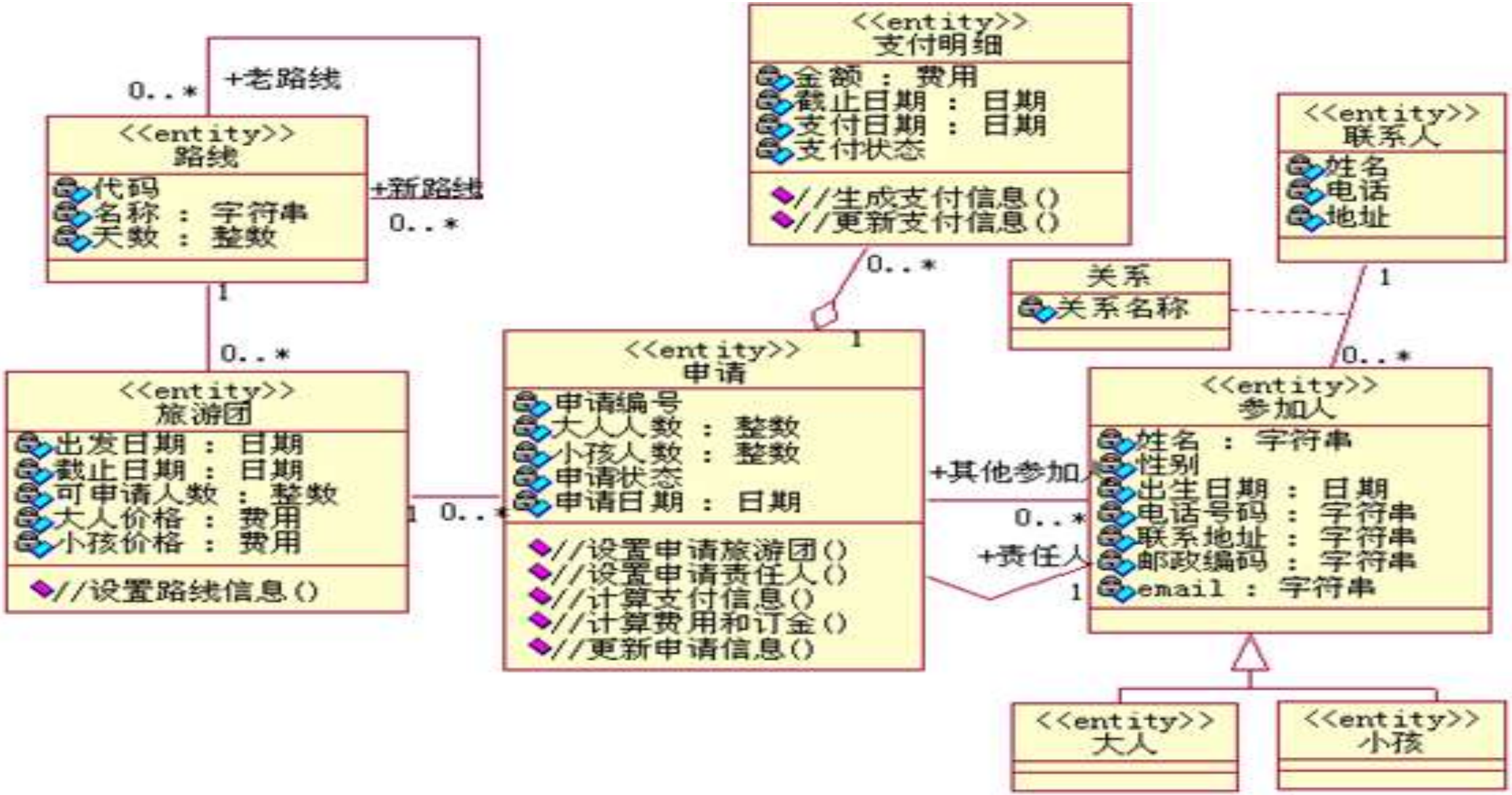
- 只有在两个设计类之间
  - 存在清晰明确的“is a”关系或
  - 为了可扩展性
  - 为了复用代码(子类复用超类的代码)

  才使用继承（但是注意不要因此引入耦合）
- 继承也有缺点
  - 类间耦合的最强形式：子类会继承超类的属性、方法、关系
  - 类层次中的封装是脆弱的，超类的改动会直接波及底下的层次
  - 关键软件(航空，航天，金融等)中，慎用继承
- 注意Liskov替换原则的应用

Back

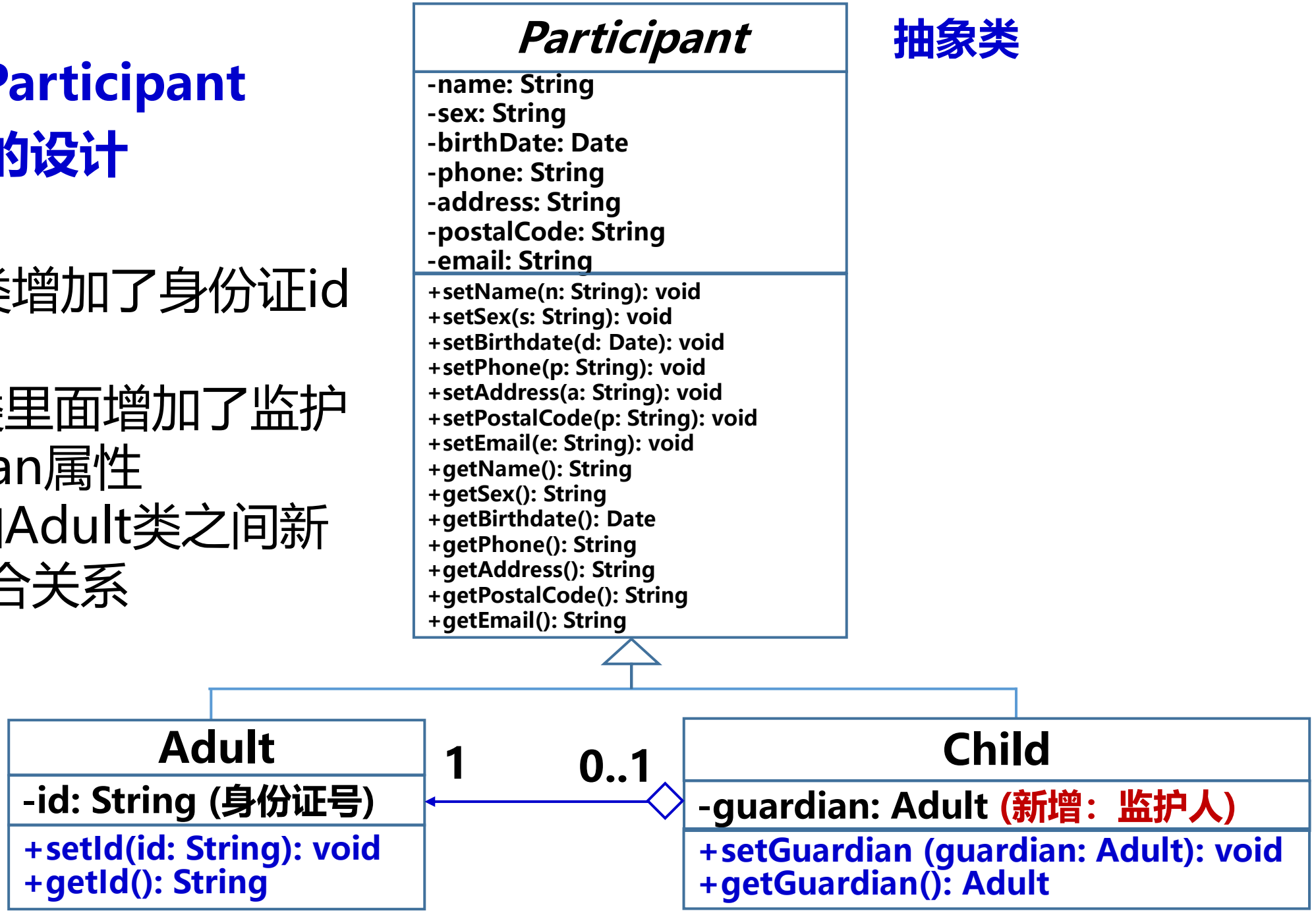
旅游申请系统类的设计实例

旅游申请系统实体类分析类图



例8：参加人Participant层次类的设计

1. 在Adult类增加了身份证id属性
2. 在Child类里面增加了监护人guardian属性
3. 在Child和Adult类之间新定义了聚合关系



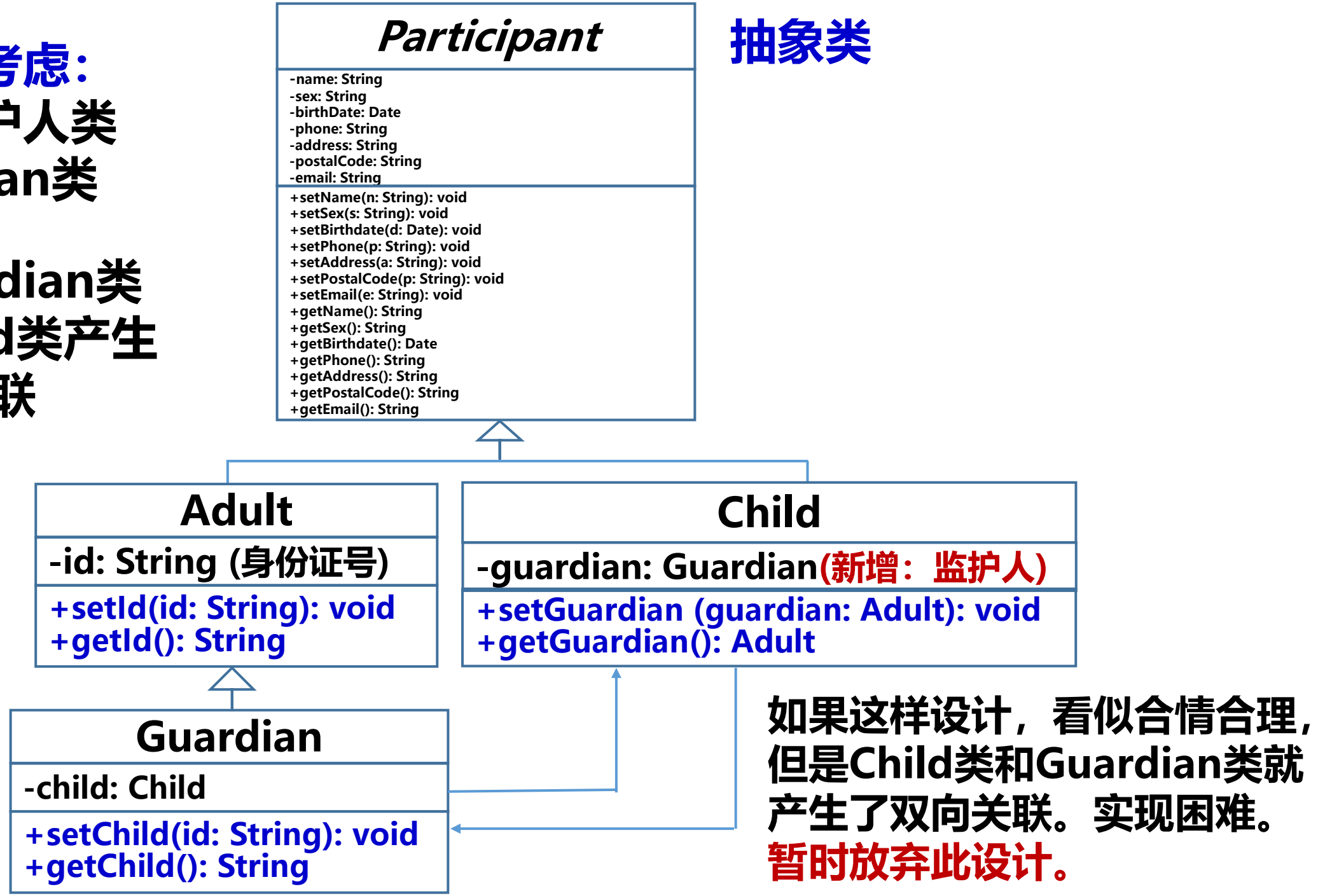
- 说明：

1. 每个子类都拥有超类的属性作为自己的属性
2. 每个子类都拥有超类的方法作为自己的方法
3. 每个子类应该有属于自己的特别属性；若没有，理论上不应该创建这个子类。例如，Adult类有自己特有的属性：id; Child也有自己的属性：guardian
4. 既然超类里面没有身份证(id)属性，而子类中只有Adult才包含id，Child只有监护人guardian属性；所以，**创建Participant对象毫无意义。**从而应该将超类设计为抽象类。该层次类只有两个子类能够被创建对象。

设计优点：可扩展性好。例如，增加老年人Senior，和Disabled子类。

另外的考虑：
增加监护人类Guardian类

Guanrdian类与Child类产生双向关联



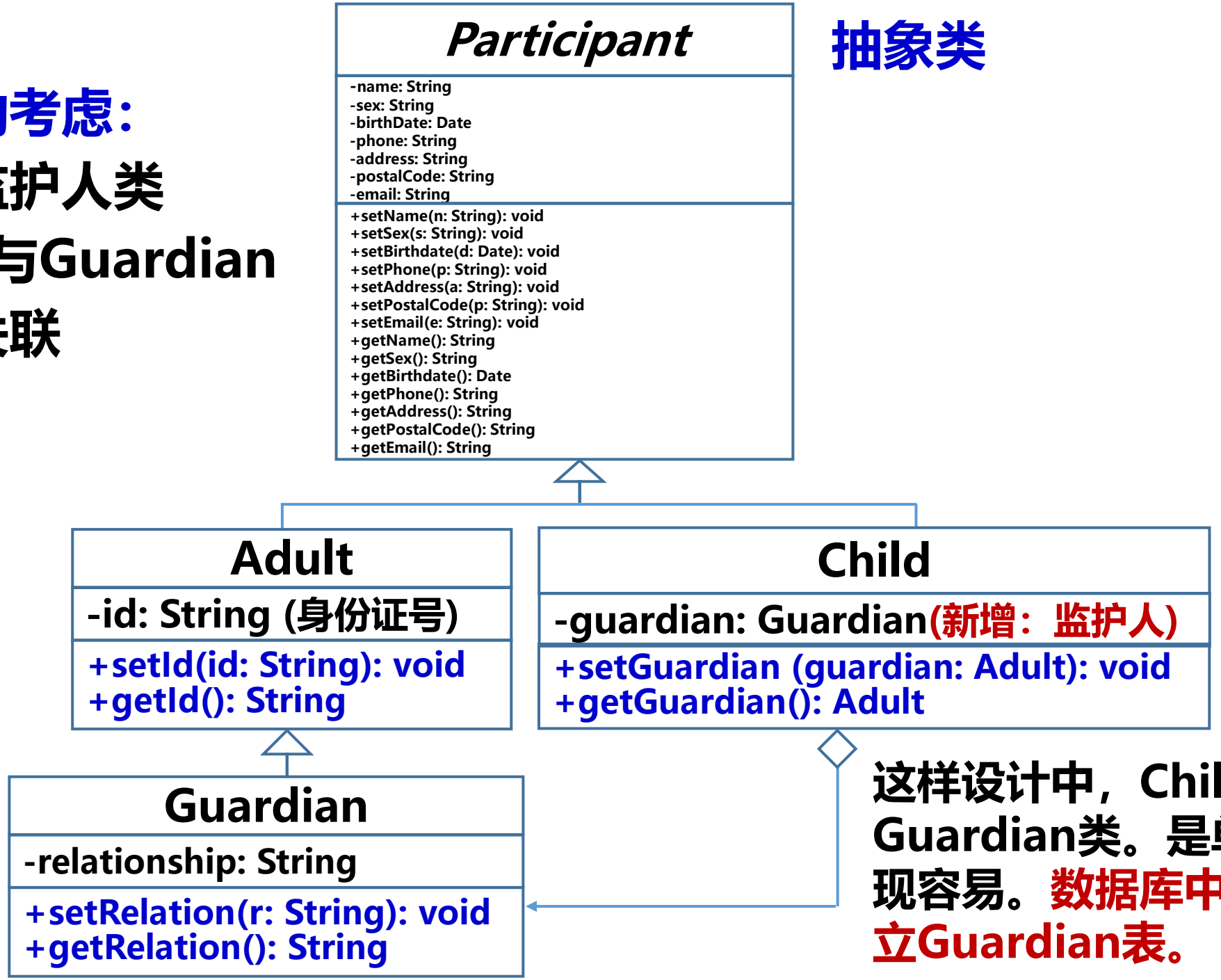
如果这样设计，看似合情合理，但是Child类和Guardian类就产生了双向关联。实现困难。
暂时放弃此设计。

另外的考虑：

增加监护人类

Child与Guardian

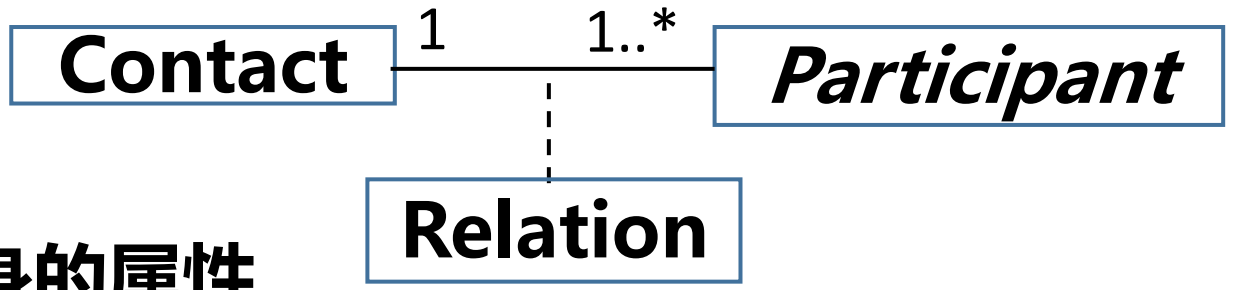
单向关联



抽象类

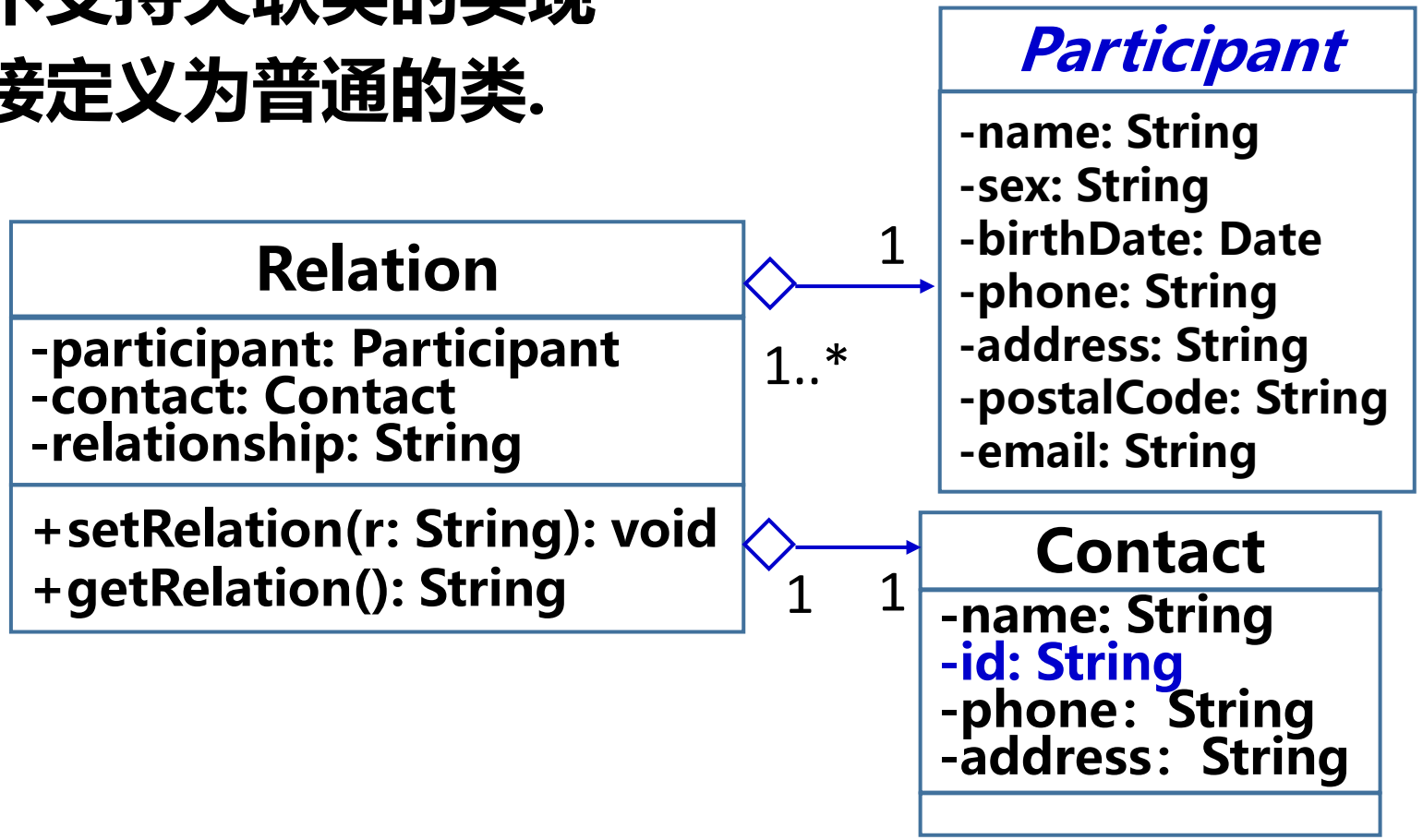
这样设计中，Child类聚合了Guardian类。是单向关联，实现容易。数据库中也可单独建立Guardian表。

例9. 参加人与联系人之间的关联类Relation设计如右

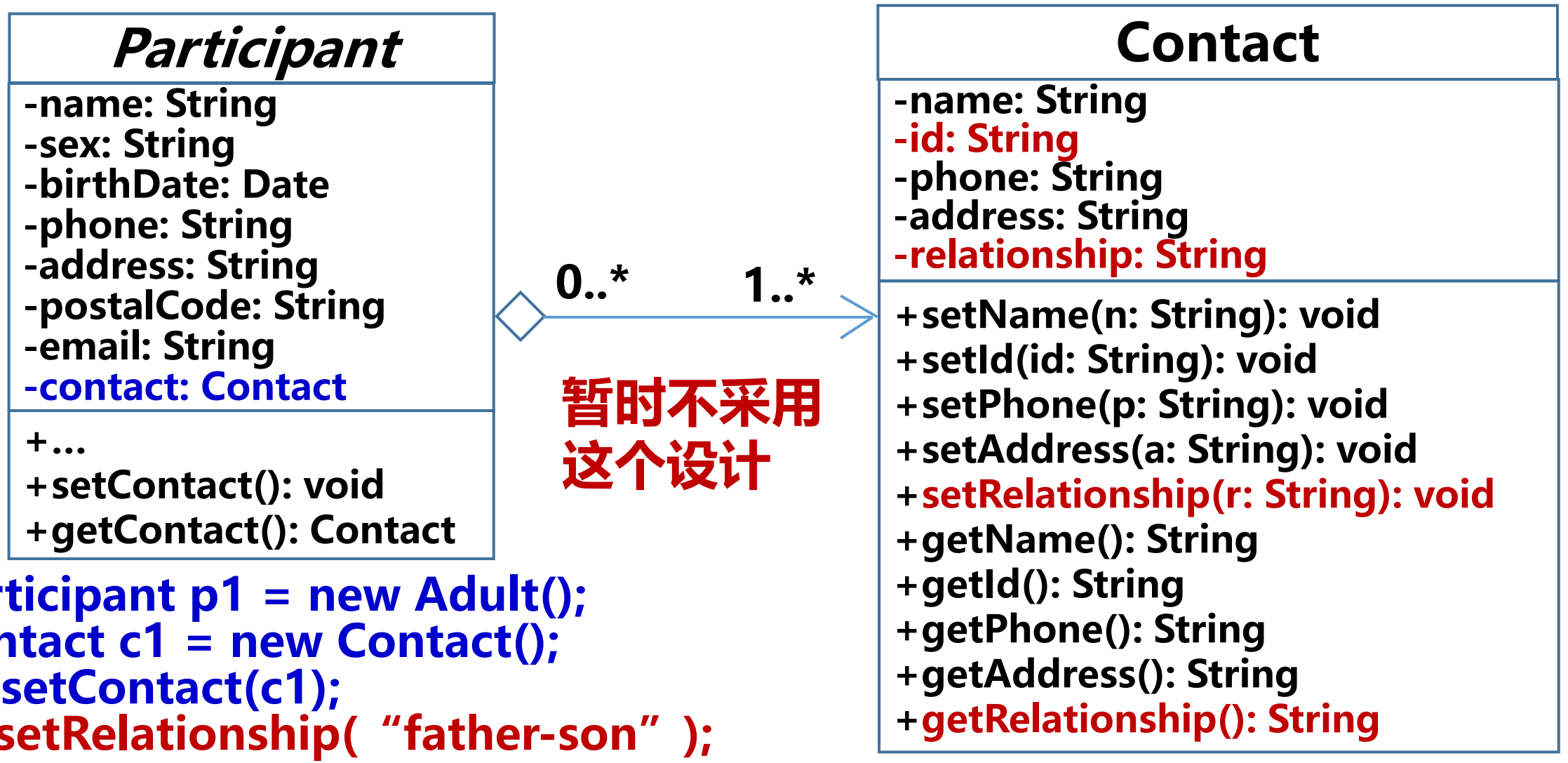


- 分析阶段使用关联类来描述关系本身的属性
  - 而面向对象的编程语言不支持关联类的实现
  - 设计时需要将关联类直接定义为普通的类.

```
Participant p1 = new Adult();
Contact c1 = new Contact();
Relation r1 = new Relation(p1, c1);
r1.setRelation("father-son");
Participant p2 = new Child();
Contact c2 = new Contact();
Relation r2 = new Relation(p2, c2);
r2.setRelation("friends");
```

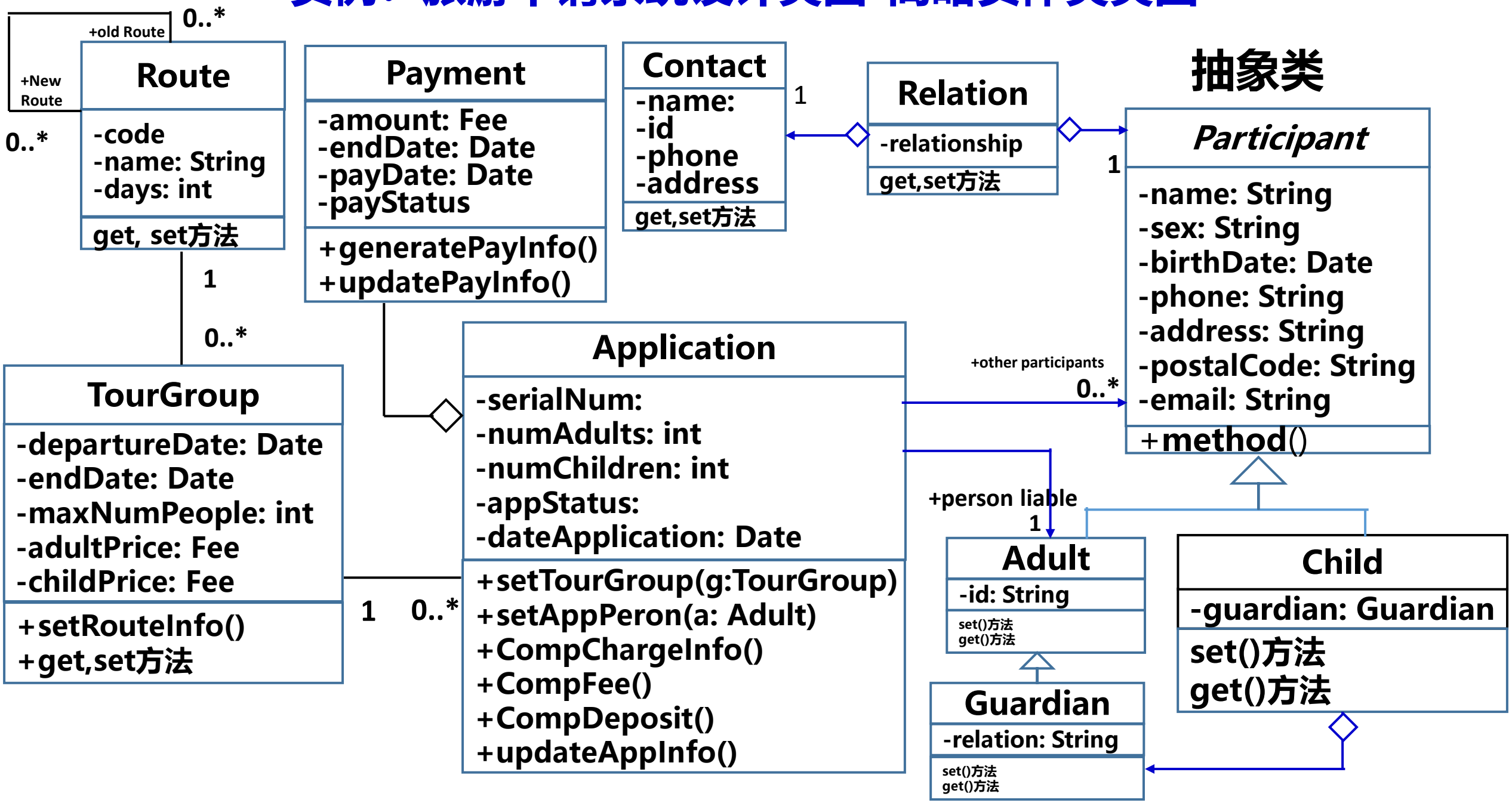


- **例10. 取消关联类的设计**
- **由Participant类添加一个Contact类型的属性；在创建Contact对象的时候，添加Contact数据。可以采取Participant类聚合Contact类的方式解决。**




```
Participant p1 = new Adult();
Contact c1 = new Contact();
p1.setContact(c1);
c1.setRelationship( “father-son” );
```

实例：旅游申请系统设计类图-简略实体类类图



旅游预定系统详细设计

Application类的详细设计

Application

-serialNum: String
-numAdults: int
-numChildren: int
-appState: String
-appDate: Date
-appCount: int
-tGroup: TourGroup
-liablePerson: Participant

+setSerialNum(s: String): void
+setNumAdults(na: int): void
+setNumChildren(nc: int): void
+setAppState(s: String): void
+setAppDate(d: Date): void
+setAppCount(c: int): void
+getSerialNum(): String
+getNumAdults(): int
+getNumChildren(): int
+getAppState(): String
+getAppDate(): Date
+getAppCount(): int
+setTourGroup(t: TourGroup): void
+setLiablePeron(p: Participant): void
+getChargeInfo(): String
+calcFee(): float
+calcDeposit(): float
+updateAppInfo(): void

说明：

1) 一个Application对象代表一次申请
2) 一个申请的人数大于等于1
3) 每次申请都有一位责任人(可能还包含几位大人、小孩)
4) 一次申请中的旅游人员都参加同一个旅游团

TourGroup, Route类的详细设计

TourGroup
-departureDate: Date -endDate: Date -maxNumPeople: int -adultPrice: float -childPrice: float
+setDepartureDate(d: Date): void +setEndDate(d: Date): void +setMaxNumPeople(mp: int): void +setAdultPrice(p: float): void +setChildPrice(p: float): void +getDepartureDate(): Date +getEndDate(): Date +getMaxNumPeople(): int +getAdultPrice(): float +getChildPrice(): float +setRouteInfo(): void

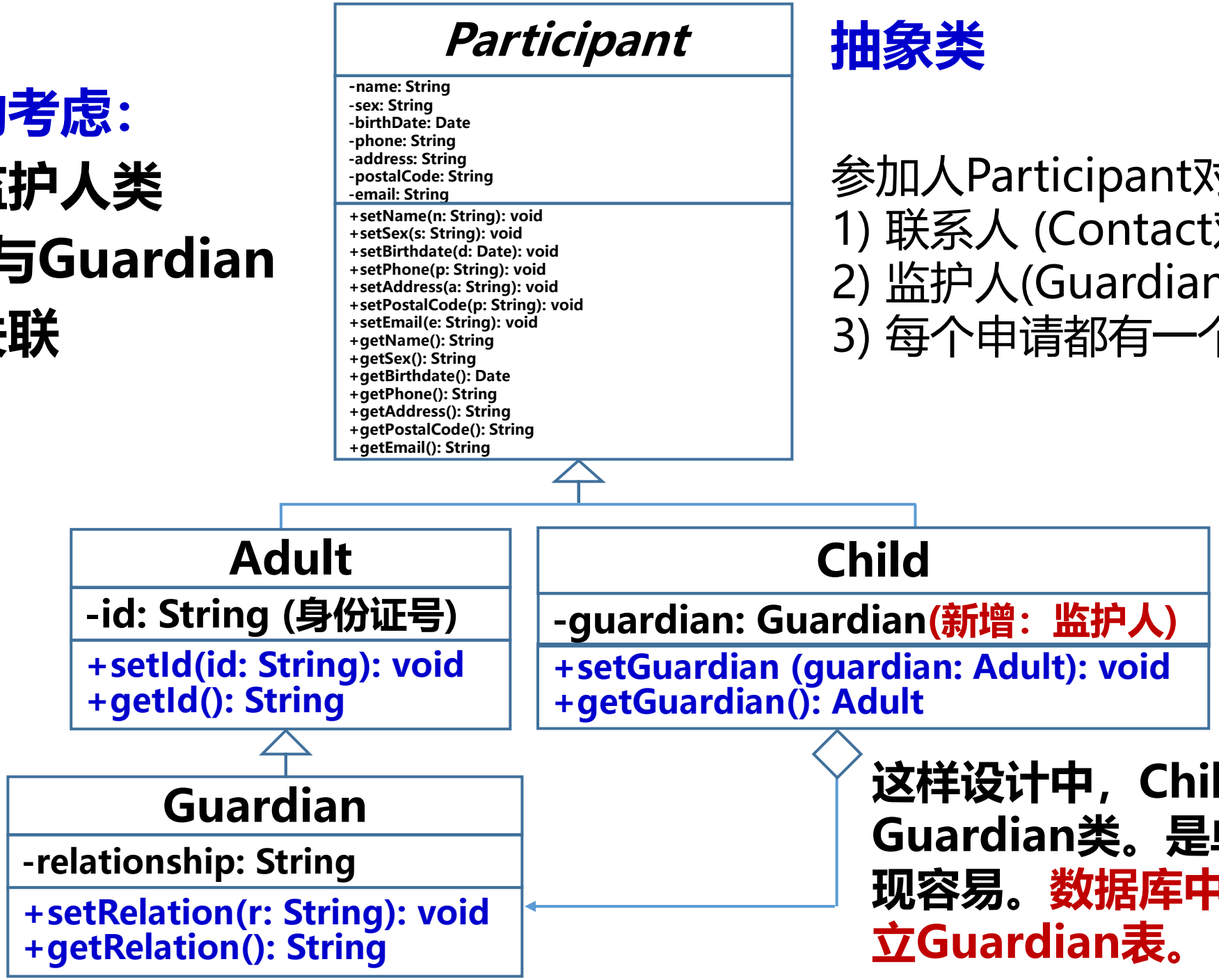
Route
-code: String -name: String -days: int
+setCode(code: String): void +setName(name: String): void +setNumDays(numDays: int): void +getCode(): String +getName(): String +getNumDays(): int

另外的考虑：

增加监护人类

Child与Guardian

单向关联



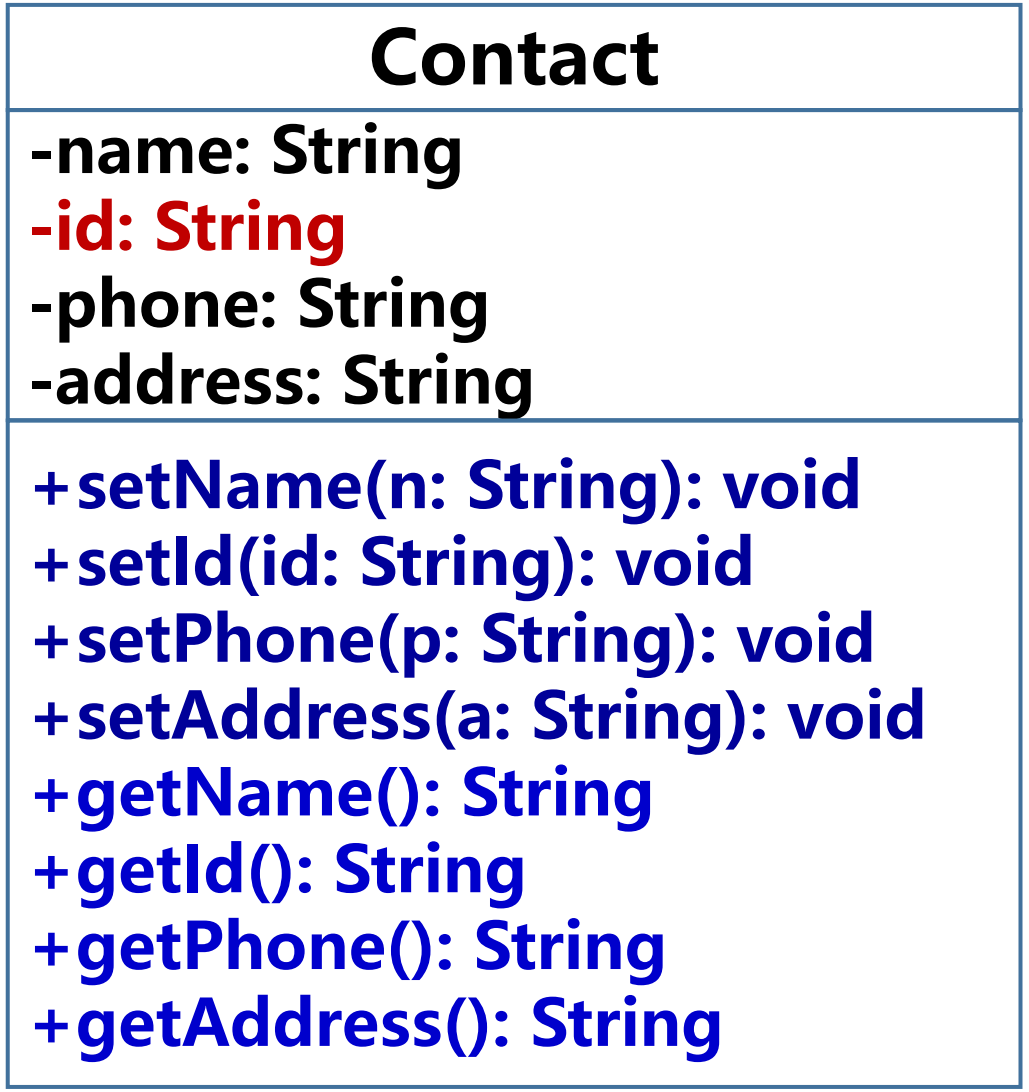
抽象类

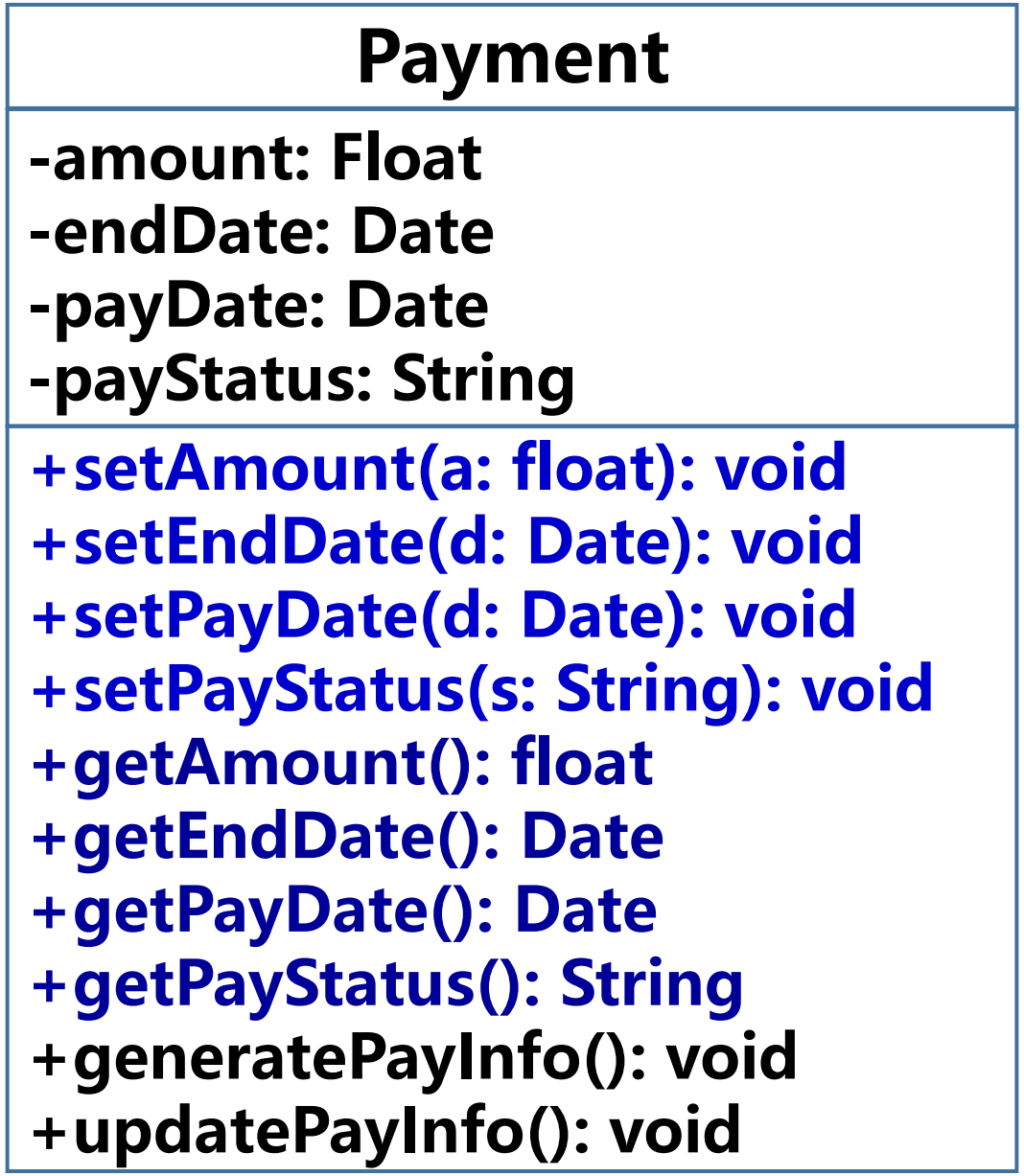
参加人Participant对象涉及到

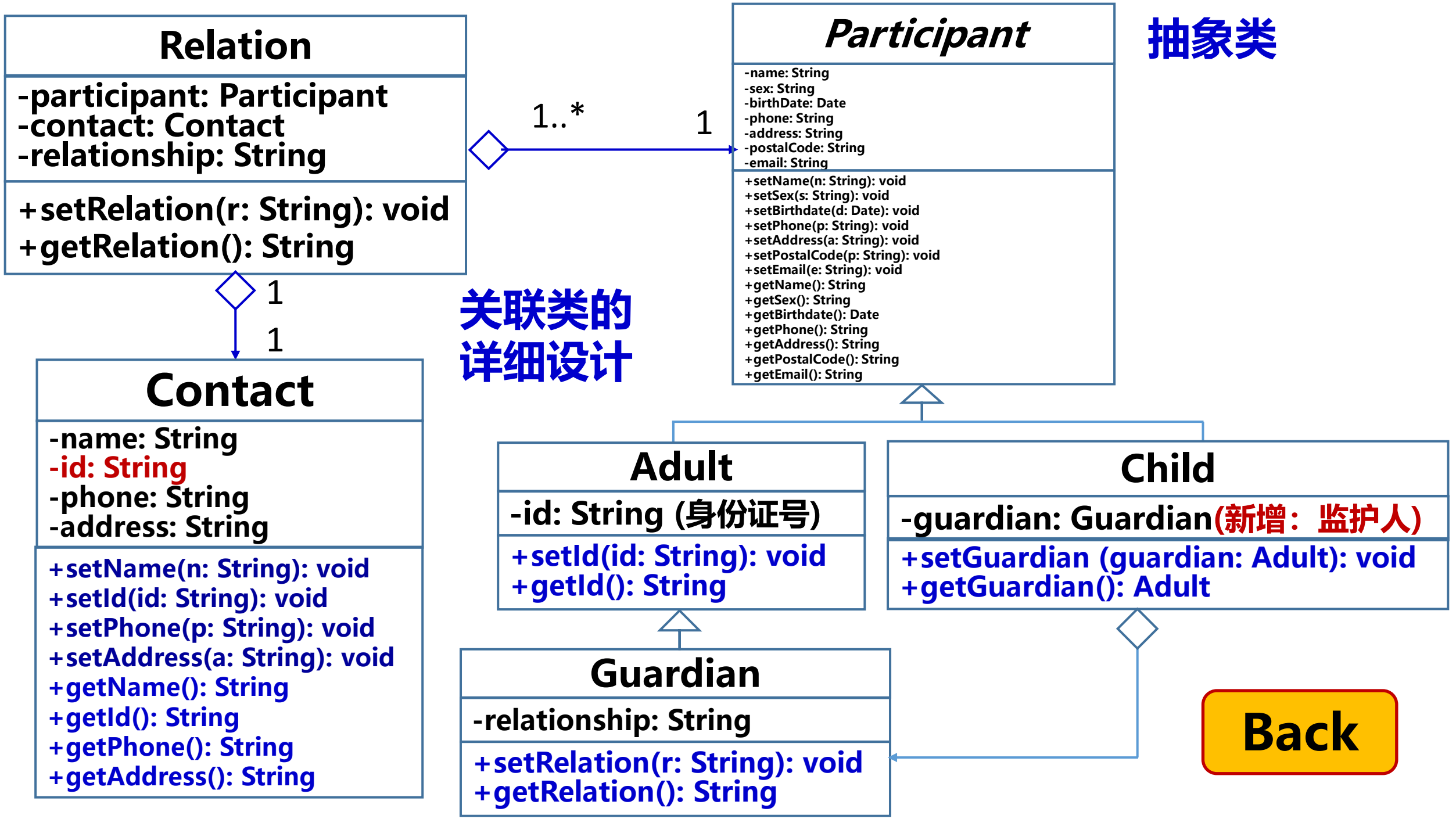
1) 联系人 (Contact对象)
2) 监护人(Guardian对象)
3) 每个申请都有一个责任人

这样设计中，Child类聚合了Guardian类。是单向关联，实现容易。数据库中也可单独建立Guardian表。

Contact和Payment类的详细设计






Relation
-participant: Participant
-contact: Contact
-relationship: String
+setRelation(r: String): void
+getRelation(): String
1..*
1
Participant
-name: String
-sex: String
-birthDate: Date
-phone: String
-address: String
-postalCode: String
-email: String
+setName(n: String): void
+setSex(s: String): void
+setBirthdate(d: Date): void
+setPhone(p: String): void
+setAddress(a: String): void
+setPostalCode(p: String): void
+setEmail(e: String): void
+getName(): String
+getSex(): String
+getBirthdate(): Date
+getPhone(): String
+getAddress(): String
+getPostalCode(): String
+getEmail(): String
抽象类
关联类的
详细设计
1
1
Contact
-name: String
-id: String
-phone: String
-address: String
+setName(n: String): void
+setId(id: String): void
+setPhone(p: String): void
+setAddress(a: String): void
+getName(): String
+getId(): String
+getPhone(): String
+getAddress(): String
Adult
-id: String (身份证号)
+setId(id: String): void
+getId(): String
Child
-guardian: Guardian(新增：监护人)
+setGuardian (guardian: Adult): void
+getGuardian(): Adult
Guardian
-relationship: String
+setRelation(r: String): void
+getRelation(): String
Back

将永久数据存入数据库

存储：对象的持久化问题

- 文件
  - 各种格式的文件（.txt, .ini…）
- 关系数据库（RDBMS）（最常用）

- 面向对象数据库（OODBMS）

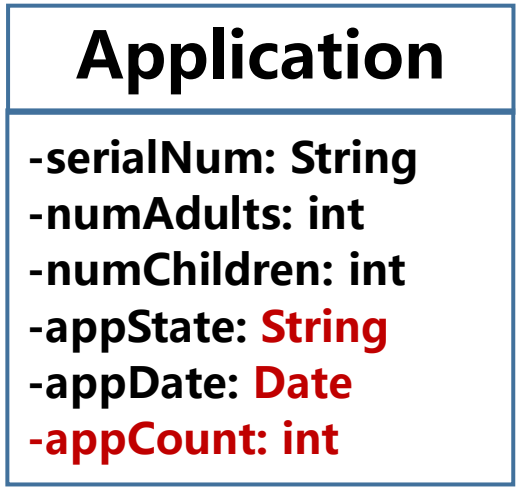
- 关系数据库正在向对象-关系数据库发展

数据库设计

- 数据库设计(Database Design)目标
  - 确定设计中的持久性类
  - 设计适当的 数据库 以及 表 以存储持久化类

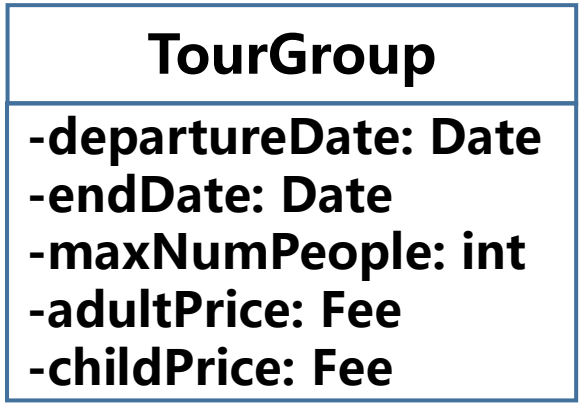
旅游团申请系统数据库表的设计

每个 实体类 所包含的数据都在数据库中被设计成一个表。



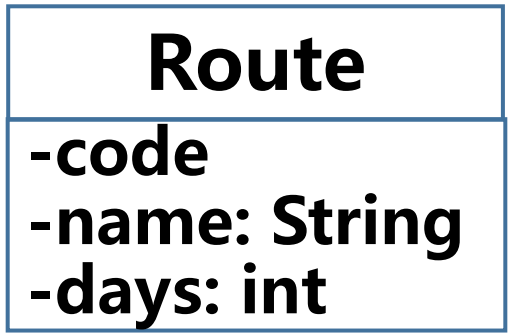
Application

	serialNum	numAdults	numChildren	appState	appDate	appCount



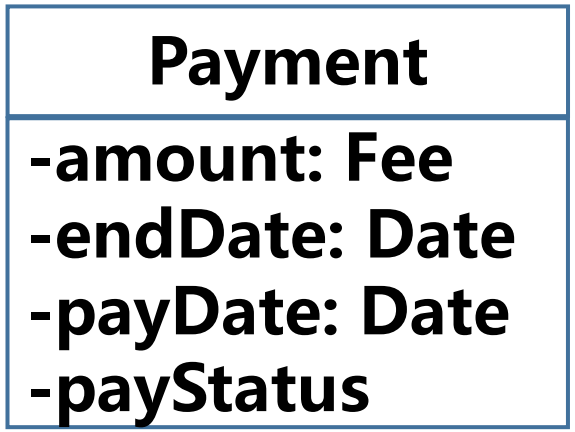
TourGroup

	departDate	endDate	maxNumPeople	adultPrice	childPrice



Route

route-oid	code	name	days

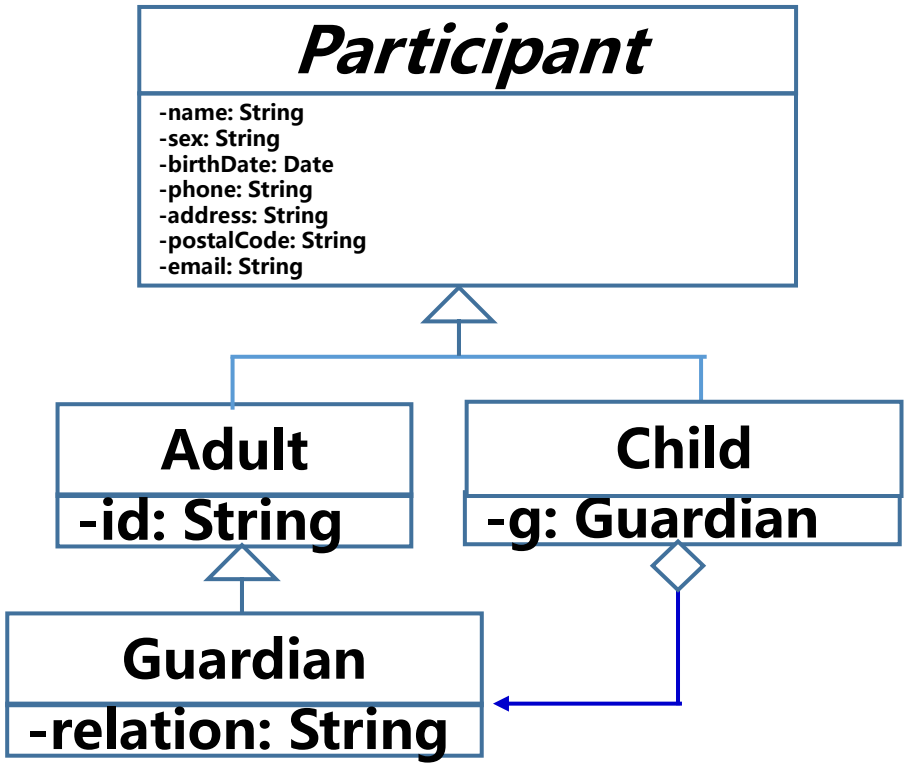


Payment

payment-oid	amount	endDate	payDate	payStatus

Contact

contract-oid	name	id	phone	address



超类*Participant*是抽象类，不会被用来创建对象，因此只有两个子类需要存储数据，所以，我们创建两个表。

Adult

adult-oid	id	name	sex	birthdate	phone	address	postalCode	email

Guardian

guardian-oid	relation	name	sex	birthdate	phone	address	postalCode	email

Child

外键

child-oid	name	sex	birthdate	phone	address	postalCode	email	guardian

Application

app-oid	serialNum	numAdults	numChildren	appState	appDate	appCount

TourGroup

tour-oid	departDate	endDate	maxNumPeople	adultPrice	childPrice

Route

route-oid	code	name	days

Payment

Payent-oid	amount	endDate	payDate	payStatus

Contact

contact-oid	name	id	phone	address

Relation

relation-oid	relationName	adult-oid	Child-oid	Contact-oid

Adult

adult-oid	id	name	sex	birthdate	phone	address	postalCode	email

Guardian

guardian-oid	relation	Adult-oid

Child

child-oid	name	sex	birthdate	phone	address	postalCode	email	guardian-oid

Back