

软件工程

大连理工大学软件学院



第6章 代码生成

- 考虑设计方案向实际运行方式的转变过程，即由概要设计产生出对应的程序代码框架的过程。
- 工程化的设计方法将导致程序代码具有更好的可实现性、更好的可维护性和可修改性以及更好的可扩展性。
- 先对类图到可运行程序的基本转换过程进行概要的说明，然后考虑对其优化和细化的过程。

CASE工具

- 软件开发环境指支持软件开发的工具及其集成机制，用以支持软件开发的过程、活动和任务，为软件的开发、维护及管理提供统一的支持，也即计算机辅助软件工程（CASE）。
- 不同类型的项目其软件开发环境的构成是不一样的，在工具选择上也会有较大的不同。
- 工具的选择会对开发过程的选择产生一定的影响，因为不同的工具组合对开发过程的支持能力会有所不同。同时涉及到工具的集成问题。
- 集成开发环境（IDE）是软件开发环境中的一种实现方式。

“变更”的管理方式

- 需求分析、概要设计和详细设计阶段只进行一次或者迭代-增量式的进行。每次修改只发生在代码，其它文档不做更新。
- 每个改动的意愿都要经过完整的分析、概要设计和详细设计流程，所有必须的改动需要在所属的文档以及代码中对应修改，并保证它们的一致性。
- 以上两种方式的选择需要根据项目需要进行确定，或者在两者之间折中。

逆向工程

- 逆向工程的作用是将代码的修改反向映射回类图的设计中，从而在设计与代码实现之间保证一致性。
- 逆向工程的一种特殊的情况是设计图纸完全由代码生成。
- 逆向工程使得所有的开发都可以在CASE工具中同时展开，并使得设计类图与实现之间的相互对应。
- 逆向工程需要设计和编码工具紧密集成和配合。

单个类的代码实现

- 一个类图如果要成功翻译成为代码的蓝图，类模型中的内容必须要完整。需要包含的信息：
 - 每个实例变量，需要指定其类型；
 - 每个方法中的参数和返回值，需要指定其类型；
 - 每个关联关系，其关联类型、使用或导航方向必须说明。

Employee
-empno : int <u>-empcount : int</u> -lastname : String -firstname : String
+getEmpno() : int <u>+getEmpcount() : int</u> +getLastname() : String +getFirstname() : String +setEmpno(in empno : int) : void <u>+setEmpcount(in empcount : int) : void</u> +setLastname(in lastname : String) : void +setFirstname(in firstname : String) : void

类的信息与基本实现

- 带有下划线的方法和属性表示静态方法和静态变量。
- 静态变量的使用需要仔细斟酌，因为破坏了面向对象的本地性（封装性）原则。
- 静态变量和静态方法通常用在一般性的常规工作，如记录文件的存储路径，常规的数值计算等（工具）功能。
- 方法参数的输入输出类型：**in, out, inout**。
- 上图中的类对应的[代码实现](#)。

实现上的补充

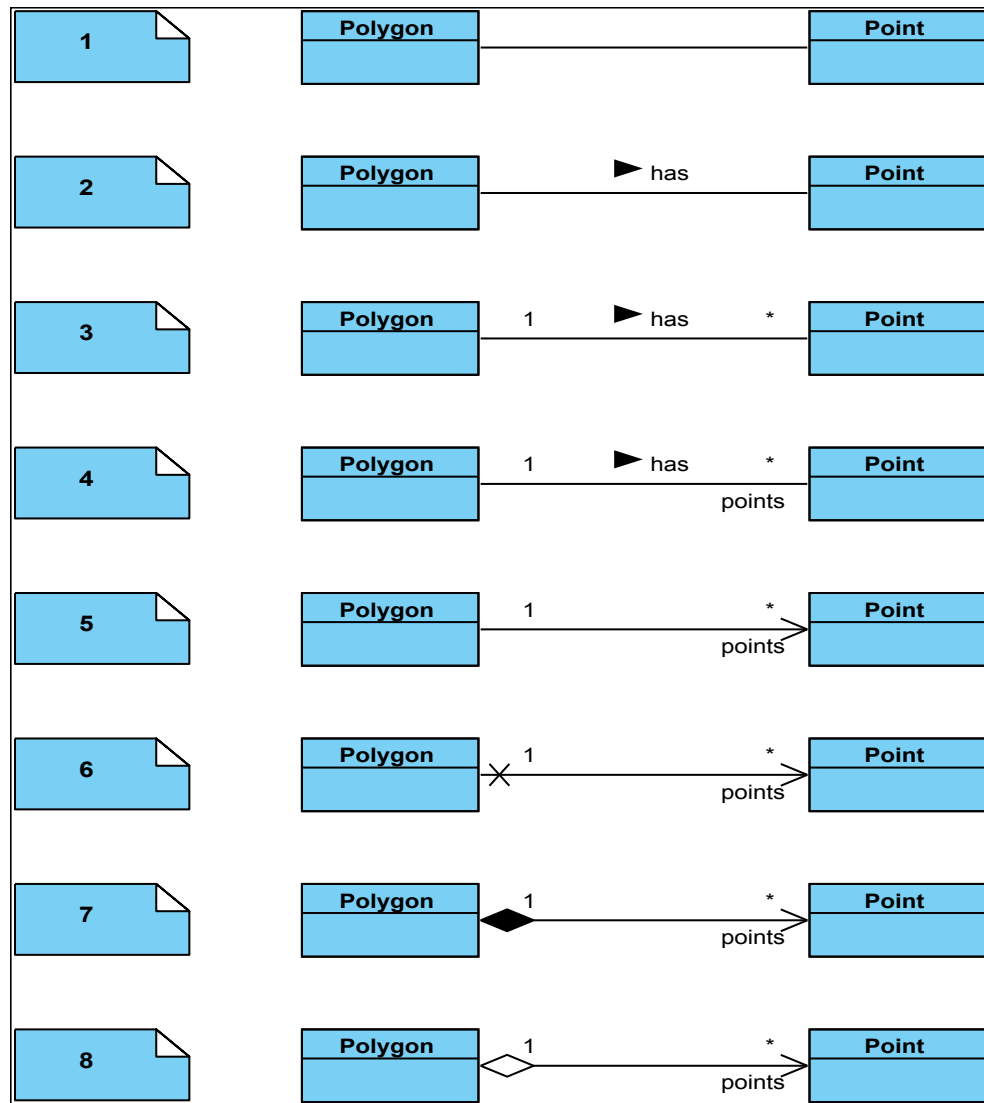
- CASE工具能够自动进行整个项目代码框架的生成，具体业务的代码一般需要人工补充。
- 业务越复杂，代码能够自动生成的可能性就越低，对于这部分需求还需要工具的开发为此付出大量的努力。

Employee
+Employee(in firstname : String, in lastname : String) +toString() : String

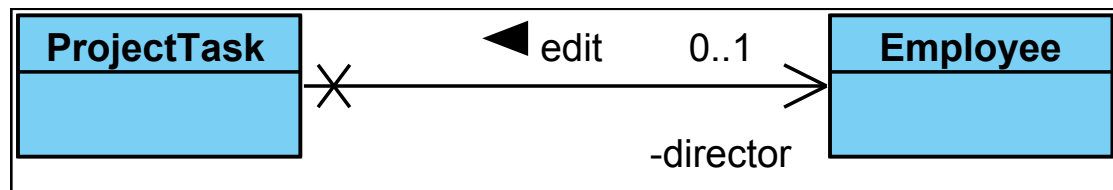
```
public Employee(String firstname, String  
lastname){  
    this.firstname = firstname;  
    this.lastname = lastname;  
    this.empno = Employee.empcount++;  
}  
@Override  
public String toString(){  
    return empno + ": " + firstname + " " +  
lastname;  
}
```


关联关系的实现

- 通过关联的定义明确了类与类之间的静态关系，关联关系的实现最终体现为对应类中增加的实例变量（成员变量）。
- 变量存在的具体形式依赖于关联的具体类型。



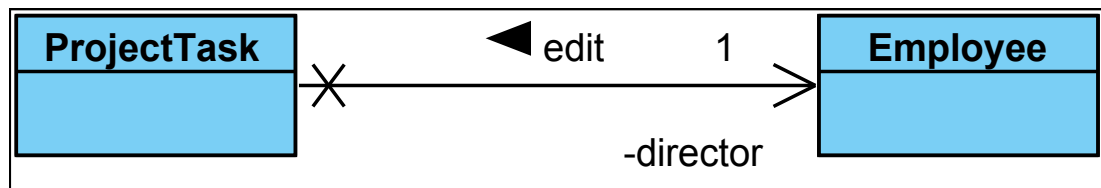
导航至“可选”方向



对应代码

- 多重性“0..1”明确了项目任务可以不分配工作人员，即在代码中**对应类的实例变量可以不赋任何值，在类构造时也可不必对该变量初始化**，在具体编程语言中通常通过一个对空值（NULL）的引用，不分配任何存储空间。
- 实例变量并不需要在声明时赋值，可以通过set方法在后期需要的时候赋值。
- 实例变量pincharge声明后可以一直为NULL值，但却不能通过任何的方法显式的对其赋予NULL值，这是隐含的业务规则。

导航至“唯一”方向

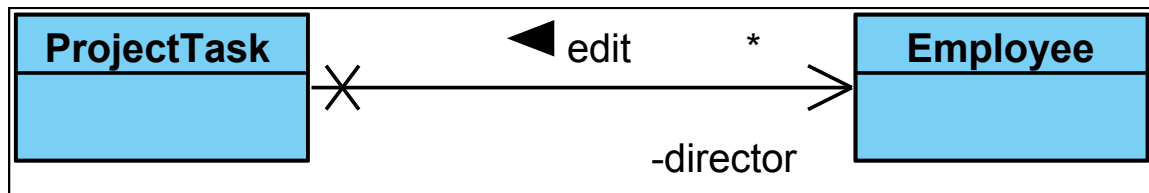


- 描述一个项目任务对象必须要有一位工作人员对应，也就不存在是否可以对该变量进行空值引用的问题。
- 为确保以上约定，可以在该实例变量声明时同时赋予初始值，可以按照如下的方式进行实现：

private Employee pincharge = new Employee();

- 该类的每个构造函数中对实例变量pincharge都需要指定一个有意义的值。这时可以将实例变量声明时的初始化去掉。

导航至“任意”方向

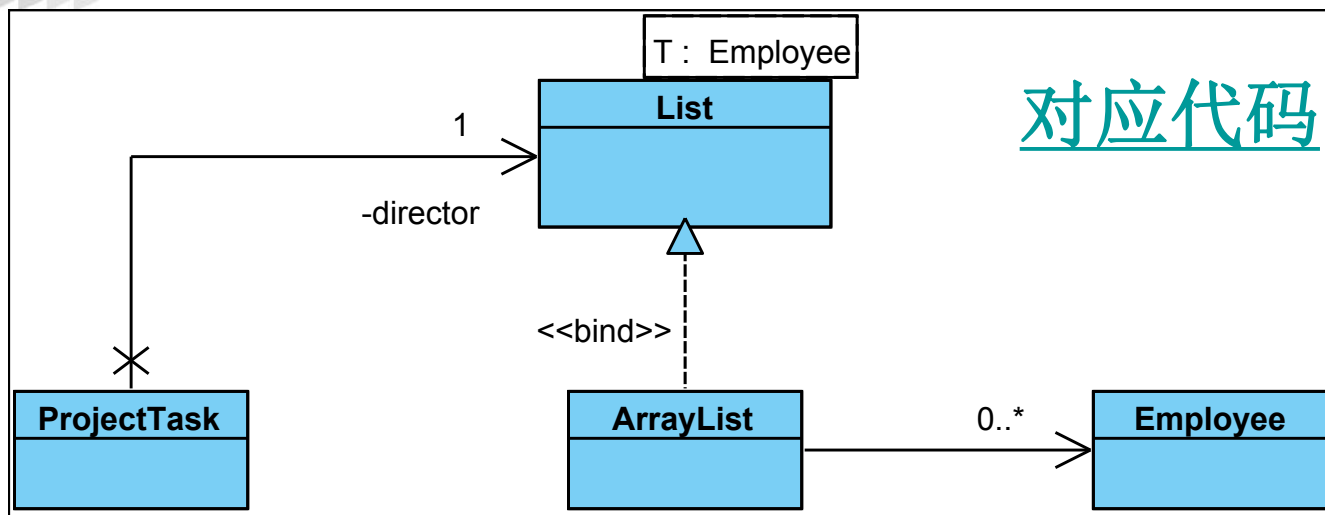


- 描述一个项目任务可以安排任意多个工作人员与之对应。
- 这些工作人员也是通过实例变量pincharge进行管理的，只是这时该变量的类型应为某种集合类型（集合类型在C++中也叫Container）。

基本的集合类型

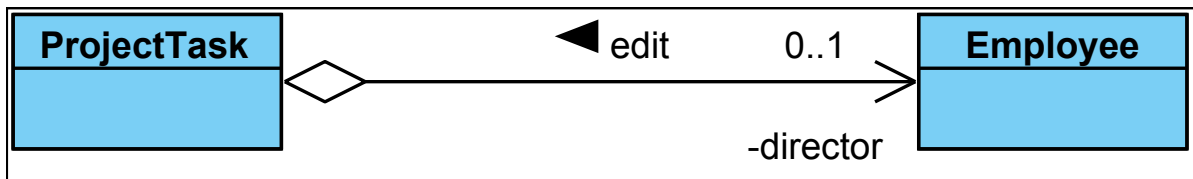
元素顺序 要求	元素唯一性 要求	集合类型
不要求	无重复 {unique}	集合类型Set, 其中的每个元素最多只能出现一次{unique}, 不要求元素的顺序性。
不要求	允许重复	集合类型Bag或Multiset, 其中的元素可多次重复出现, 不要求元素的顺序性。
要求 {ordered}	无重复 {unique}	集合类型OrderedSet, 所有元素具有顺序性{ordered}, 每个包含的元素最多只能出现一次{unique}, 元素依据其所在位置进行操作。
要求 {ordered}	允许重复	集合类型List或Sequence, 所有元素具有顺序性{ordered}, 每个元素允许重复, 并依据其所处位置进行操作。

使用List模板类的模型



- 在Java语言中，可以使用集合List接口类型对应允许元素重复的集合类型，并使用ArrayList进行实现。
- 在选用具体的集合类型时要仔细考虑，是否在运行时间和存储空间上能够真正满足要求并进行了最佳的实现。
- 比如List允许元素重复出现，如果这不是希望的，则应该在元素加入前进行目标元素是否已经存在的检查。

对象间归属——聚合



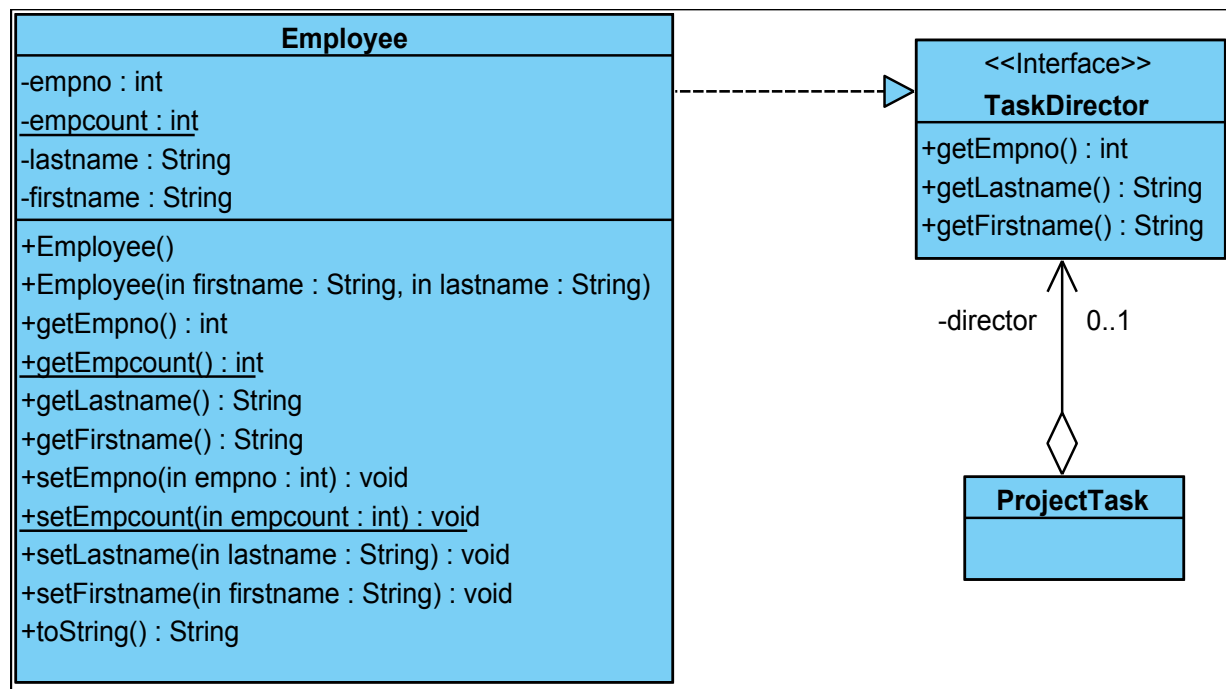
对应代码

- **Aggregation**
- 描述一个员工对象不是仅仅与一个项目任务对应，而是可以同时参与到多个项目任务，即在这些项目间共享。

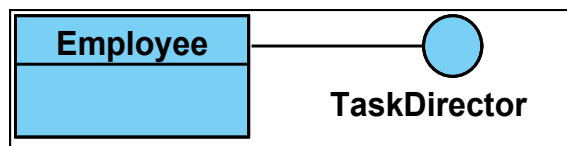
关联的问题

- 关联关系（包括聚合）的代码在一定程度上破坏了实例变量的封装性。
- 比如某个获得Employee对象引用的类就可以直接调用Employee的所有公共方法，从而可以直接对ProjectTask的实例变量进行修改。

使用接口的设计



- 接口中只含有 **ProjectTask**需要的方法，屏蔽 **Employee**中其它方法。
- 但通过强制类型转换（**Cast**）也可以从该接口转化为类型为 **Employee**的对象。



“棒棒糖”

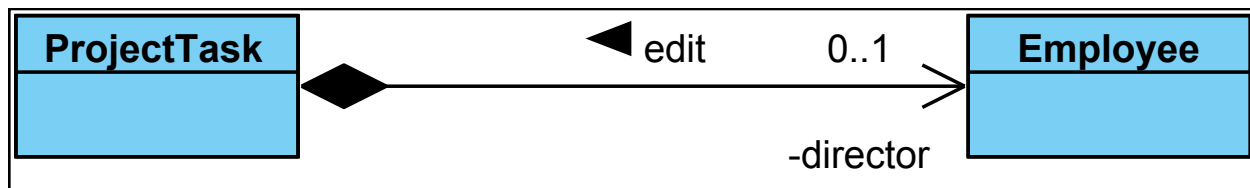
另一种设计

- 另外一种不破坏封装性的设计想法是类ProjectTask的方法不返回任何Employee类型的对象，这也意味着只有类ProjectTask允许对Employee的对象进行操作。
- 如需要对外提供修改员工对象的服务，则在类ProjectTask对应提供一具有修改能力的公共函数：

public void updateLastNameDirector(in lastname: String)

- 其实现是通过调用Employee类中的setLastname()函数并通过参数的传递完成实际的修改。这种方式的使用在聚合（Aggregation）关系中是很常用的。

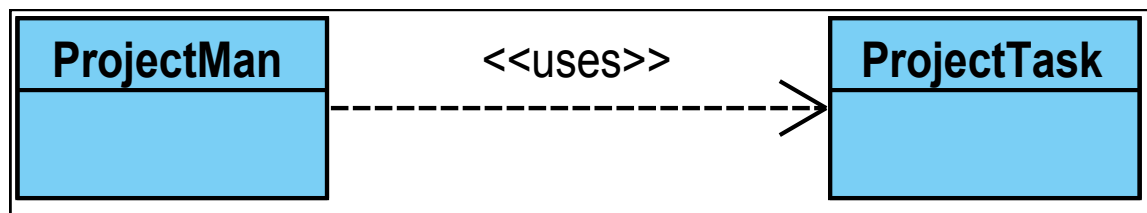
对象间归属——组合



对应代码

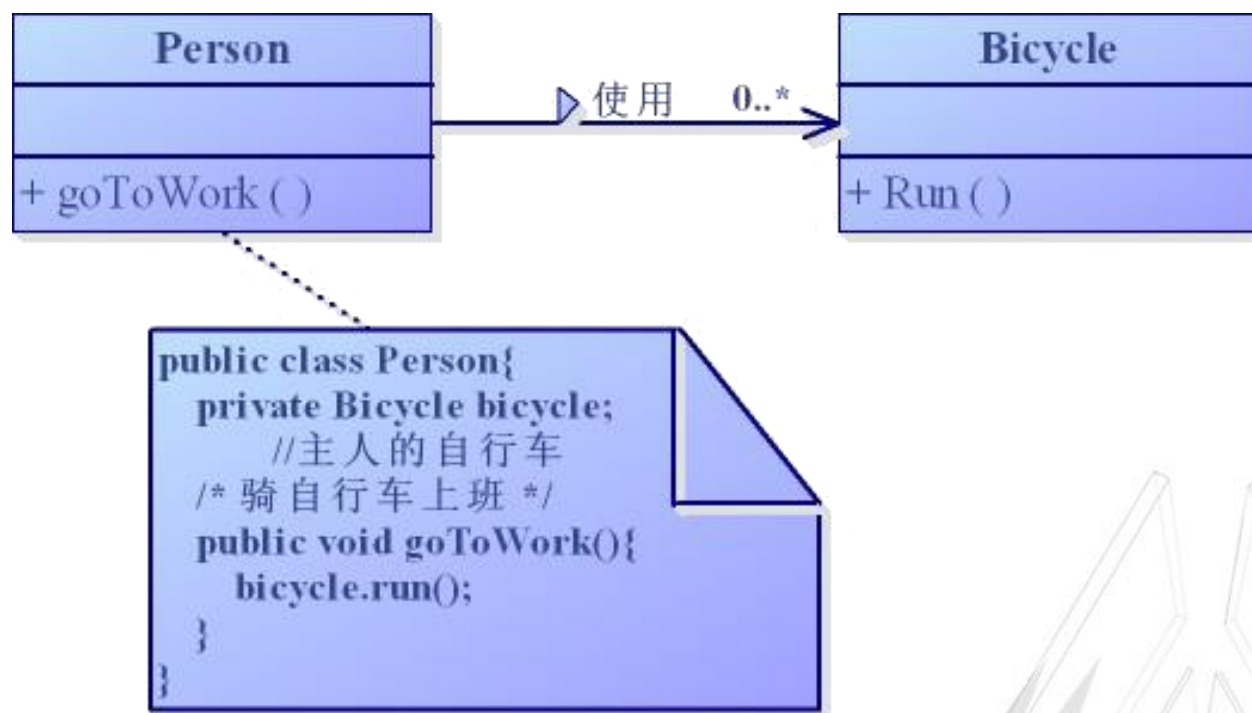
- **Composition**
- 描述两个类之间的一种“存在依赖性”，即如果类 **ProjectTask** 的对象被删除，其所属的所有工作人员会被同时删除。
- 这种情况可能的一种解释为：员工指代一种工作合同，通过该工作合同被雇佣，当任务合同结束后工作关系自动解除，合同也没有存在的必要了，会被同时删除。

对象间归属——依赖



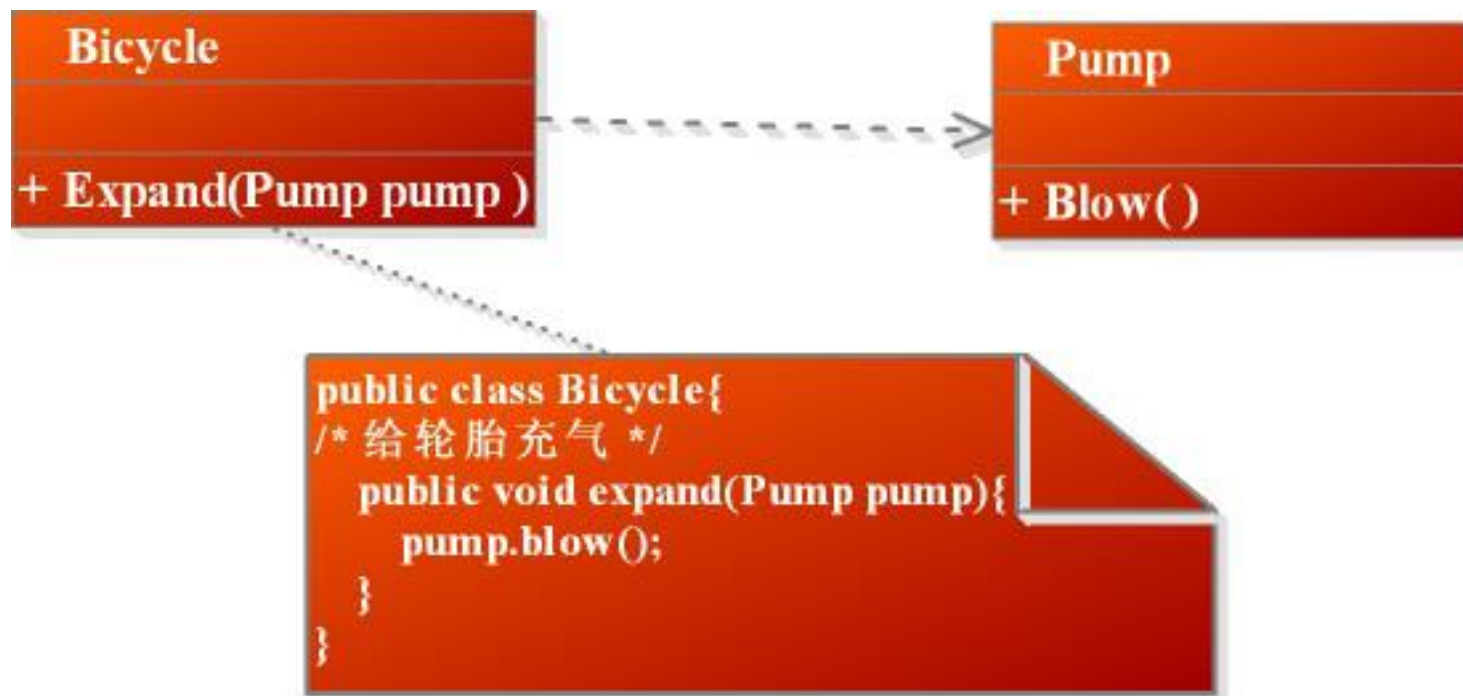
- 与静态的关联关系不同，依赖关系主要用来描述对象间访问的瞬时性，比如将某些类向其它类进行传递，这种瞬时关系并不在对象间保持
- 可以使用<<include>>代替<<uses>>
- 如果不影响类图的清晰性和阅读性，关联关系和依赖关系都可以在设计类图中说明，这种动态的瞬时关系在实现中的一种形式为（[对应代码](#)）

关联举例



关联关系

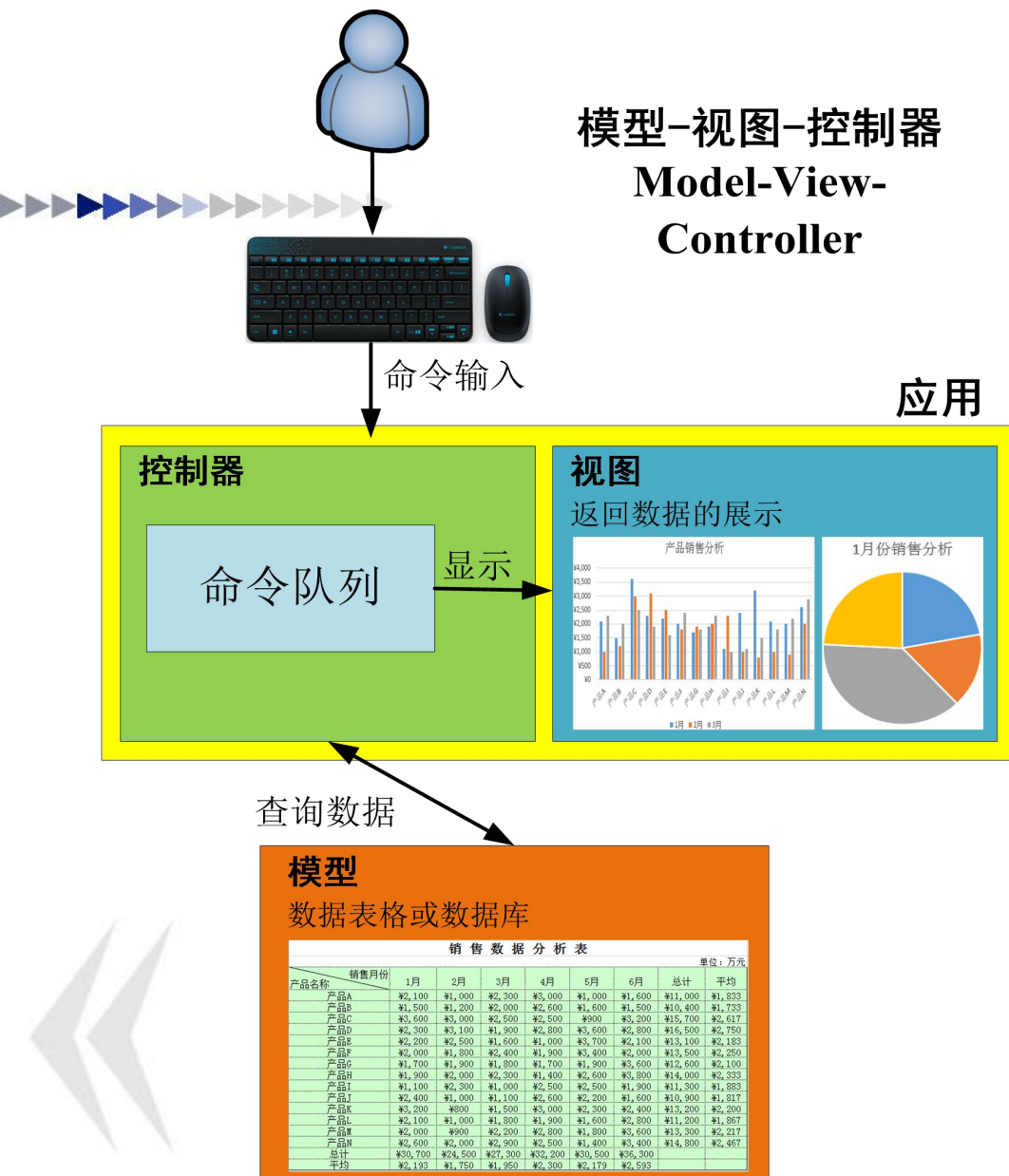
依赖举例



依赖关系

软件架构：MVC的实现

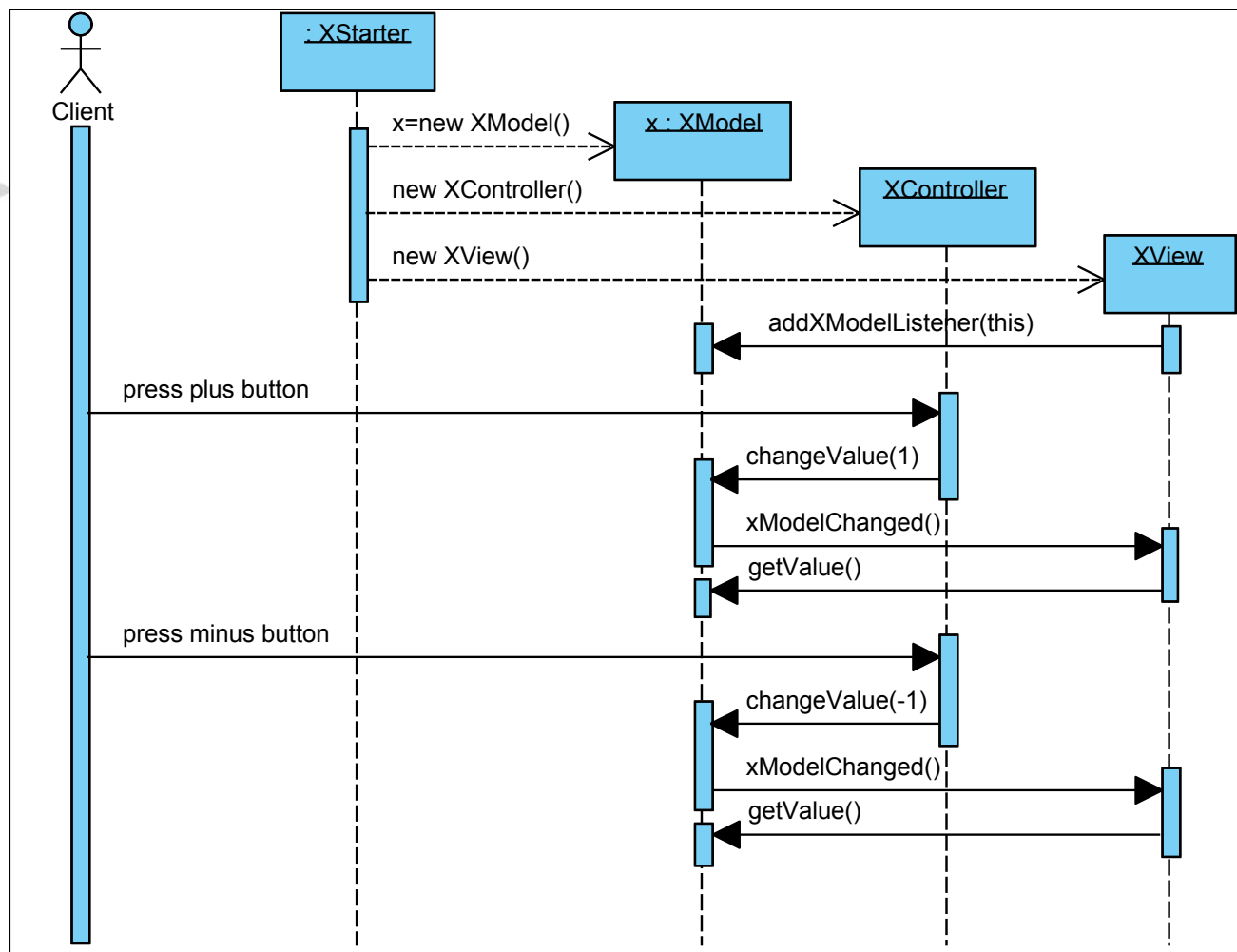
- MVC，Model-View-Controller模式，能够很好的对设计的灵活性进行解释。
- MVC的核心思想是将数据本身与其修改的方式以及数据的展现形式进行分离。
- 通过MVC提供的模式，使得数据能够以各种不同的修改方式进行处理，而不影响对数据的管理和对外展现的形式。
- 同时，系统中可以具有不同的数据展现方式，与其它组件是完全独立的。



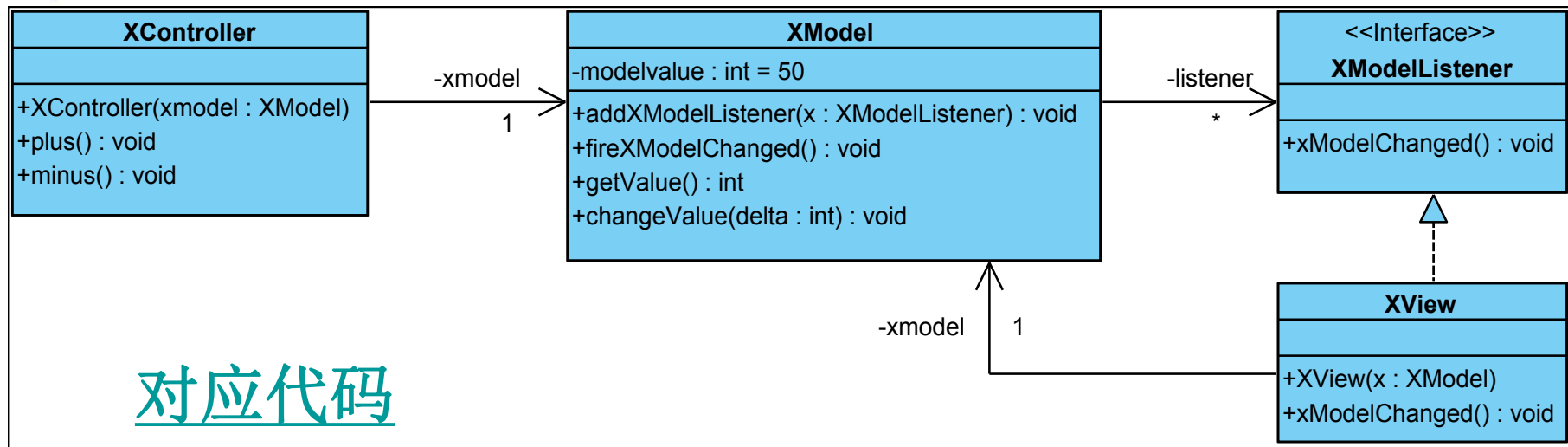
- 模型：业务数据实际的组织与存储。
- 视图：向外界显示结果。
- 控制器：改变模型中的值。

- 例子模型仅是一个简单类型的变量
- 为能够对该变量的值进行显示，上面的窗口作为一个视图用于以滑块的形式显示该变量的值
- 下方的窗口代表一个控制器，通过其中的两个按钮实现对模型中变量值的修改




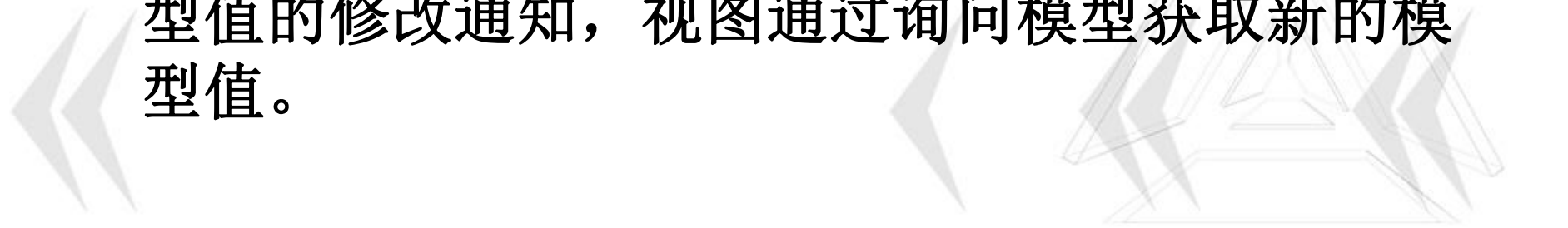


- MVC的使用是通过两个阶段进行的。
- 第一阶段，与MVC模式相关的对象被创建并进行关联；
- 第二阶段，实现对MVC结构的实际使用。



- 图中给出了一个适合MVC模式的类图，为了使得不同的视图能够在模型中进行注册以便日后方便进行通知，所有的视图都需要实现一个统一的接口，通过此接口能够对所有注册的视图进行统一的管理。

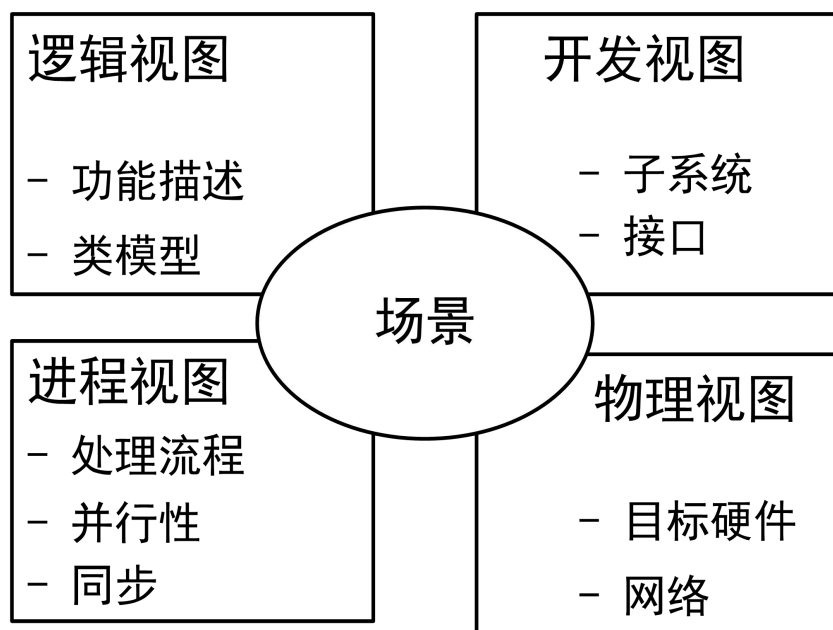
- 模式能够提供一个一般性的解决方案，但本质上并不强求每种实际的解决方案是完全一样的。
- ▶▶▶▶▶• 在模式的应用中可以根据需要进行调整，加入一些必要的设计方案上的变化，从而更加适合业务场景的需要。
- 对于MVC模式同样也存在一些考虑，比如：
- 如果确定只有一类视图对象在模型中注册，可以不提供接口XModelListener的使用。
- 如果仅存在一个视图，使用MVC来对各模块进行组织也是好的设计习惯。
 - 在这种情况下，模型实例变量listener不再是一个集合，而是一个简单的对视图的引用
 - MVC对于各部分的划分，有助于厘清各部分的职责

- 
- 有时候某个控制元素，比如按钮控件，也可以参与到输出的工作，如将结果输出为按钮的显示文本。这种情况下，控制器和视图就合二为一了。在Java的GUI库Swing中提供了所需的函数，如 **.JButton.setText()**。
 - 对于视图的管理可以不在模型中进行。经常会有这样的设计，即控制器负责所有视图的管理和模型值的修改通知，视图通过询问模型获取新的模型值。
- 

其他物理实现

- 类图主要用来对功能性需求进行建模，对非功能性需求的描述能力往往不足。
- 而且将所有信息都并入同一模型中进行展示也是不明智的，因为将导致模型本身的繁琐并难以阅读，软件系统各种信息应以不同的“视图”进行重点展示才更有意义。
- 视图表示对系统进行观察和理解的角度，每种角度专注于系统某方面的特征。
- 所有视图的集成保证最终实现的结果是全面而且成功的软件项目。

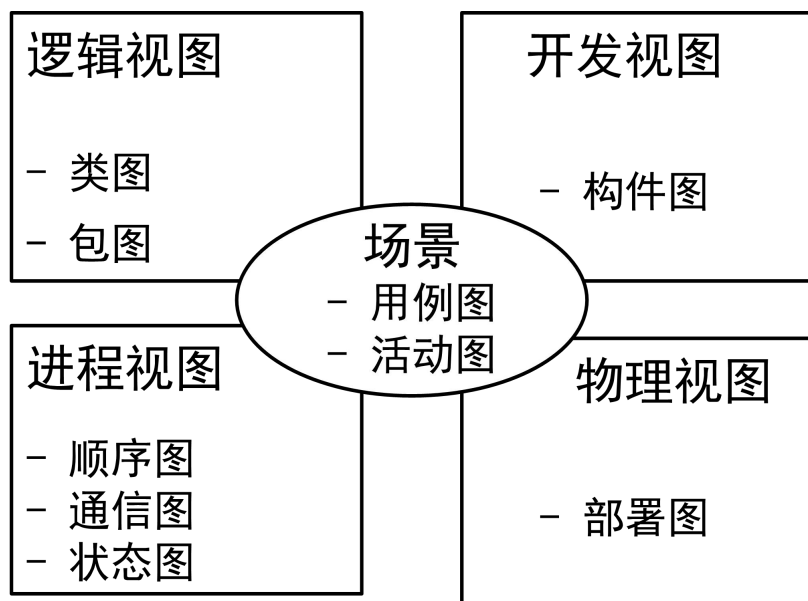
系统的4+1视图



- **逻辑视图**：支持面向对象的分解。逻辑架构主要用来支持功能性需求——满足用户服务的系统功能
- **进程视图**：进程的分解。进程架构考虑一些非功能性的需求，如性能和可用性
- **开发视图**：子系统的分解
- **物理视图**：软件至硬件的映射

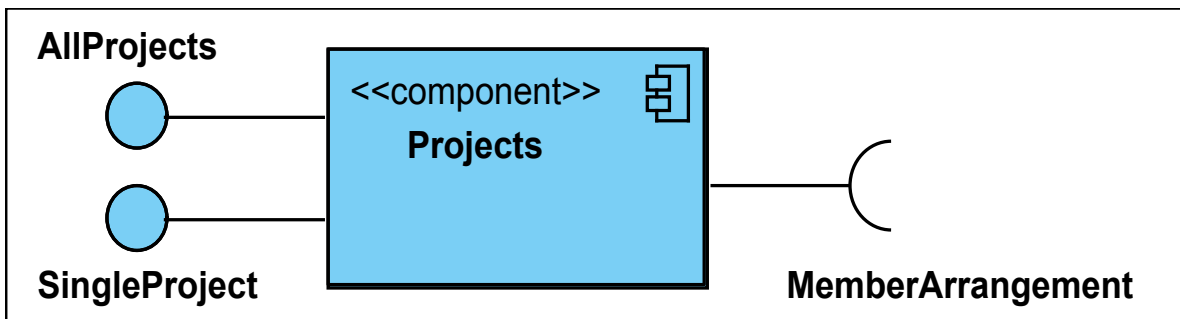
- **用例视图**：综合所有的视图。（+1）

UML的视图



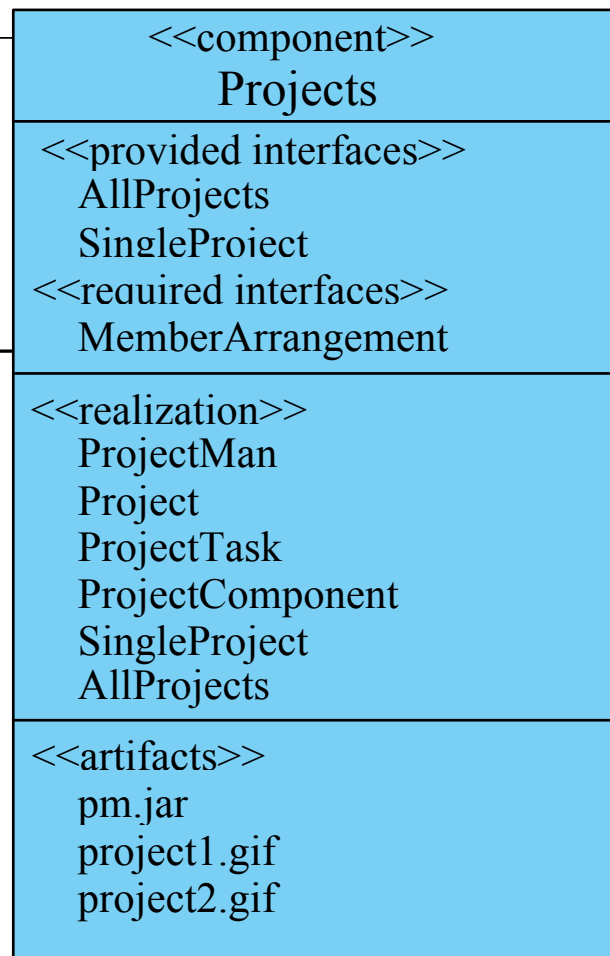
进程视图:

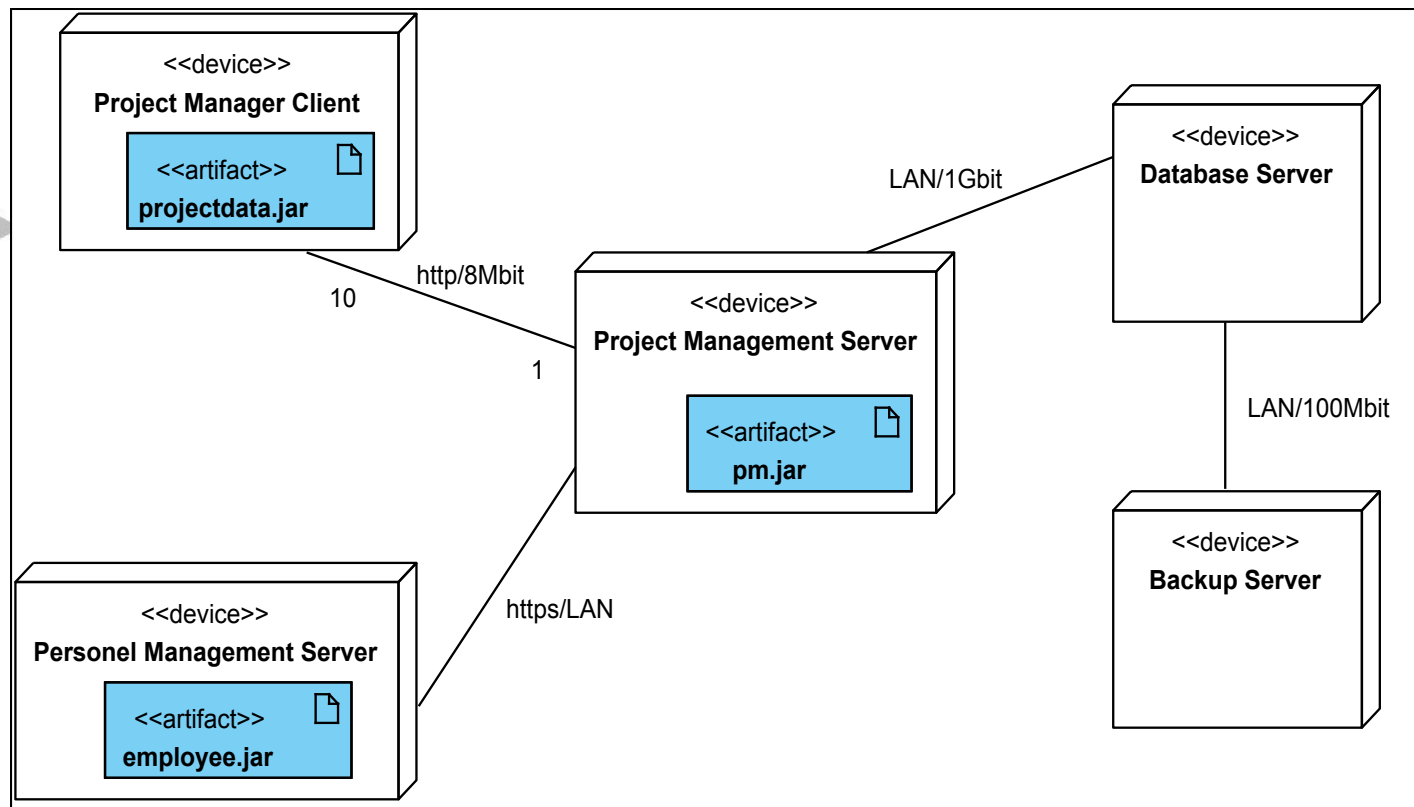
- 可能的独立运行的进程或者线程以及关联的活动对象。
- **Java**是从**Thread**类继承或**Runnable**接口实现的类，具有方法**run()**。
- **UML**中另外的描述使用构造型**<<thread>>**或**<<process>>**对活动类进行表示。



开发视图：

- 构件图能够描述多个构件的构成及它们之间的联系。
- 除了接口说明外，可以通过 **<<realization>>** 说明该构件中包含的类、其它工件 **<<artifact>>**、相关文件和资源等。





物理视图

- 部署图描述哪些软件构件最终在哪些机器上运行的情况
- 硬件节点、可运行程序<<executable>>和相关工件<<artifacts>>
- 网络连接数目

作业

- 习题1~2
- 习题3（选做）

