思考题：

1.面向对象设计阶段的输入和输出产物是什么？

**输入应该是客户的需求 输出数据流图+数据字典+需求分析文档**

2. 如何设计一个健壮的类层次结构？

**深层调用，ClassD1和ClassD2需要调用ClassA关联的ClassX、ClassY和ClassZ等，对于这种情况，经常见到通过构造函数一层层往下传递做法。采取两种方法：**

**1.尽量让ClassA成为一个单例，这样ClassD要获取ClassX等就非常方便了，即使增加一个ClassX1也非常方便，符合开闭原则，简单明了；**

**2.但并不是每种情况下，都允许ClassA成为单例，这个时候采用第二种办法，即总是通过构造函数将ClassA往下传递，如ClassB(ClassA\*)；ClassC(ClassA\*)；ClassD(ClassA\*)，这种办法也是符合开闭原则的，再增加一个ClassX1也非常方便；**

3. 自己找一段代码，分析它是如何遵守Demeter法则。

**Demeter法则基本介绍**

**一个对象应该对其他对象保持最少的了解**

**类与类的关系越密切，耦合度越大**

**Demeter法则又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管有多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的public犯法，不对外泄露任何信息**

**Demeter法则还有个更简单的定义：只与直接的朋友通信**

**直接的朋友：每个对象都会有与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖，关联，组合，聚合等。其中，我们称出现在成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友。也就是说，陌生的类最好不要以局部变量的形式出现在类的内部。**

**应用实例**

**要求：有一个学校，下属有各个学院和总部，现要求打印出学校总部员工ID和学院员工的id**

package com.atguigu.principle.demeter;

import java.util.ArrayList;

import java.util.List;

//客户端

public class Demeter1 {

public static void main(String[] args) {

//创建了一个 SchoolManager 对象

SchoolManager schoolManager = new SchoolManager();

//输出学院的员工id 和 学校总部的员工信息

schoolManager.printAllEmployee(new CollegeManager());

}

}

//学校总部员工类

class Employee {

private String id;

public void setId(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

//学院的员工类

class CollegeEmployee {

private String id;

public void setId(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

//管理学院员工的管理类

class CollegeManager {

//返回学院的所有员工

public List<CollegeEmployee> getAllEmployee() {

List<CollegeEmployee> list = new ArrayList<CollegeEmployee>();

for (int i = 0; i < 10; i++) { //这里我们增加了10个员工到 list

CollegeEmployee emp = new CollegeEmployee();

emp.setId("学院员工id= " + i);

list.add(emp);

}

return list;

}

}

//学校管理类

//分析 SchoolManager 类的直接朋友类有哪些 Employee、CollegeManager

//CollegeEmployee 不是 直接朋友 而是一个陌生类，这样违背了 迪米特法则

class SchoolManager {

//返回学校总部的员工

public List<Employee> getAllEmployee() {

List<Employee> list = new ArrayList<Employee>();

for (int i = 0; i < 5; i++) { //这里我们增加了5个员工到 list

Employee emp = new Employee();

emp.setId("学校总部员工id= " + i);

list.add(emp);

}

return list;

}

//该方法完成输出学校总部和学院员工信息(id)

void printAllEmployee(CollegeManager sub) {

//分析问题

//1. 这里的 CollegeEmployee 不是 SchoolManager的直接朋友

//2. CollegeEmployee 是以局部变量方式出现在 SchoolManager

//3. 违反了 迪米特法则

//获取到学院员工

List<CollegeEmployee> list1 = sub.getAllEmployee();

System.out.println("------------学院员工------------");

for (CollegeEmployee e : list1) {

System.out.println(e.getId());

}

//获取到学校总部员工

List<Employee> list2 = this.getAllEmployee();

System.out.println("------------学校总部员工------------");

for (Employee e : list2) {

System.out.println(e.getId());

}

}

}

**应用Demeter法则修改后的代码**

**将输出学院的员工方法，封装到CollegeManager，使得SchoolManager的陌生类CollegeEmployee 不再出现在 SchoolManager类中。**

package com.atguigu.principle.demeter.improve;

import java.util.ArrayList;

import java.util.List;

//客户端

public class Demeter1 {

public static void main(String[] args) {

System.out.println("~~~使用迪米特法则的改进~~~");

//创建了一个 SchoolManager 对象

SchoolManager schoolManager = new SchoolManager();

//输出学院的员工id 和 学校总部的员工信息

schoolManager.printAllEmployee(new CollegeManager());

}

}

//学校总部员工类

class Employee {

private String id;

public void setId(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

//学院的员工类

class CollegeEmployee {

private String id;

public void setId(String id) {

this.id = id;

}

public String getId() {

return id;

}

}

//管理学院员工的管理类

class CollegeManager {

//返回学院的所有员工

public List<CollegeEmployee> getAllEmployee() {

List<CollegeEmployee> list = new ArrayList<CollegeEmployee>();

for (int i = 0; i < 10; i++) { //这里我们增加了10个员工到 list

CollegeEmployee emp = new CollegeEmployee();

emp.setId("学院员工id= " + i);

list.add(emp);

}

return list;

}

//输出学院员工的信息

public void printEmployee() {

//获取到学院员工

List<CollegeEmployee> list1 = getAllEmployee();

System.out.println("------------学院员工------------");

for (CollegeEmployee e : list1) {

System.out.println(e.getId());

}

}

}

//学校管理类

//分析 SchoolManager 类的直接朋友类有哪些 Employee、CollegeManager

//CollegeEmployee 不是 直接朋友 而是一个陌生类，这样违背了 迪米特法则

class SchoolManager {

//返回学校总部的员工

public List<Employee> getAllEmployee() {

List<Employee> list = new ArrayList<Employee>();

for (int i = 0; i < 5; i++) { //这里我们增加了5个员工到 list

Employee emp = new Employee();

emp.setId("学校总部员工id= " + i);

list.add(emp);

}

return list;

}

//该方法完成输出学校总部和学院员工信息(id)

void printAllEmployee(CollegeManager sub) {

//分析问题

//1. 将输出学院的员工方法，封装到CollegeManager

sub.printEmployee();

//获取到学校总部员工

List<Employee> list2 = this.getAllEmployee();

System.out.println("------------学校总部员工------------");

for (Employee e : list2) {

System.out.println(e.getId());

}

}

}

4. 总结JAVA中的抽象类和接口，写简单例子理解特性。

#### 1）抽象类

**在 Java 中，被关键字 abstract 修饰的类称为抽象类；被 abstract 修饰的方法称为抽象方法，抽象方法只有方法声明没有方法体。  
  
抽象类有以下几个特点：**

1. **抽象类不能被实例化，只能被继承。**
2. **包含抽象方法的类一定是抽象类，但抽象类不一定包含抽象方法（抽象类可以包含普通方法）。**
3. **抽象方法的权限修饰符只能为 public、protected 或 default，默认情况下为 public。**
4. **一个类继承于一个抽象类，则子类必须实现抽象类的抽象方法，如果子类没有实现父类的抽象方法，那子类必须定义为抽象类。**
5. **抽象类可以包含属性、方法、构造方法，但构造方法不能用来实例化对象，只能被子类调用。**

#### 2）接口

**接口可以看成是一种特殊的类，只能用 interface 关键字修饰。  
  
Java 中的接口具有以下几个特点：**

1. **接口中可以包含变量和方法，变量被隐式指定为 public static final，方法被隐式指定为 public abstract（JDK 1.8 之前）。**
2. **接口支持多继承，即一个接口可以继承（extends）多个接口，间接解决了 Java 中类不能多继承的问题。**
3. **一个类可以同时实现多个接口，一个类实现某个接口则必须实现该接口中的抽象方法，否则该类必须被定义为抽象类。**

#### 3）抽象类和接口的区别

**接口和抽象类很像，它们都具有如下特征。**

* **接口和抽象类都不能被实例化，主要用于被其他类实现和继承。**
* **接口和抽象类都可以包含抽象方法，实现接口或继承抽象类的普通子类都必须实现这些抽象方法。**

**但接口和抽象类之间的差别非常大，这种差别主要体现在二者设计目的上。下面具体分析二者的差别。  
  
接口作为系统与外界交互的窗口，接口体现的是一种规范。对于接口的实现者而言，接口规定了实现者必须向外提供哪些服务（以方法的形式来提供）；对于接口的调用者而言，接口规定了调用者可以调用哪些服务，以及如何调用这些服务（就是如何来调用方法）。当在一个程序中使用接口时，接口是多个模块间的耦合标准；当在多个应用程序之间使用接口时，接口是多个程序之间的通信标准。  
  
从某种程度上来看，接口类似于整个系统的“总纲”，它制定了系统各模块应该遵循的标准，因此一个系统中的接口不应该经常改变。一旦接口被改变，对整个系统甚至其他系统的影响将是辐射式的，会导致系统中大部分类都需要改写。  
  
抽象类则不一样，抽象类作为系统中多个子类的共同父类，它所体现的是一种模板式设计。抽象类作为多个子类的抽象父类，可以被当成系统实现过程中的中间产品，这个中间产品已经实现了系统的部分功能（那些已经提供实现的方法），但这个产品依然不能当成最终产品，必须有更进一步的完善，这种完善可能有几种不同方式。**

**门和警报的例子：门都有open( )和close( )两个动作，此时我们可以定义通过抽象类和接口来定义这个抽象概念：**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **abstract class Door {**  **public abstract void open();**  **public abstract void close();**  **}** |

**或者：**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **interface Door {**  **public abstract void open();**  **public abstract void close();**  **}** |

**但是现在如果我们需要门具有报警alarm( )的功能，那么该如何实现？下面提供两种思路：**

**1）将这三个功能都放在抽象类里面，但是这样一来所有继承于这个抽象类的子类都具备了报警功能，但是有的门并不一定具备报警功能；**

**2）将这三个功能都放在接口里面，需要用到报警功能的类就需要实现这个接口中的open( )和close( )，也许这个类根本就不具备open( )和close( )这两个功能，比如火灾报警器。**

**从这里可以看出， Door的open() 、close()和alarm()根本就属于两个不同范畴内的行为，open()和close()属于门本身固有的行为特性，而alarm()属于延伸的附加行为。因此最好的解决办法是单独将报警设计为一个接口，包含alarm()行为,Door设计为单独的一个抽象类，包含open和close两种行为。再设计一个报警门继承Door类和实现Alarm接口。**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20** | **interface Alram {**  **void alarm();**  **}**    **abstract class Door {**  **void open();**  **void close();**  **}**    **class AlarmDoor extends Door implements Alarm {**  **void oepn() {**  **//....**  **}**  **void close() {**  **//....**  **}**  **void alarm() {**  **//....**  **}**  **}** |

5. 写一个完整的从需求到设计模型的过程

**一、需求分析阶段**

**主要是跟客户（领域专家）沟通，进行需求的收集和分析，然后通过标准的文书准确地表达出来，并形成需求规格说明书之类的文档，交由设计人员进行后续的系统设计工作。**

**UML中的**[**用例图**](https://so.csdn.net/so/search?q=%E7%94%A8%E4%BE%8B%E5%9B%BE)**正是用于需求收集和表达的有力工具，但是如何找出用例并非易事，这是因为从用户那里收集来的信息很可能是零散的、没有系统性的，要直接从中找出正确的用例非常困难。**

**因此在分析用例之前，可以先对企业级的业务流程进行规划和设计，抓住企业的本质工作流，为后续进行详细的需求收集和用例分析做好准备。**

## 1、业务流程设计

**对于企业的经营管理团队来说，业务流程规划与企业的永续经营之间存在着密切关系。简单来说，业务流程就是为了服务客户而执行的一连串业务内部活动。业务流程分析的目的在于了解整体流程对企业目标的支持分别有何贡献，进而对流程的细节进行规划。**

**那么如何进行业务流程的设计呢？Jacbson认为，利用“用例”的“目标导向”特性，可以通过一个“企业级的用例”来完善工作流程的规划与设计。不过衡量实际状况，大部分领域专家对“用例”的接受度较差，因此可以使用另一个工具来进行企业的建模，这个工具是由Erickson和Penker所提出的一个活动图的构造型，称为“Eriksson-Penker业务扩展模型”。**

### 1）业务流程规划——Eriksson-Penker业务扩展模型

**Eriksson-Penker业务扩展模型是一种“目标导向”的流程分析方式，主要是将与业务流程相关的重要人、事、物以及这个业务流程所要实现的目标做一个链接，描述了企业中重要的人、事、物与流程的关系，这个图中通常不会过多地介绍业务流程的内部细节。在项目开始阶段，需求分析人员可以通过“Eriksson-Penker业务扩展模型”找出要开发系统的重要性，利用“目标导向”方式，对业务流程进行适当的切割。**

### 2）业务流程分析——活动图

**在与领域专家进一步沟通后，就可以对“Eriksson-Penker业务扩展模型”中的每一个“处理”绘制一个对应的活动图，在绘制活动图时，应该将重点放在“活动”本身，而不需要加入其他因素（文件、数据、表单等）。在活动图中，这些因素应该要在上层的“Eriksson-Penker业务扩展模型”就表达完成。**

**活动图最适合用来描述企业的本质工作流。在绘制活动图时千万不要去研究活动的细节，活动图所要捕捉的是整体业务流程的“大方向”。有关细节的相关描述应该是在讨论“用例”时才需要捕捉。**

**在完成各主要业务流程的分析，绘制出活动图以后，便可以开始下个分阶段的工作——从业务流程中找出用例，进行需求收集，完成用例模型。**

## 2、需求收集——用例图

### 1）关于用例的相关介绍

**用例是一个系统中所进行的一连串的处置活动，该活动主要是要能够满足系统外部的执行者对于系统的某种期望。  
每一个信息系统的用例代表着用户对于系统的“某一个完整期望”。  
通常来说，用例是“需求收集及整理”的工具，通过用例与执行者的关系，可以让需求分析人员“聚焦”在特定的“相关人员”（也就是执行者）与”主题“（也就是用例）中。**

### 2）找出用例的三个步骤

**根据前面所绘制的业务流程的活动图，可以通过以下三个步骤找出用例：**

#### ① 利用与用户的对话找出信息系统的用例

**将活动图中的每个“活动”当作“用例”的候选，接着针对每个”活动“询问用户以下几个问题：**

* **在这个活动中谁是主要参与者？**
* **这个活动的进行中需要系统提供服务吗？**
* **系统需要提供什么服务？**
* **系统需要其他信息系统的支持吗？**

**然后对候选用例进行必要的合并和关系（比如“包含”）分析， 从而得出业务流程相关的用例图。**

#### ② 完成用例的正常流叙述

**编写用例叙述时遵循的原则：**

* **每个叙述都必须是肯定句**
* **在叙述中，切记不要描述过多细节**

#### ③ 完成用例的替代流及意外处理叙述

**替代流本身仅仅只是正常流的“分支”而非“主干”。举例来说，如果在正常流2有三个替代流，则在替代流的区块中，就会有2a、2b、2c三个分支，而在这三个分支的编写中，仍然必须遵循着每一句都是“肯定句”的原则。如果在其中又有替代流，则一样必须要利用分支的方式来编写。这样，由于每个叙述都是简短的肯定句，自然而然增加了未来的扩展空间。**

### 3）关于用例的用例叙述

#### 用例的叙述一般来说至少分成四种：

* **用例的简述：通常是用一两句话来说明这个用例的目的是什么。**
* **用例的正常流：在这个流程中，必须说明执行者与系统交互的过程，不过在这个交互过程中，必须假设整个流程都必须实现，也就是说这是一个“快乐路径”，在这个流程描述中，所有句子都必须是“肯定句”。**
* **用例的替代流：在正常流中，如果有“替代路径”，必须要利用另外的替代流来说明，而不是直接在正常流中写“if-then-else“。**
* **用例的意外处理：通常指系统例外状态的处理，与替代流不同，替代流往往是执行者对于流程有不同的指示，因为将流程导向不同的结束点，而意外处理则通常是系统发生错误导致的正常流的意外状况。**

**用例的叙述是非常关键的部分，必须能够准确地把握用户的真正期望是什么，后续的设计工作都将围绕用例特别是用例叙述来展开。**

### 4）编写用例的测试案例

**一般来说，在找出用例后就应该编写用例的测试案例，测试案例的编写主要利用“输入→预期产出“的方式来描述，每个测试案例都需要准备对应的测试数据。**

# 二、系统设计阶段

**前一阶段的主要产物是用例图，后续的设计和开发阶段都将以用例驱动，围绕用例展开，而系统设计阶段的主要工作，便是实现用例。**

## 1、实现用例

**实现用例的目的在于保证系统的设计可以满足用户的功能性需求，在实现用例的过程中，应该利用Jacobson所分类的三种分析类：（注意：这里的“对象”并非指类的实例那种对象）**

* **控制对象（Control Object）     ：控制对象包装了一个或多个用例的功能性需求，属于功能性对象，而且这个功能与用例有相当密切的关系。**
* **实体对象（Entity Object）        ：实体对象管理了信息及其衍生资源的存取，是属于系统本质面的概念性对象，这类对象并不会随着用例的增多而有所变动。**
* **边界对象（Boundary Object）  ：边界对象是属于与外部桥接的对象，这类对象将与外部直接接轨，直接受到外部的限制。**

### 1）勾勒用例的控制对象

**①  针对每个用例提供一个“控制对象”  
②  明确这个控制对象的责任（Responsibility）是什么  
      从“主执行者”在正常流的叙述中出现的次数来决定系统要提供几个服务；  
      再从每一个“对话块”中，“系统”当主语的最后一句话，找出这个责任的名称。  
③  明确这个服务的输入输出  
      判断这个服务中，是否需要“主执行者”提供什么信息，而“系统”又需要回复主执行者什么信息  
④  进入到服务内部，审视服务的实现方式  
      在控制对象的内部，每一个以“系统”当主语的叙述都可以独立成一个新的功能函数；  
      只是该功能函数并非是提供给主执行者的，因此是一个“私有”的函数，只提供给控制对象使用。**

### 2）针对控制对象绘制序列图

**前面探讨了如何找出信息系统中所需的控制对象，但这样仍然不够，因为前面并没有完整描述出究竟对象与对象之间是如何通力协作，来满足用例所描述的用户需求。因此，必须要使用序列图来说明这个交互过程。**

**在绘制序列图时，可以采用两阶段序列图绘制法：**

**① 把信息系统当黑箱，利用用例叙述找出系统所应负责的服务。**

**这个步骤可以先绘制一个序列图，然后把用例叙述放在该序列图的右方（这样便于对比），然后参照用例图，把相对应的用例转换为一个叫做“系统”的对象。**

**② 把黑箱打开，加入找出的分析对象，并把系统所需实现的责任分配给适当的对象。**

**把上个步骤得到的“黑箱”序列图中的“系统”换成实际的控制对象，并且依据找出的控制对象的责任，看看是否一致，这样就完成了序列图的设计了。**

### 3）找出用例的实体对象

**可以通过Peter Coad的“交易模式”找出用例的实体对象，这个模式的假设是：当发现企业所关心的问题领域存在必须要记录的某些事件时，这代表着这个事件是一个交易。而系统设计人员可以从交易出发，依次去找出与这些交易相关的企业概念（人、地、物），如此就可以迅速地得出这个企业的概念模型。**

**总之，实体对象主要是根据对于问题领域的理解来找出问题领域中的重要概念，对于实体对象的分析，无论是对于进行“实体关系图的”的数据库设计，或是利用“对象模型”做的“结构分析”来说，都是相当重要的设计准则。**

**实体对象属于领域模型的重要概念，将在下一节“建立领域模型”中重点讲解。**

### 4）系统设计阶段的开发流程

**①  通过对用例的理解以及对用例叙述的分析，找出系统的控制对象及其操作。**

**②  通过与领域专家的访谈过程，找出系统的实体对象以及重要熟悉。**

**③  设计人员利用两阶段绘制的序列图，验证前述的控制对象及操作的正确性。**

**前面通过三种分析类实现用例的方式，会从用例出发分别找出控制对象、实体对象和边界对象，在找出这些“对象”（这里的对象并非指类的实现，而是指一种分析类）之后，便可以建立完整的“领域模型”了。**

## 2、建立领域模型

### 1）“领域模型”的概念

**要了解领域模型，就要先了解何为软件的“本质”：“本质”指得就是要想办法直指想要解决的问题的“核心”。**

**从软件结构的层面来看，“本质”指的就是你所要解决的问题领域中的重要“概念”在抽象层次的呈现。一般来说，这样的呈现方式的会通过“概念模型”来表示。  
“概念模型”就是能够用最简化的方式表达一个完整的“问题领域”的抽象表示法。概念模型的原始定义是表达问题领域中的概念，因此，通常将概念模型称为“领域模型”。**

### 2）使用类图表达领域模型

**在UML中通常建议使用“类图”作为表达领域模型的图形。  
类图主要表达的是问题领域的“抽象概念”，在这个抽象概念中，除了表达该抽象概念的名称外，另外需要表达该抽象概念的“属性”与”行为”。  
类图的主要目的是在进行软件开发前，先对软件所需面对问题领域的本质作一个通盘性的了解，但类图在软件设计之初并不完全正确，必须通过后续的检查才能够逐渐趋近于真实世界的领域模型。**

## 3、表达对象交互

**一般来说，我们在用例分析中将系统应该满足的用户期望找出来了；而在类图中则将系统的架构构造出来。但是，针对每个特定的用例的场景，要如何利用类图所规范的对象，通过交互协作来完成用例所交付的任务，就必须要用序列图来表达。**

### 1）序列图

**序列图的主要目的在陈述用例的正常事件流中，对象彼此之间的交互关系。也就是说，序列图的主要来源是用例的叙述。**

序列图的主要任务包括：

* **表达设计人员心中关于将来程序在运行时的对象协作模型**
* **验证软件领域模型的正确性**
* **为程序员提供编码的蓝图**

### 2）通信图

**通信图与序列图其实都是在表达同一件事情：对象相互合作，以实现用例的“事件流”。**

为什么要使用通信图进一步验证呢？

**由于序列图是以时间做横轴，因此对未来的程序设计而言，序列图具有“蓝图”的效果，但如果需要同时表达对象的结构与彼此间的协作关系，则只有通信图才能较为完整地进行呈现。**

**究竟项目设计人员在设计序列图时，心中是否对象模型，因此希望项目设计人员能利用“通信图”来重新审视自己对对象模型的理解，来确认序列图有没有违反领域模型。**

### 3）交互概述图

在绘制序列图和通信图等交互图时，需要注意：

* **不能“务求精细”过于详尽，因为交互图只需要描述一个“蓝图”而不是完整的“”施工计划；**
* **一张交互图不能太大，最好能在一张A4纸的可以打印的范围内，顶多一张A3纸，否则会成为无效的产出；**
* **每个交互图应该有表达的重点，不要在一个图中表达所有的逻辑，如果有替代流，那么就针对一个替代流再绘制一个单独的交互图。**

**那么，这些分散的交互图怎么才能组合在一起呢？这时可以利用交互概述图。**

**交互概述图主要是利用活动图作为基础，只是在“控制流”间连接的UML元素并非活动，而是交互图（包括：序列图、通信图、时间图以及交互概述图）。**

## 4、表达微观设计

### 1）对象图

**对象图旨在描述特定时间点中所有对象在系统中的结构；因此，可以将对象图当成系统在某一个时间点的快照。**

**对象图表达的是在某一个特定时间点中，系统所存在的所有对象的快照，其主要目的是验证设计师设计的类图是否符合实际状况。**

对象图的使用场景： **当与领域专家沟通时，可以用对象图解释类图的设计，以验证类图的正确性。  
当与编码人员沟通时，可以利用部分的对象图，来解释类图中的复杂结构。**

### 2）状态机图

**类图中某一个实体对象，它的状态迁移分散在不同的用例中，需要在这些状态和事件之间进行一番整理，才能让项目开发人员更简便地完成设计，这时可以使用状态机图来表达。  
为了成功地设计软件，将“状态”分配到不同的“领域模型”中，并利用“状态机图”来表达这些状态的迁移情形。**

### 3）时间图

**如果在状态迁移中牵涉到时间因素，则可以利用时间图来强调事件因素的重要性。设计人员可以把时间图当成状态机图的辅助说明工具。**