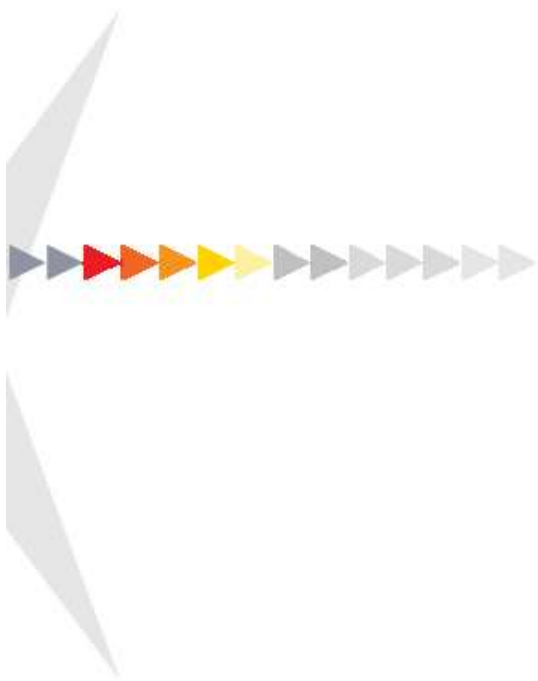


软件工程

大连理工大学软件学院



第7章 类的详细设计



- 详细设计是以概要设计说明书作为基础，完成各模块的算法设计、用户界面设计以及数据结构设计的细化等。
- 对于开发者来说详细设计的主要工作就是在每个类的方法中补充对应的业务实现。
- 类中方法的设计，使用状态图对类的行为，使用**OCL**对类的实现细节进行准确定义。

详细设计的主要活动



- 为每个模块进行详细的**算法设计**。用某种图形、表格、语言等工具将每个模块处理过程的详细算法描述出来。
- 为模块内的**数据结构**进行设计。对于需求分析、概要设计确定的概念性的数据类型进行确切的定义。
- 为数据结构进行物理设计，即确定**数据库的物理结构**。物理结构主要指数据库的存储记录格式、存储记录安排和存储方法，这些都依赖于具体所使用的数据库系统。

详细设计的主要活动



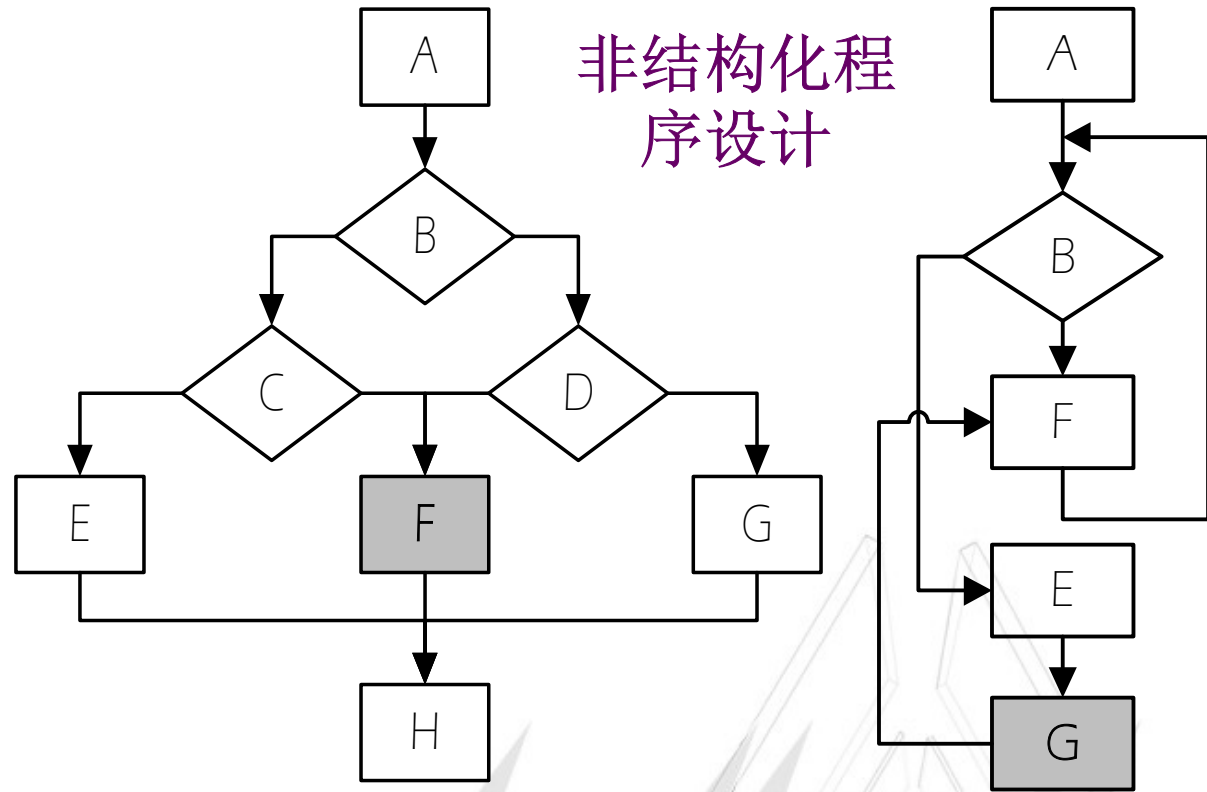
- **其他设计**：根据软件系统的类型，还可能要进行以下设计：输入/输出格式设计；人机对话设计；对于一个实时系统，用户与计算机频繁对话，因此要进行对话方式、内容、格式的具体设计等。
- 编写**详细设计说明书**。
- **评审**。对处理过程的算法和数据库的物理结构等进行评审。

类方法的详细设计

- 结构化的程序

如果程序的代码块仅仅通过顺序、选择和循环三种控制结构进行连接，并且每个代码块只有一个入口和一个出口，则称这个程序是结构化的。

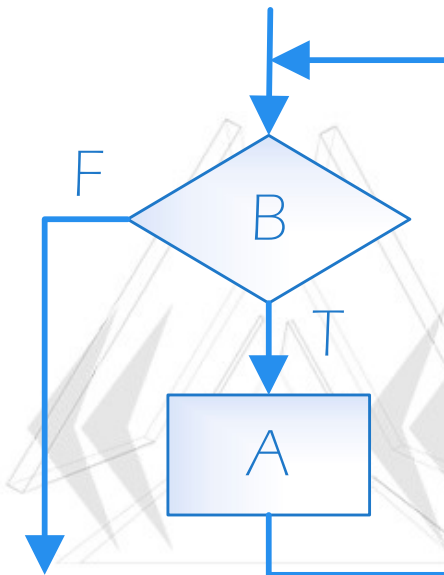
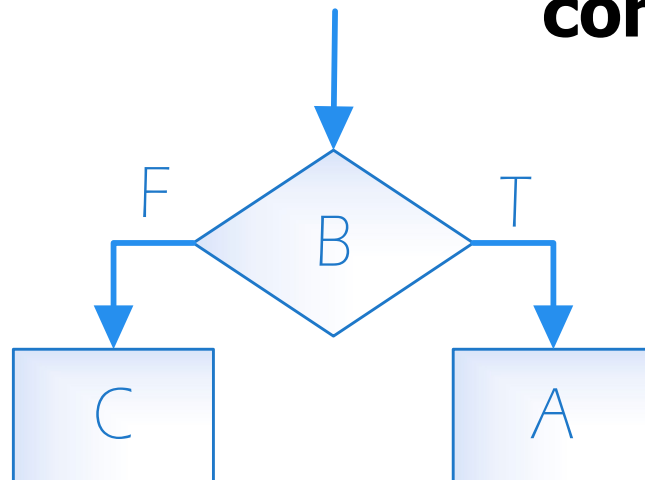
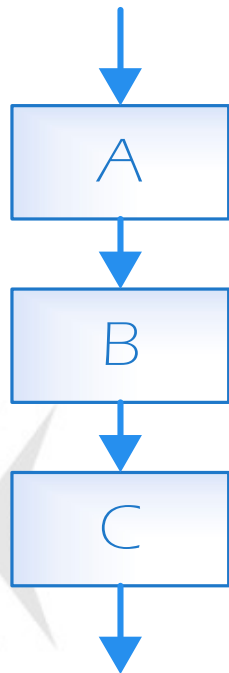
- 目的是简化设计过程，仅使用有限的可预测的操作即可完成相应的算法流程



图形设计工具

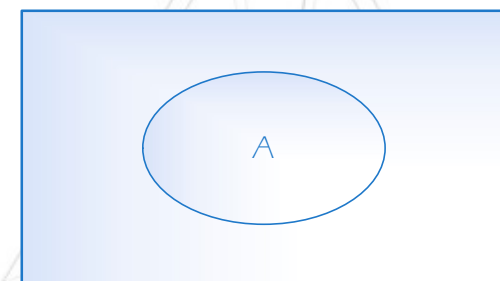
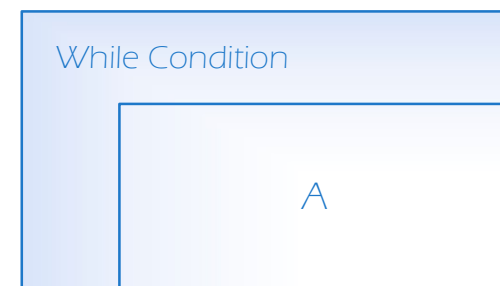
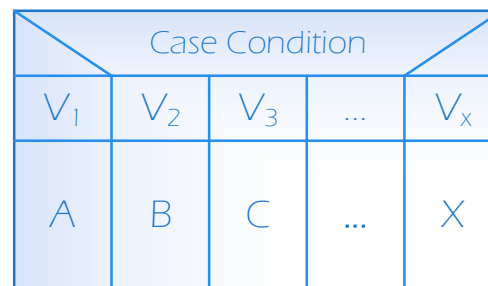
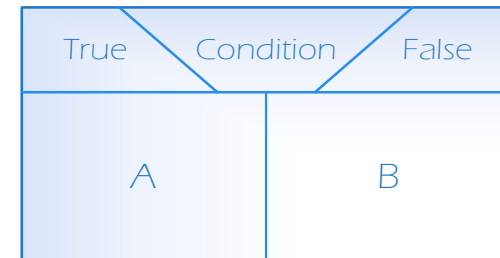


- 程序流程图（**flowchart**）
- 简单、直观、易于学习
- 应尽可能保持结构化的特点，限制控制流的随意跳转
- 适当使用**goto**、**break**、**continue**等语句



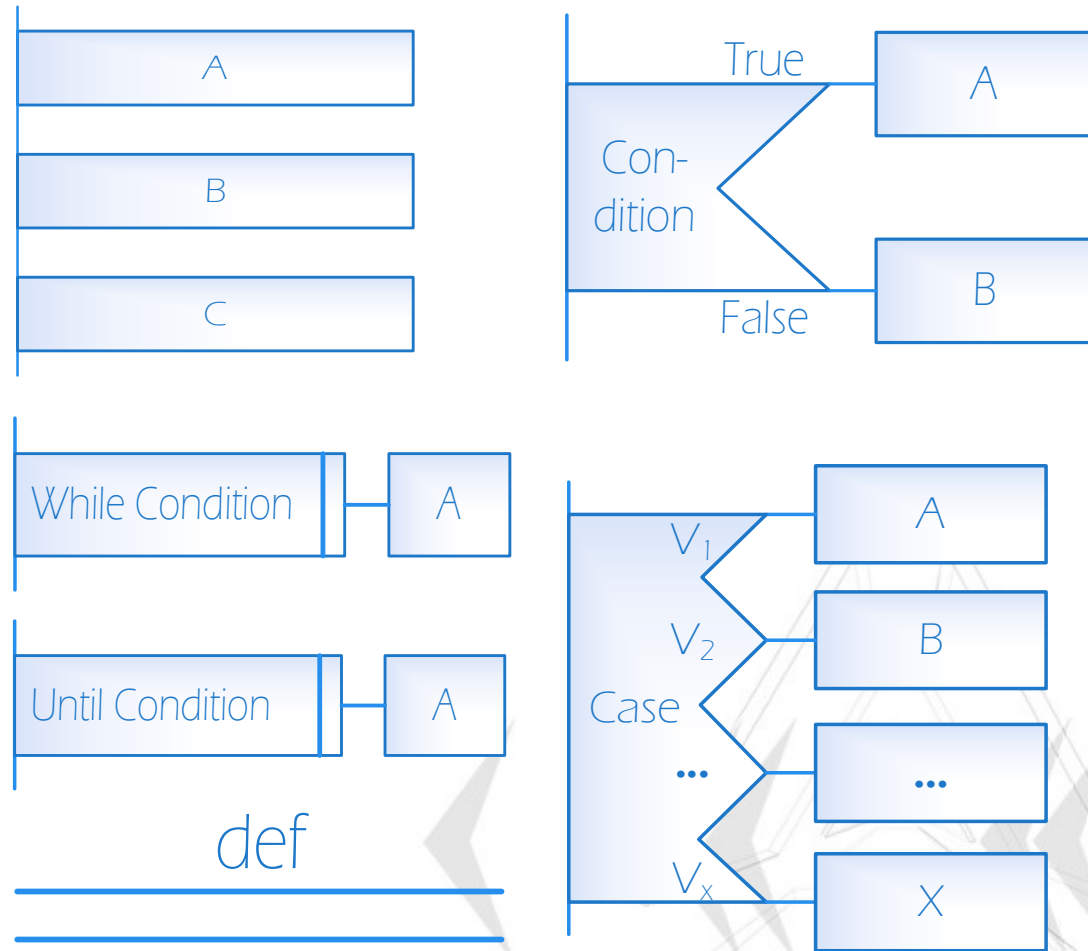
图形设计工具

- **盒图 (NS)**
- 一种符合结构化程序设计原则的图形描述工具
- 不允许随意跳转
- 通过组合嵌套方式构成更大的“盒子”
- 复杂逻辑绘制繁琐



图形设计工具

- 问题分析图（PAD）
- 使用二维树形结构的图来表示程序的控制流，绘制方便
- 结构清晰，翻译到代码较为容易



表格工具



- **判定表**是一种进行详细设计的表格工具，又称为决策表。
- 判定表适用于描述判断条件较多，各条件又相互组合、有多种决策方案的情况。
- 判定表有着准确而又简洁的描述方式，能够将复杂的条件组合与对应的执行动作相对应。
- 例如：某工厂机器维修的方式：对功率大于**50**马力的机器或已运行**10**年以上的机器，应送到专业的维修公司处理；否则如果功率小于**20**马力，并且有维修记录，则在车间维修；否则送到本厂的维修中心维修。

判定表示例



规则#	1	2	3	4	5	6	7	8	9	10	11	12
机器功率 w	A	A	A	A	B	B	B	B	C	C	C	C
运行时长<10年?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
有维修记录	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y
送外维修	※	※	※	※	※	※			※	※		
本厂维修							※				※	※
本车间维修								※				
$A : w > 50, B : w < 20, C : 20 \leq w \leq 50$												

判定表构成



- 判定表有4个部分构成，分别是条件列表、条件组合、动作列表及动作入口；
- 每个条件对应一个变量、关系或者预测，如上例中的机器功率、运行时长、维修记录；
- 条件组合是各种条件可能取值的所有组合，如果每个条件有真假两种取值，则 n 个条件的取值组合数量为 2^n 个；
- 动作指要执行的过程或操作列表，如上例中的送外修或者送本厂维修中心；
- 动作入口指某个条件组合下与动作的对应，与条件组合一起构成了判定表的一列，也叫做规则。

判定表化简

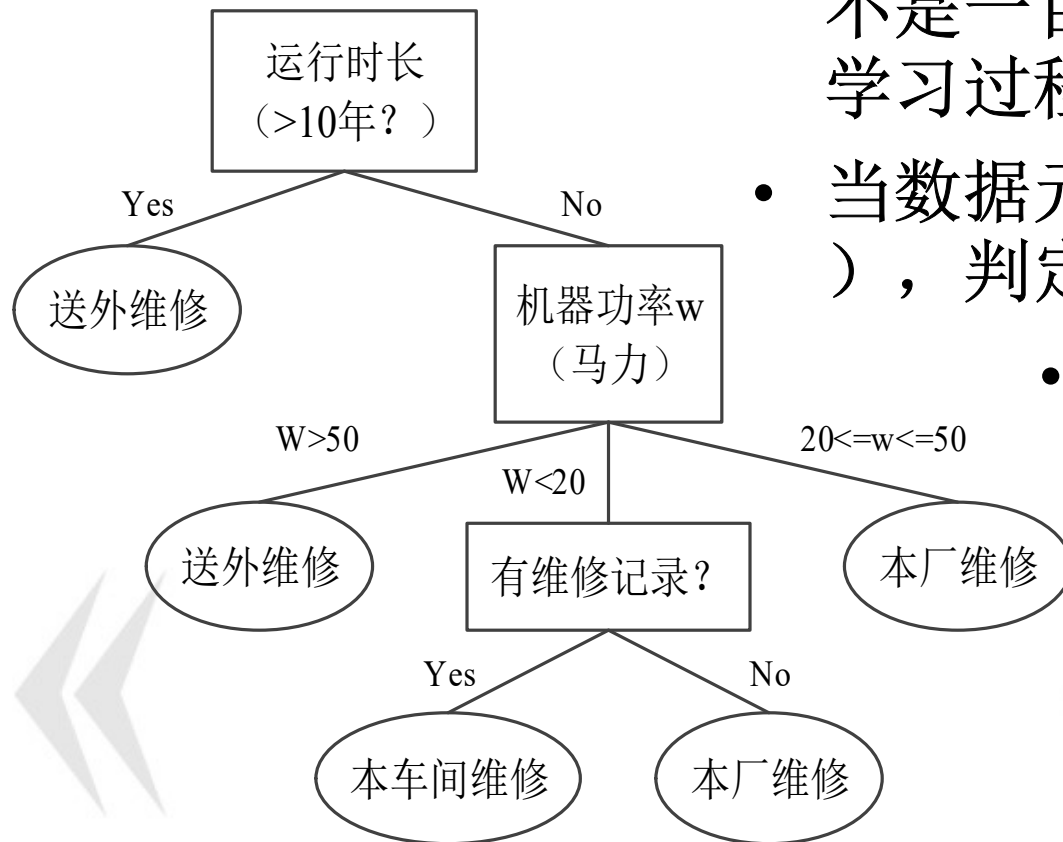


规则#	1	2	3	4	5
机器功率 w	A	—	B	C	B
运行时长<10年?	N	Y	N	N	N
有维修记录	—	—	N	—	Y
送外维修	※	※			
本厂维修			※	※	
本车间维修					※
$A : w > 50, B : w < 20, C : 20 \leq w \leq 50$					

- 使用“—”来表示对此条件的不关心或不适用
- 任意两个条件组合之间不能有交集



判定树



- 判定表虽能表示复杂的条件组合与动作之间的对应关系，但其含义却不是一目了然，理解它也要有一个学习过程。
- 当数据元素值多于两个时（如功率），判定表的简洁程度也将下降。
 - 判定树是判定表的变种，也能清晰地表示复杂的条件组合与应做的动作之间的对应关系。

判定树



- 判定树又称为决策树，是应用于数据分类的一种树结构。
- 其中的每个内部结点 (**internal node**) 代表对某个属性的一次测试，每条边代表一个测试结果，叶结点 (**leaf**) 代表某个类 (**class**) 或者类的分布 (**class distribution**)，最上面的结点是根结点。
- 判定树提供了一种展示类似在什么条件下会得到什么值这类规则的方法。
- 最佳判定树的寻找是一个**NP**困难问题，经典的算法有**ID3**算法，即决策树归纳 (**Induction of Decision Tree**)。

语言工具



- 程序设计语言（**Programming Design Language, PDL**）是一种用来进行详细设计的语言类工具，又称为结构化语言或伪代码。
- 与自然语言相比，**PDL**借用某种编程语言的语法构筑其逻辑流程；
- 与编程语言相比，其中又使用了自然语言的词汇（如英语）。
- **PDL**不能被编译或解释运行，主要是供开发人员使用的。

PDL特点



- **PDL**采用关键字的固定语法，提供了结构化控制结构、数据说明和模块化的特点。
- **PDL**程序中会有一些能够标明程序结构的关键字。
- **PDL**语言仅有少量的简单语法规则，大量使用人们习惯的自然语言。
- 使用**PDL**语言常常按逐步细化的方式写出程序。
- **PDL**程序的注释行对语句进行解释，起到提高可读性的作用。

PDL示例



```
procedure: sort
1:      do while records remain
        read record;
2:      if record field 1 = 0
3:      then process record;
        store in buffer;
        increment counter;
4:      else if record field 2 = 0
5:      then reset counter;
6:      else process record;
        store in file;
7a:     end if
        end if
7b:     end do
8:     end
```

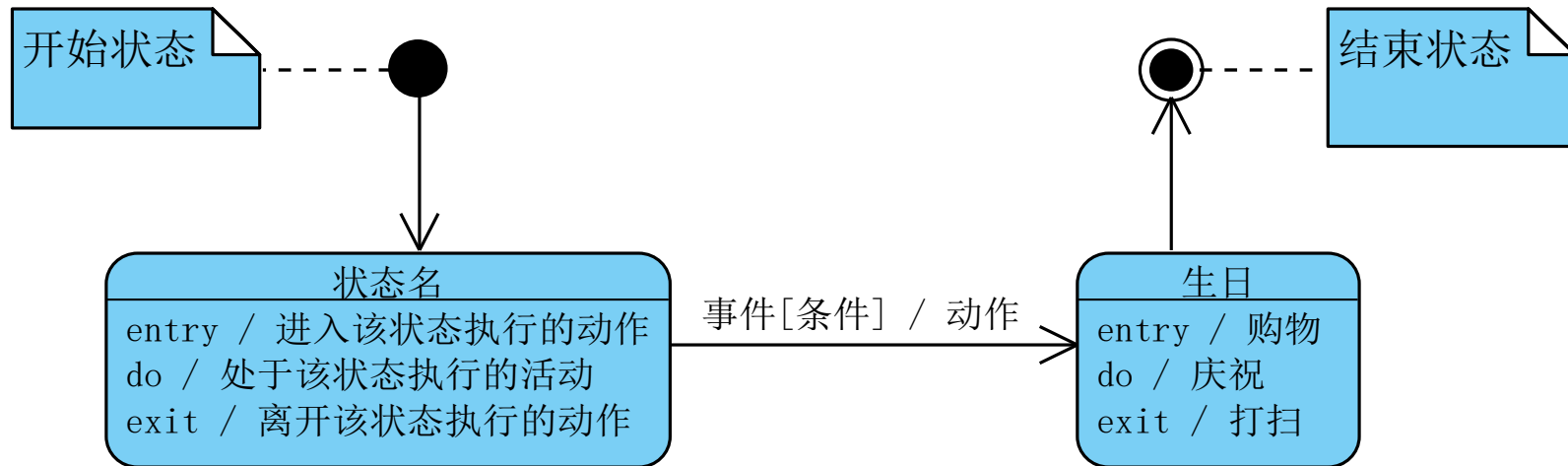
- **PDL**最大的优势是容易翻译成某种编程语言描述的代码，如**C**、**Java**等。
- 但**PDL**不是编程语言，因此不必过多担心存在语法错误，而是将精力集中在设计上。
- **PDL**作为设计工具没有图形工具形象和直观，但其表现形式是最接近代码实现的一种形式。

类的详细设计



- 一些方法的返回值对实例变量值没有任何影响，把这些类的实例对象称为无记忆对象（**memoryless**）或无状态对象（**stateless**）。
- 相反，一些对象的功能需要依赖于其内部状态，把这类对象称为是有记忆对象或者有状态对象。
- 对象的状态以及状态变化可以借助状态图（**State Diagram**）或有穷状态机（**Finite State Machine**）进行描述。

状态图的基本结构



- 左侧描述的是状态的一般构成模板，右侧是一个具体的状态。
- 应含有一个唯一的开始状态，可以有多个结束状态，开始状态和结束状态统称伪状态。
- 状态图中的状态变化都是针对确定性行为的描述。

状态



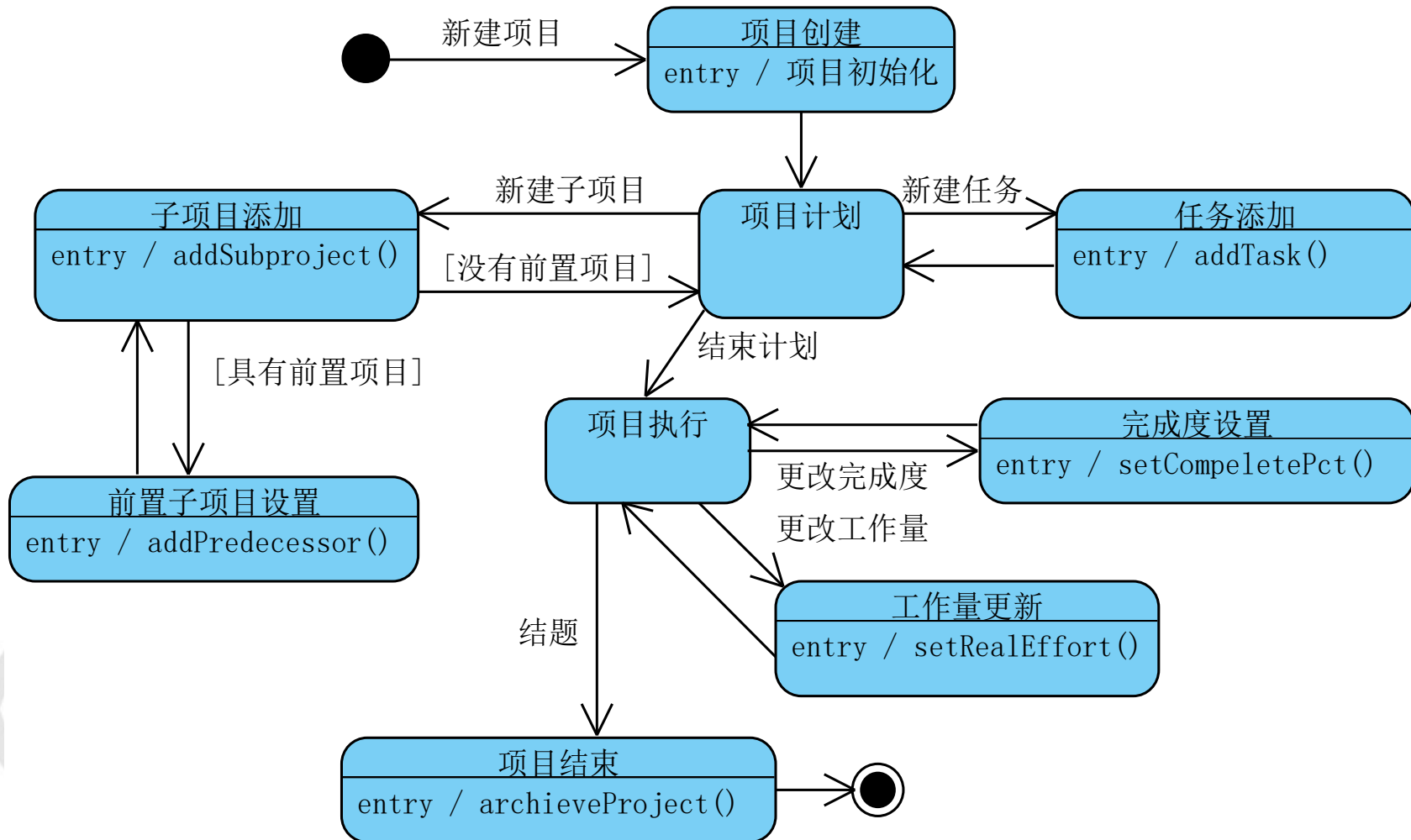
- 每个状态描述中除了状态名字以外，还可以包含以下三个预定义的事件的描述：
- **Entry:** 给出当刚进入该状态时应该进行的动作（**action**）。在这里可以表示一个简单的赋值操作，也可以是对一个或多个方法的调用。
- **Do:** 给出在保持该状态的过程中，对象应执行的活动。这个部分一般对那些受时间控制行为的对象比较适用，因为它们通常要求能够持续的读取信息。
- **Exit:** 这个部分描述当离开该状态时应进行的动作。
- 这三个部分的内容是可选的，根据需要进行取舍。

状态转换



- 状态通过状态转换进行过渡（**Transition**）。
- 事件部分：转换的主要内容，状态图主要是对被动系统的行为描述，对外界的刺激事件进行相应的响应。
- 条件部分：状态间的转换只有在事件被触发并且满足某个特定条件的情况下才会进行，可选。
- 动作：表示当转换发生时执行的一个动作，该动作执行的时机是在转换对应的目标状态的**entry**事件被执行之前，即还未进入到目标状态前。

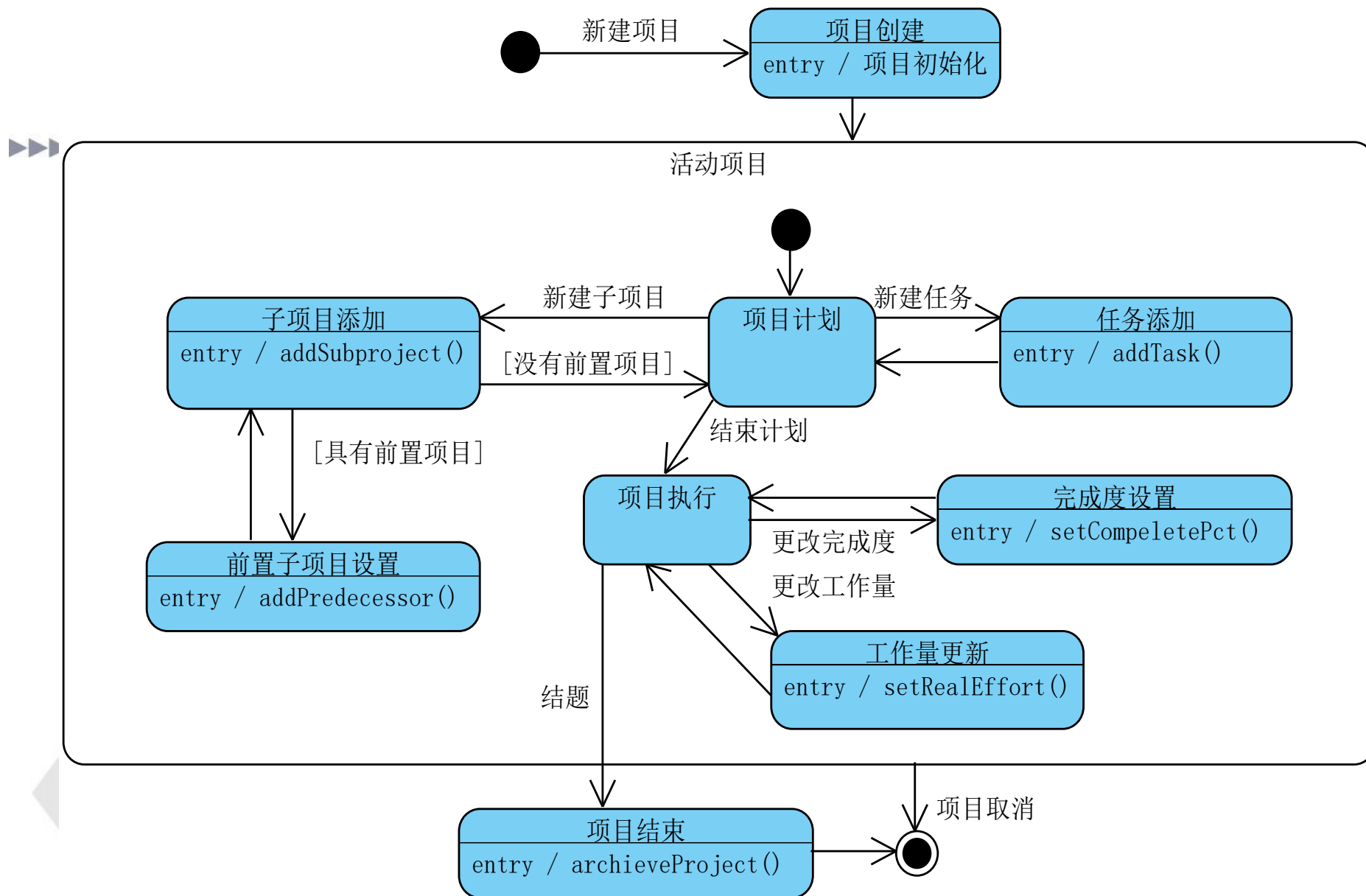
类Project的状态

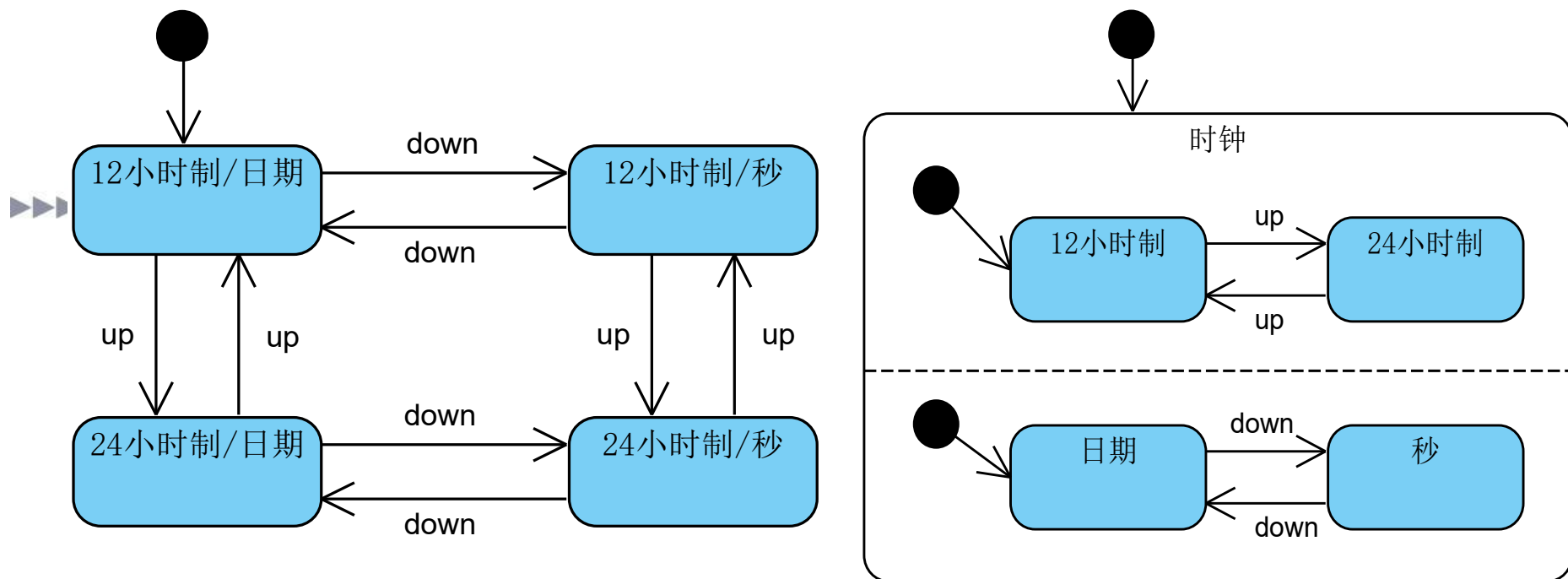


状态图的扩展



- 状态图本身如果过于庞大，容易让人陷入局部而丢失整体上的可理解性。
- 一种经常见到的情况是状态图中若干状态在同一事件作用下具有相同的行为，比如对于异常的处理或者运行的停止。
- 状态可以以一种层次化的方法进行组织，每个状态通过多个子状态细化，该状态称为复合状态。
- 复合状态中的所有子状态只需一个转换来描述共同行为，节省了转换的个数，同时使得图形的绘制保持整洁，不会过于杂乱无章。



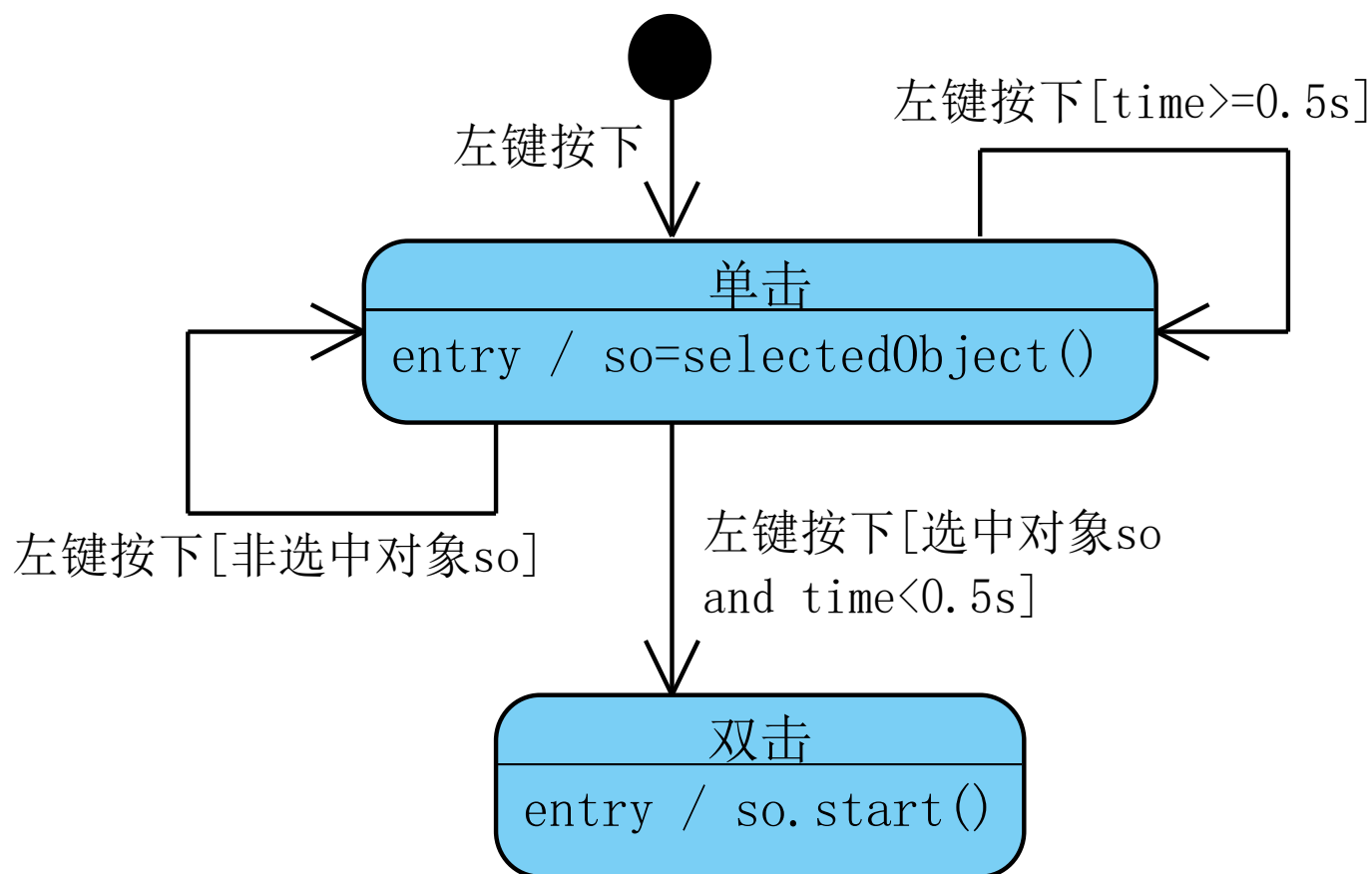


- 对象状态也可能是由多个互不依赖的子状态构成的。
- 每种可能的行为变化，可使用独立的状态图分别进行描述。
- 每个子状态图相互独立，组合在一起又构成了一个全局的状态。
- 使用并行的组合方式减轻状态图的描述负担。

并行状态的考虑

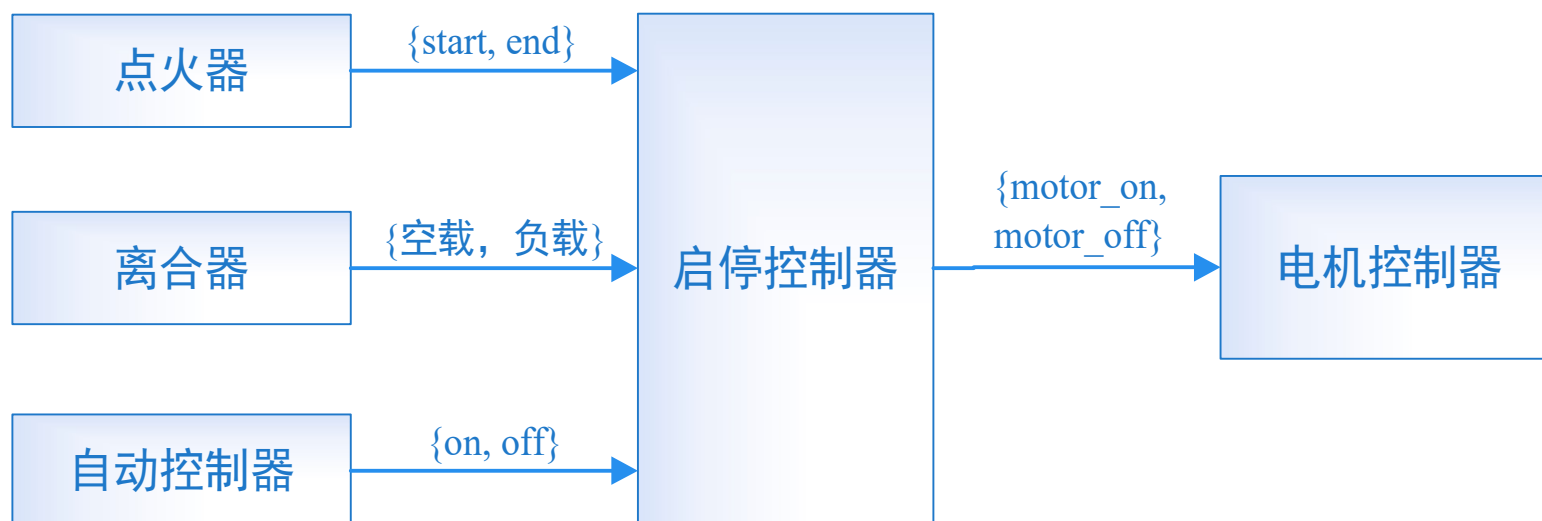


- 如果一个事件在两个子状态图中都需要进行处理，为两个子状态图建立一个到各自结果状态的转换。
- 从含有并行组合的状态出发的转换表示只有当其含有的所有子状态图都位于结束状态时，该转换才会被触发。
- 如果某个复合状态中的一个子状态图的分支导致离开了此复合状态，则所有的子状态图都将结束。也就是说如果离开其中的一个子状态，其它所有的都将自动结束。

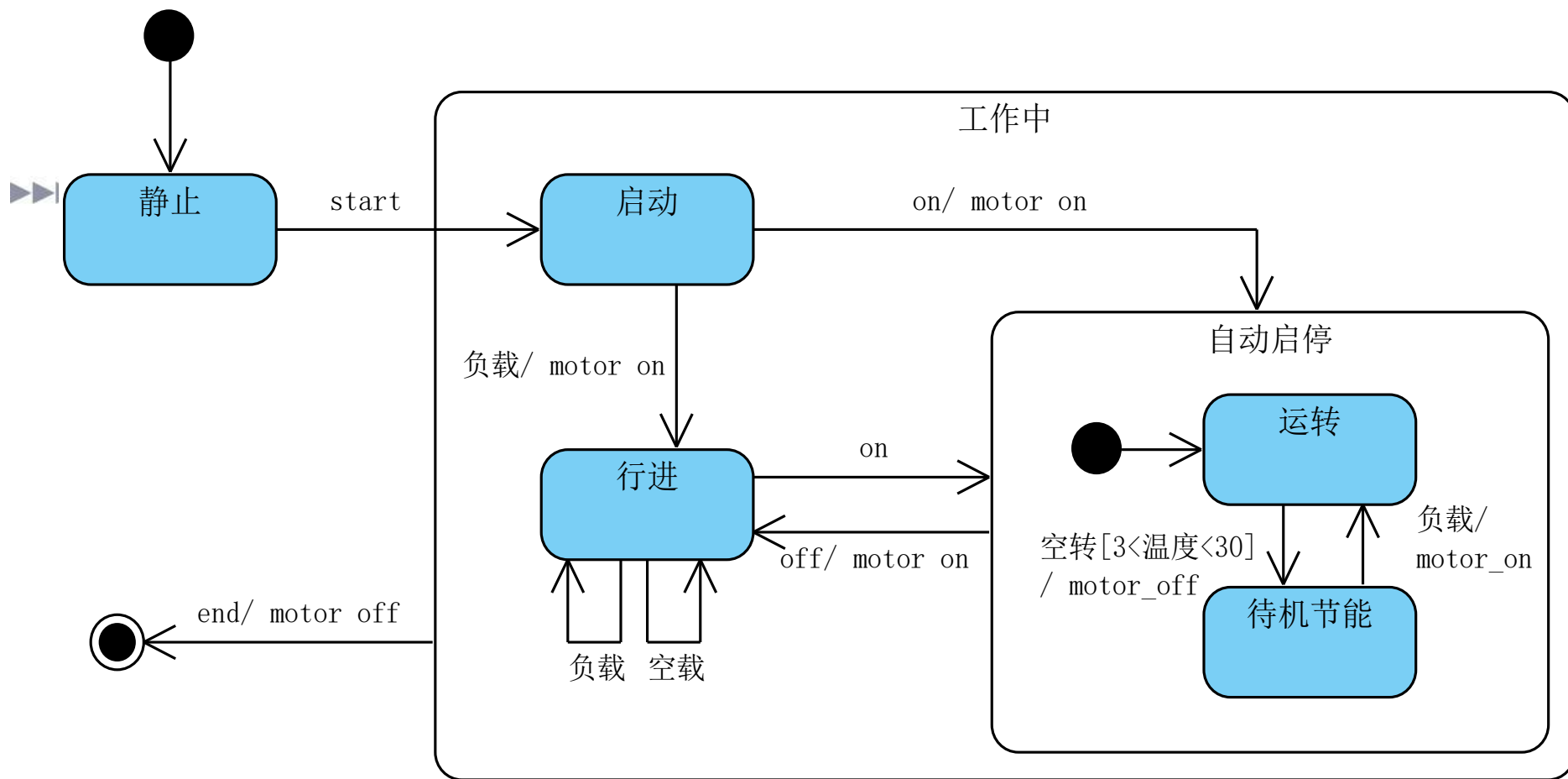


- 带有时间控制的状态图，可以将间隔时间或发生的次数作为条件和动作执行的依据。

状态图应用

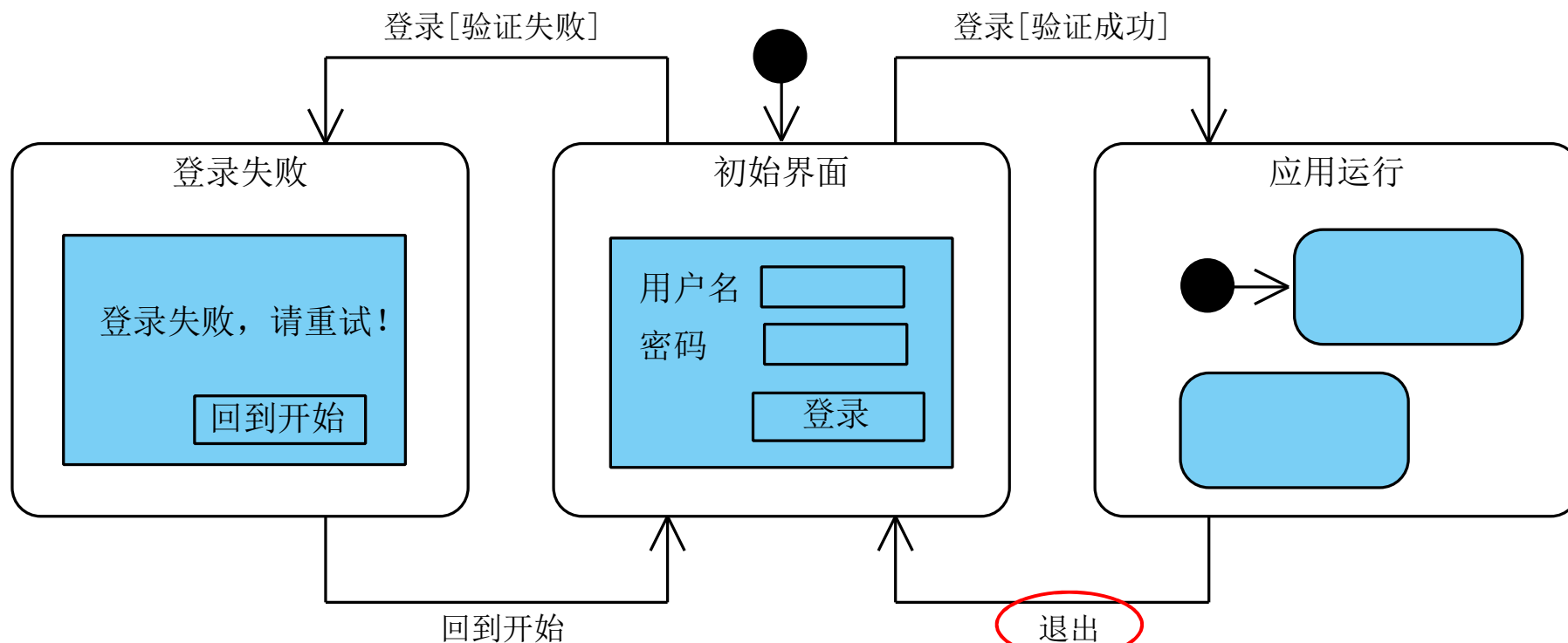


- 点火器组件能够发出**start**和**end**信号；自动控制器能够发出**on**和**off**信号；离合器能够发出空转或者负载的信号。
- 外部温度通过传感器连续测量，存于本地温度变量中。
- 启停控制器本身会产生信号**motor_on**和**motor_off**，并作为动作输入给电机控制器。



- 电机自动启停控制系统的主要状态变化，电机运转时可自动待机节能，如电机加电空载的时候。
- 对条件也进行了指定，如当电机在外部温度范围在**3-30**度时允许自动待机。

操作界面的状态描述



- 状态用来表示每个独立的图形界面，事件是用户的输入，如某个按键的按下，进而弹出另外一个界面，即到达了一个新的状态。

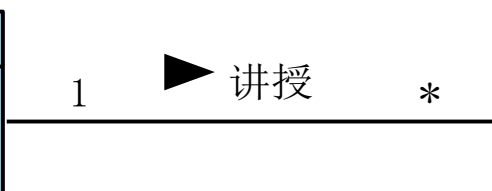
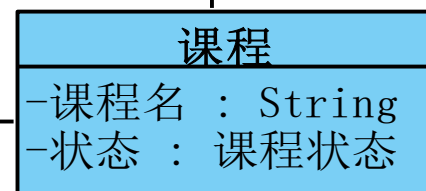
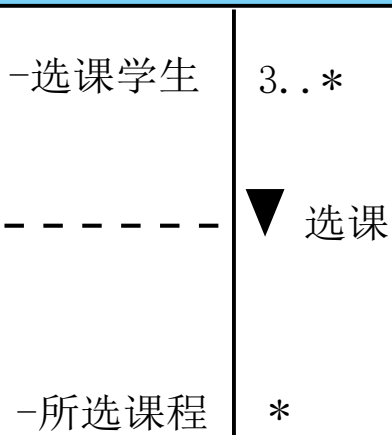
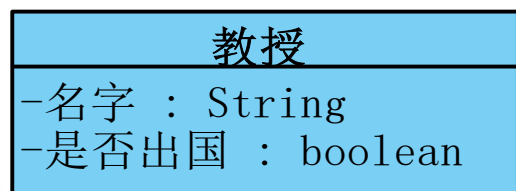
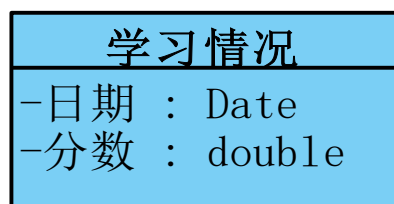
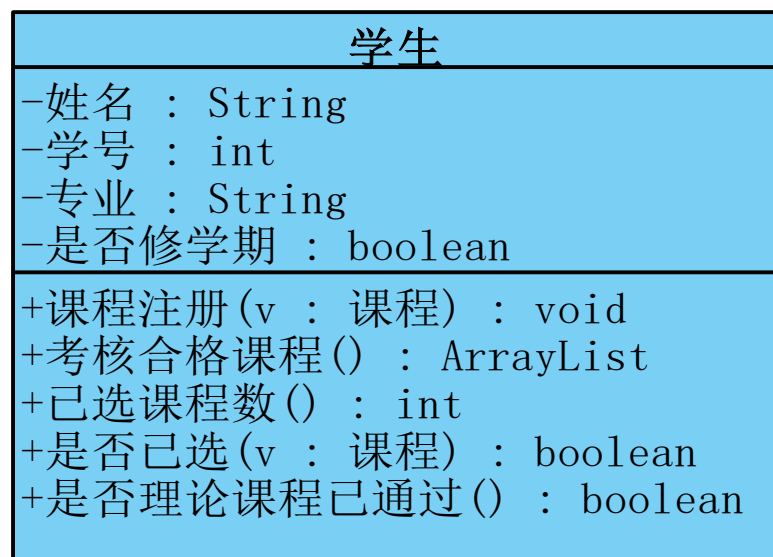
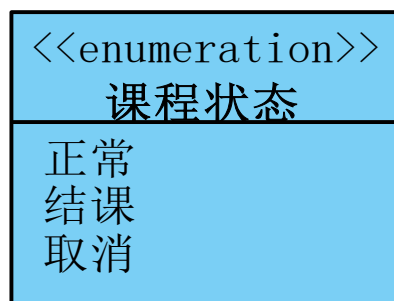
对象约束语言



- 很多业务需求和规则无法利用模型本身的机制进行描述，如“项目类不能以自身作为其子项目”。
- **UML**提供一种对每个对象进行具体条件的约束机制，如：
 - **projectnumber: int {projectnumber >= 0}**
- 以上方法会使类图不够简洁，而且复杂业务规则也不能清晰描述，因此引入了**OCL**。
- 这里引入一个新的类图作为例子：



关联类，只与其对应的一个学生对象和一个课程对象具有联系。



- 学号要求至少五位数字长，并且构成学号的数字要大于或等于10000。若使用OCL进行描述：

context Student inv regStudentId:


self.stuId >= 10000

- 关键字context表明此段OCL的描述与哪些UML对象或类相关。
- 关键字inv表示这是不变的（invariant），也就是说所有的Student对象都要遵守这个约束
- **self**关键字指的是对context对象的引用，正如在Java中的this或C++中的*this的作用一样。

context s:Student inv:

s.stuId >= 10000

- 将**OCL**同样以包的形式进行组织，便于使用和管理：



```

package com::myCompany::mySW
    context Student inv:
    context Student inv:
endpackage

```

- 除了实例变量的约束外，**OCL**同样可以对实例的方法进行，描述方法执行的条件，如未休学的学生至少要选择一门课程：

```
context Student::selectedLectures(): Integer
pre stustatus: self.freeseimester = false
post selectedLectures: result > 0
```

- 当该生选择了一门之前没有学过的课程时，该生选课总数加一：

 **context Student::registerLecture(v: Lecture)**

```
pre: notSelected(v)
```

```
post: self.selectedLectures()@pre =
self.selectedLectures() - 1
```

- **@pre**表示引用的是方法执行之前的状态值。

类型名	取值示例	相关方法
Boolean	true, false	and, or, xor, not implies, if then else endif
Integer	1, -2, 50, 464646	*, +, -, /, abs()
Real	3.14, 42.42, -99.99	*, +, -, /, floor()
String	‘Hello’, ‘Dalian’, “”	Concat(), size(), substring()

- 某门课的授课教授在上课时是不能在国外的：

context Lecture inv:



self.status = Lecturestatus::running

implies

not self.prof.inForeign

- 关键字implies表示 “if-then”的条件说明。
- 对于一个结课的课程成绩要求分数在1-5之间，可以进行如下的描述：

context Examination inv:

self.selectedLecture.status =

Lecturestatus::closed implies

(self.note >= 1.0 and self.note <= 5.0)

- OCL目前只支持一些标准集合类型，如**Collection**、**Set**、**OrderedSet**、**Sequence**以及**Bag**。

collection -> method(parameter)

- 一个学生在一个学期内最多可以选择**12**门课程：

context Student inv:

Student.selectedLecture

-> select (s | s.status = Lecturestatus::running)

-> size() <= 12

- 竖线前的s变量为一个循环变量，用于遍历集合**student.selectedLecture**中的每一个元素。
- 返回来的集合又进一步使用了**size()**方法，提供所含元素数量的整型值。

- 如果在已考试的课程中存在“理论”课程，则方法**hasTheoryLect()**返回真值：

context Student::hasTheoryLect():Boolean

post: result = self.examination

**-> exists(p | p.note >= 60 and
p.selectedLecture.title='Theory')**

- 要求方法**passedLectures()**返回的所有课程必须是已经修过的课程：

context Student::passedLectures():Collection

post: result = self.examination

**-> select(p | p.note>=60) ->
iterate(p:Examination; res: Collection = Collection{} |
res->including(p.selectedLecture))**

OCL小结



- 以**OCL**作为约束的构建时，如果有一些编程经验会使其这个构建过程变得简单。
- **OCL**是一种条件约束语言，因此并不能直接从**OCL**翻译到具体的代码实现；
- 又由于**OCL**是一种描述性语言，因此对于建模来说可视情况进行指定，并不需要强制给出。

作业



- 习题1~6
- 习题7（选做）

