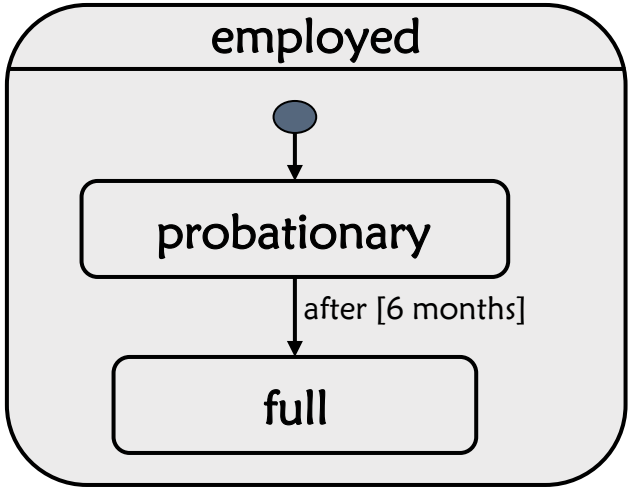
UML状态图中的组合状态（Superstates）

- 可以通过状态嵌套的方式简化图表
  - 一个组合状态可以包含一个或多个状态
  - 组合状态可以实现从不同抽象层次去体现状态图

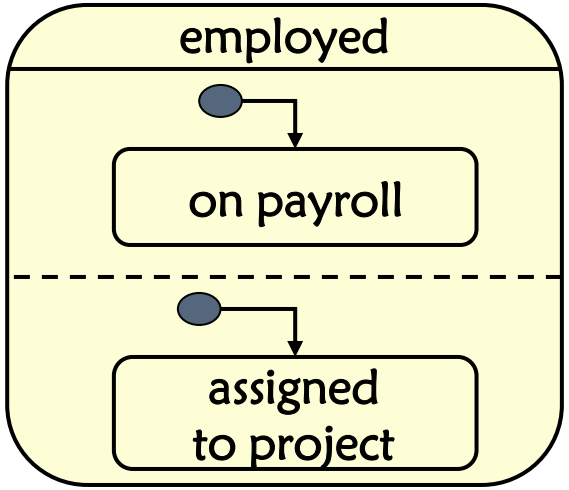
“OR”的组合状态

- 处于组合状态时只能满足其中一个子状态

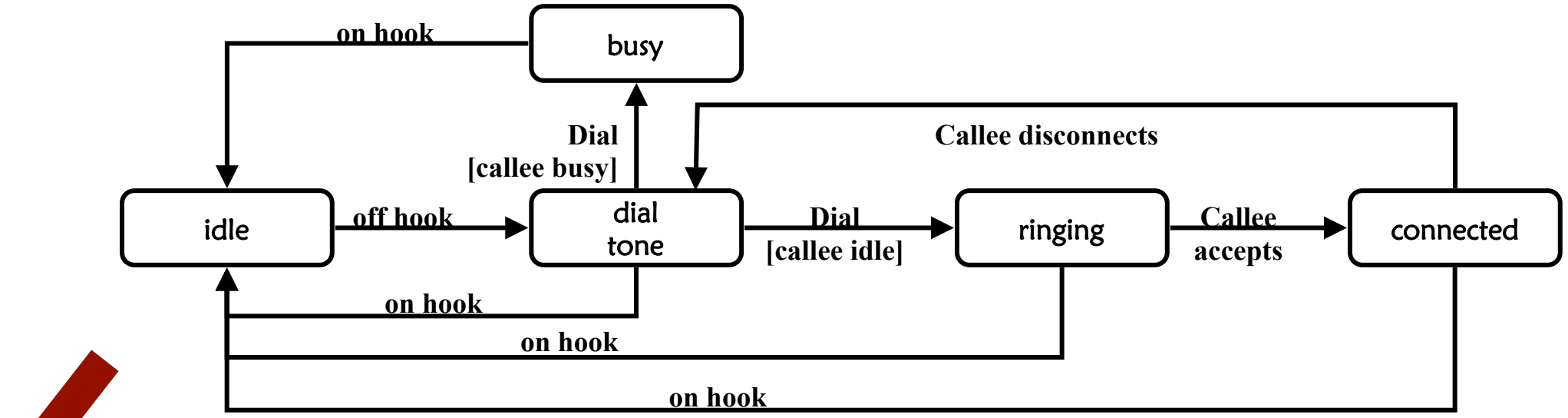


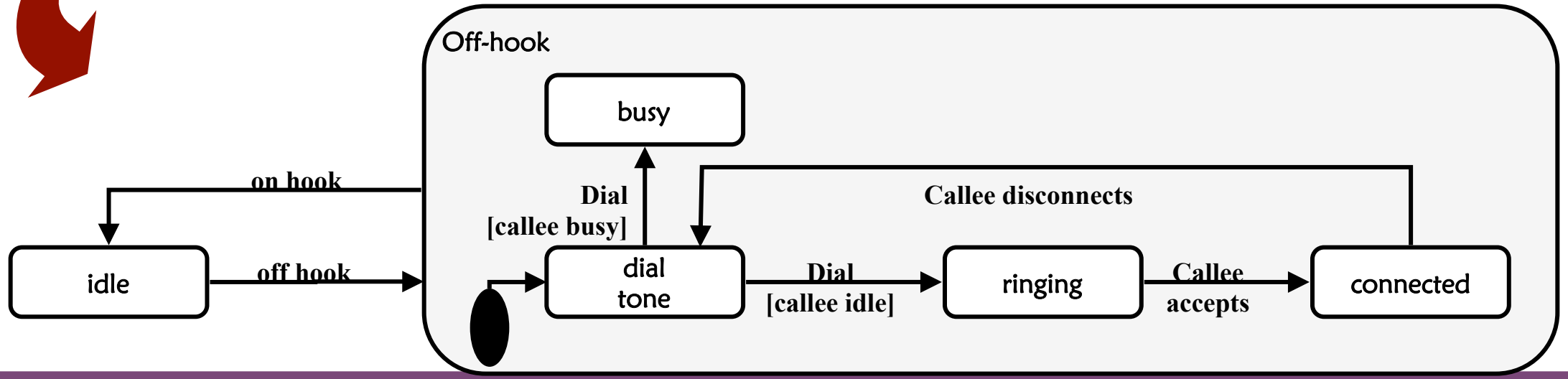
“AND”的组合状态(并发状态)

- 处于组合状态时，满足所有的子状态
- 通常，AND的子状态会进一步嵌套为OR的子状态

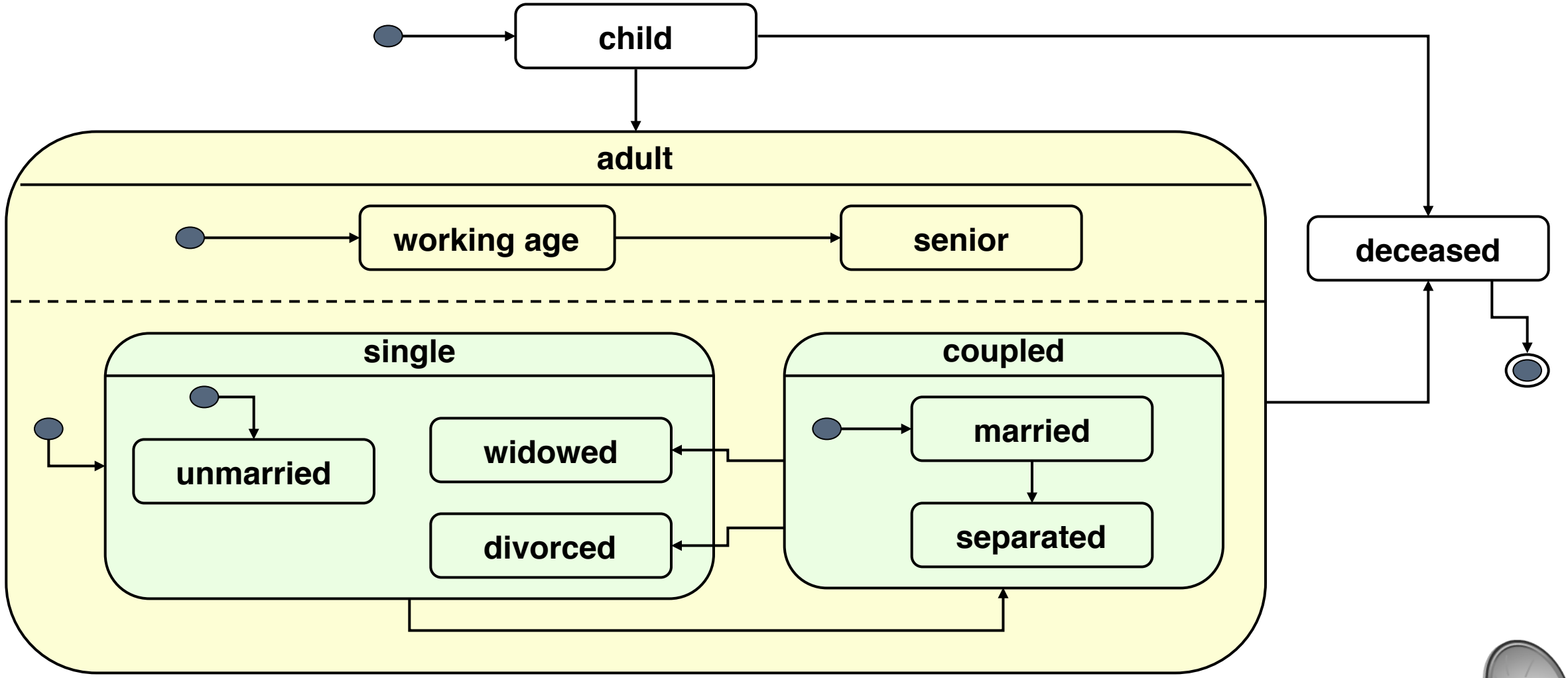


组合状态的例子


busy
on hook
Dial
[callee busy]
Callee disconnects
idle
off hook
dial
tone
Dial
[callee idle]
ringing
Callee
accepts
connected
on hook
on hook
on hook

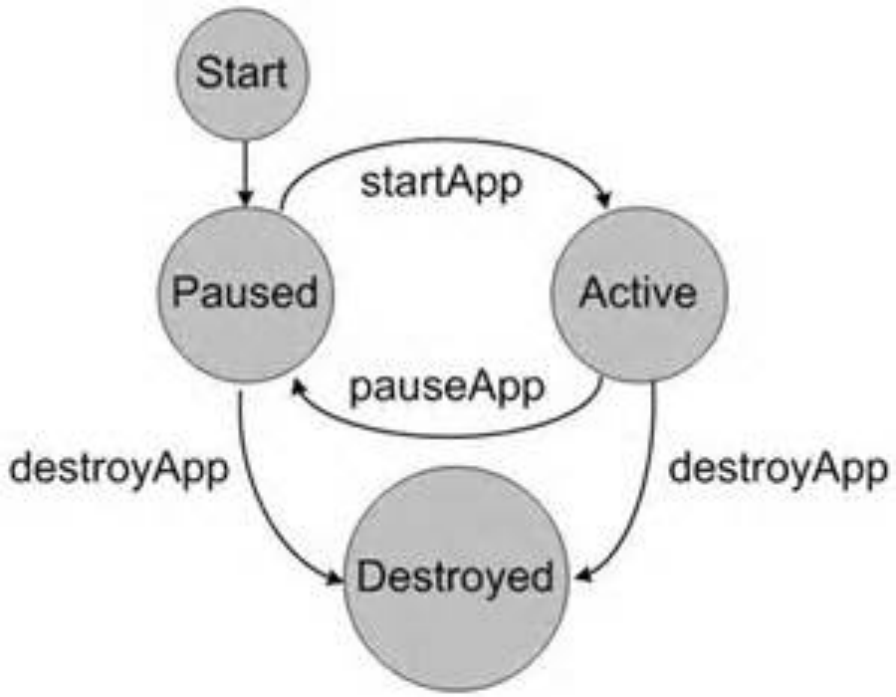

Off-hook
busy
on hook
Dial
[callee busy]
Callee disconnects
idle
off hook
dial
tone
Dial
[callee idle]
ringing
Callee
accepts
connected

组合状态的例子

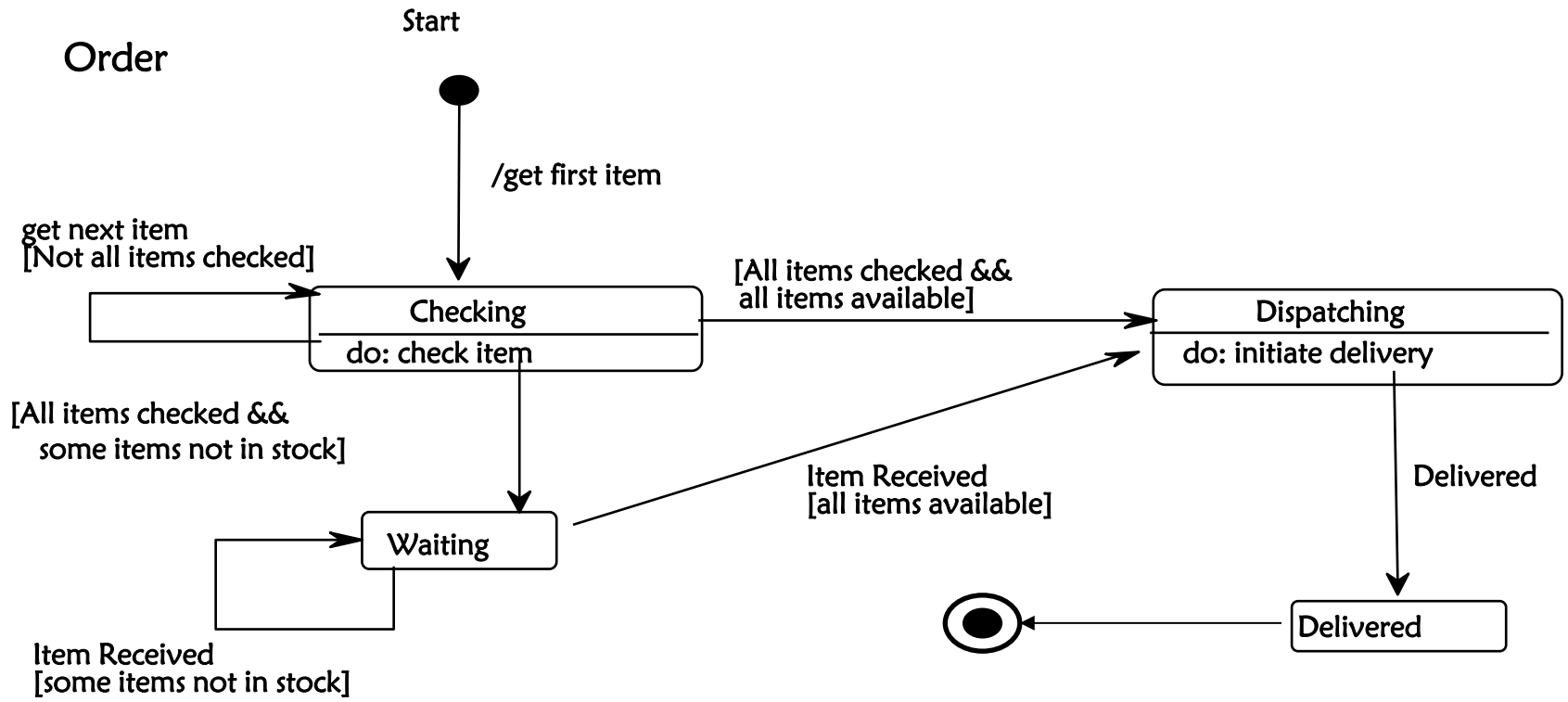

child
adult
working age
senior
deceased
single
unmarried
widowed
divorced
coupled
married
separated

组合状态的状态迁移

- 指向组合状态边界的状态迁移等价于指向该组合状态初态的迁移
  - 所有属于该组合状态的入口条件将被执行
- 从组合状态边界转出的迁移等价于从该组合状态的终态发出迁移
  - 所有出口条件均将被执行
- 迁移可直接指向组合状态的子状态

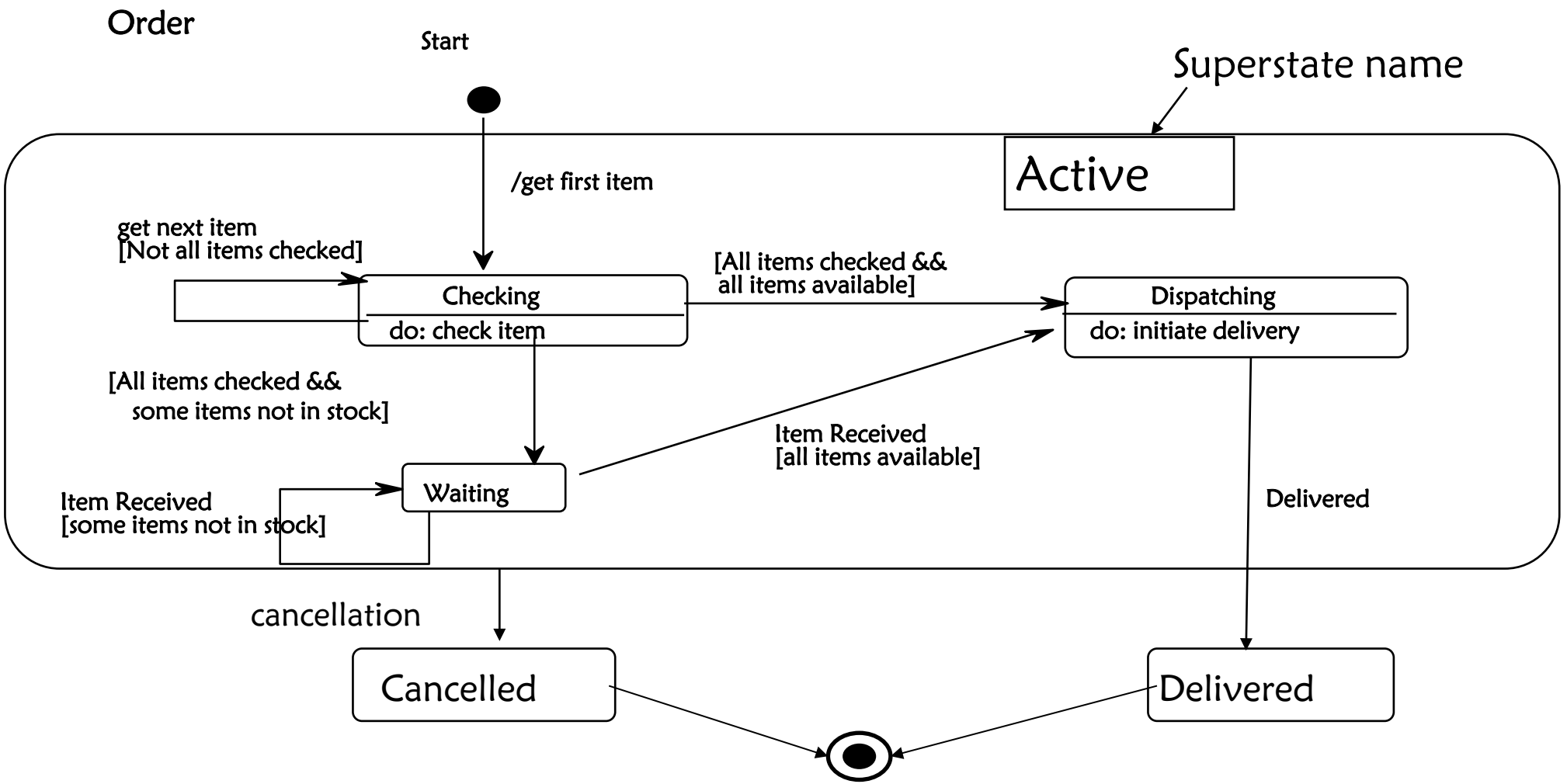


练习：组合状态



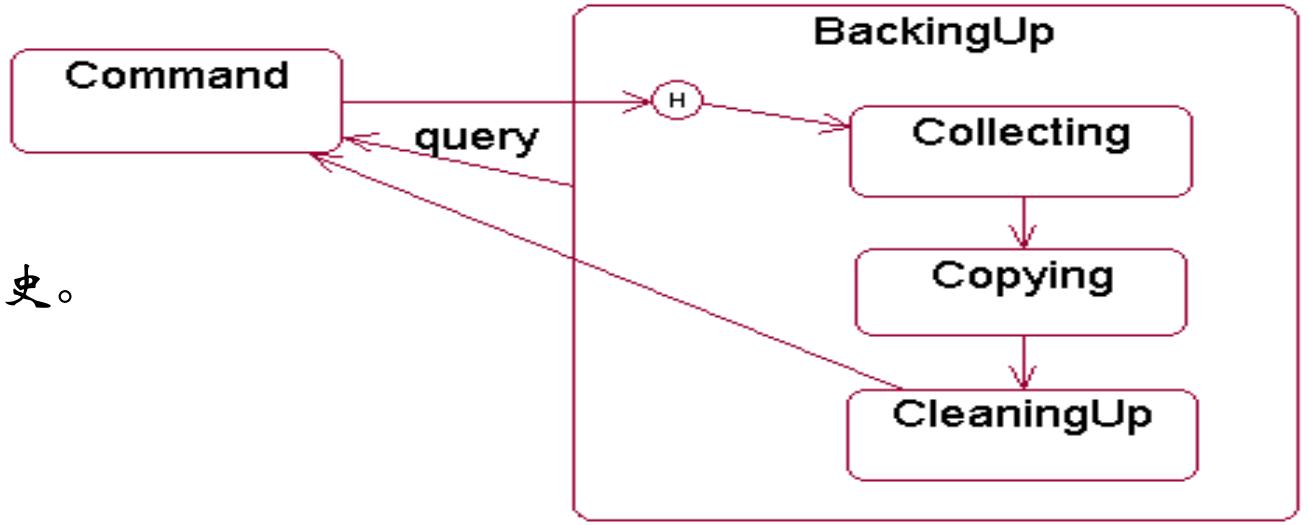
练习：在图中增加一个新的状态和相关的状态迁移，表示在物品投递之前的任何环节都可以取消订单

参考答案


Order
Start
Superstate name
Active
/get first item
get next item
[Not all items checked]
Checking
do: check item
[All items checked &&
all items available]
Dispatching
do: initiate delivery
[All items checked &&
some items not in stock]
Item Received
[all items available]
Item Received
[some items not in stock]
Waiting
Delivered
cancellation
Cancelled
Delivered

UML状态图中的历史状态(History State)

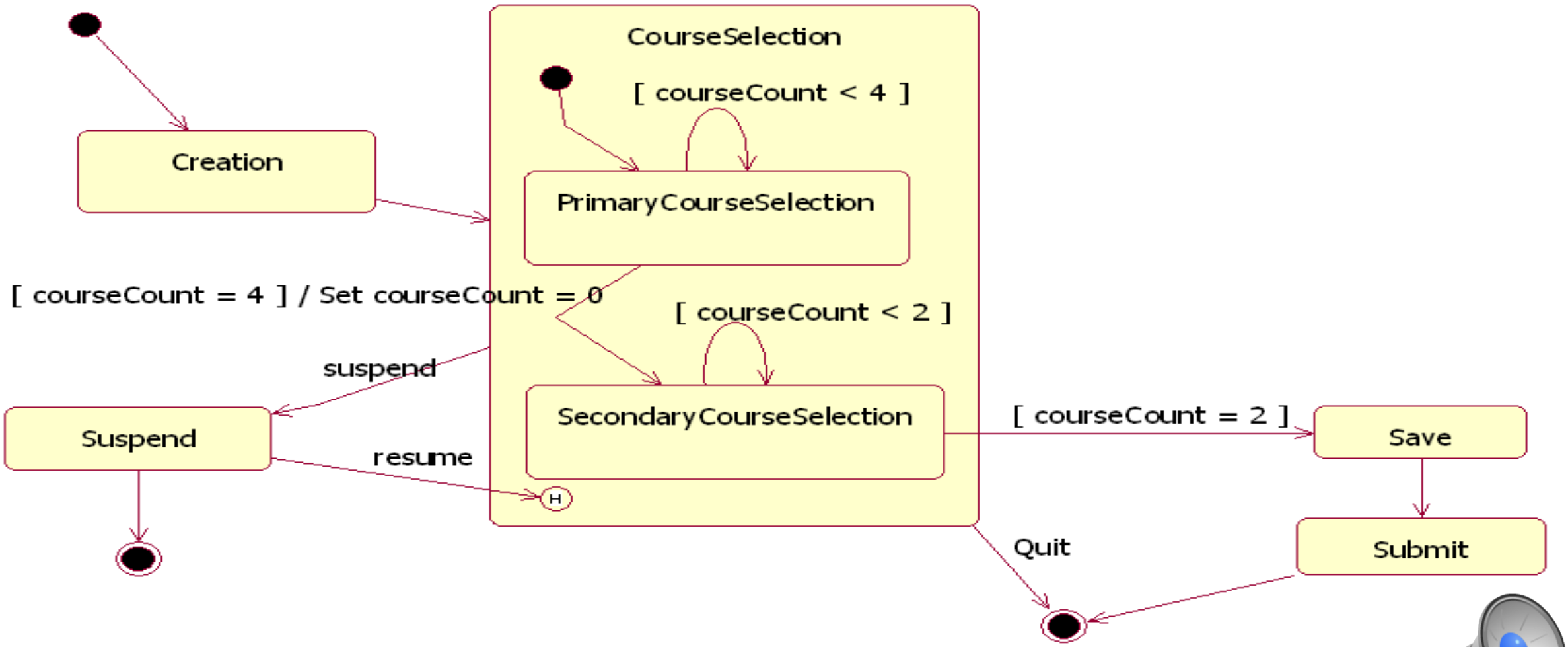
- 历史状态是一种伪状态。当激活这个状态时，会保存从组合状态中退出时所处的子状态，用H表示
- 当再次进入组合状态时，可直接进入到这个子状态，而不是再次从组合状态的初态开始。



- H和H*的区别：
  - H只记住最外层的组合状态的历史。
  - H*可记住任何深度的组合状态的历史。

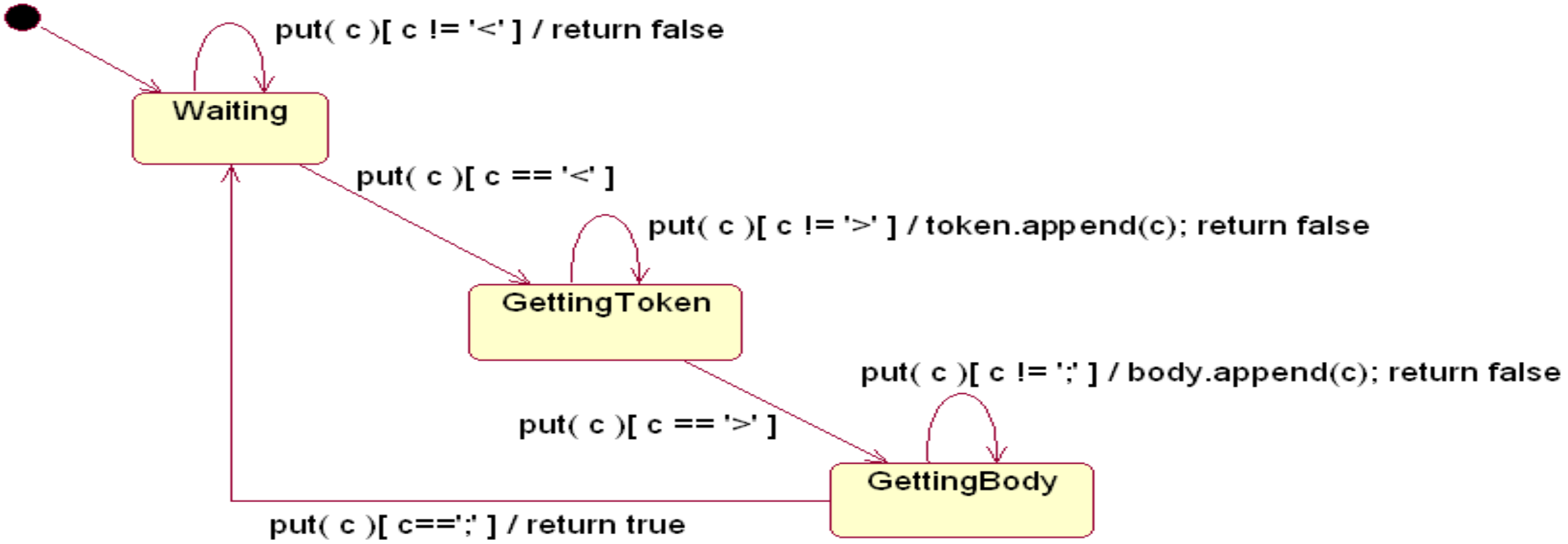
例：历史状态的例子。

例：课程注册


CourseSelection
[courseCount < 4]
Creation
PrimaryCourseSelection
[courseCount = 4] / Set courseCount = 0
[courseCount < 2]
suspend
SecondaryCourseSelection
[courseCount = 2]
Save
Suspend
resume
H
Submit
Quit

状态图的工具支持

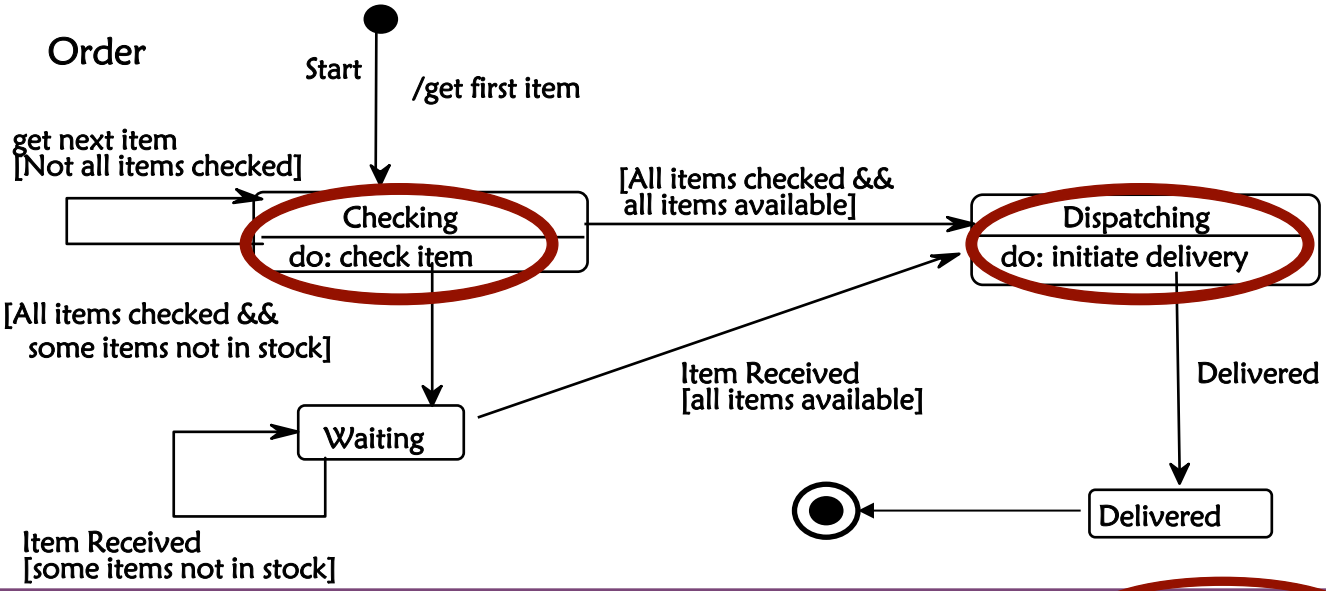
- 正向工程：根据状态图生成代码。例：

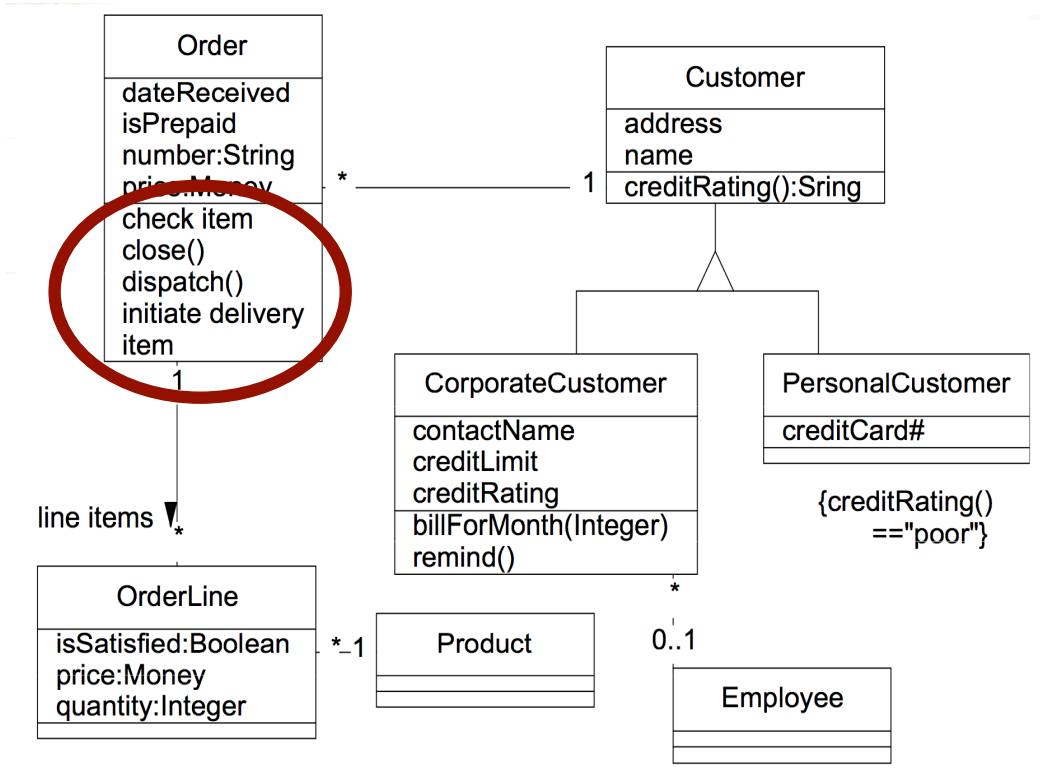


```
class MessageParser {
public boolean put(char c) {
    switch (state) {
        case Waiting:
            if (c == '<') {
                state = GettingToken;
                token = new StringBuffer();
                body = new StringBuffer();
            }
            break;
        case GettingToken :
            if (c == '>')
                state = GettingBody;
            else
                token.append(c);
            break;
        case GettingBody :
            if (c == ';') {
                state = Waiting;
                return true;
            }
            else
                body.append(c);
        }
        return false;
    }
public StringBuffer getToken() {
    return token;
}
public StringBuffer getBody() {
    return body;
}
private final static int Waiting = 0;
private final static int GettingToken = 1;
private final static int GettingBody = 2;
private int state = Waiting;
private StringBuffer token, body;
}
```

状态图与其他UML图的关系

- 状态图中的事件为顺序图/交互图中该对象的输入消息
- 状态图应针对类图中具有重要行为的类进行建模
- 每个事件、动作对应于相应类中的一个具体操作
- 状态图中每个输出消息对应于其他类的一个操作
- 状态图中的操作定义等价于类图中的操作定义





状态图建模风格

- 建模风格1：把初态放置在左上角；把终态放置在右下角
- 建模风格2：用过去式命名转移事件
- 建模风格3：警戒条件不要重叠
- 建模风格4：不要把警戒条件置于初始转移上

状态图的检查表

- 一致性检查
  - 状态图中所有的事件应该是
    - 类图中本对象类的方法
  - 状态图中所有的动作应该是
    - 类图中其他对象类的方法

- 绘图风格
  - 每个状态的命名应该是唯一的，意义明确的
  - 只对行为复杂的状态使用组合状态建模
  - 不要在一个图中包含太多细节
  - 使用警戒条件时要特殊注意不要引入二义性
    - 状态图应该具有确定性（除非特殊原因）

下述情况不适宜使用状态图：

- 当大部分的状态转移为“当这个状态完成时”
- 有很多来自对象自身发出的触发事件
- 状态代表的信息与类中的属性赋值并不一致