【只有一个仲裁者】

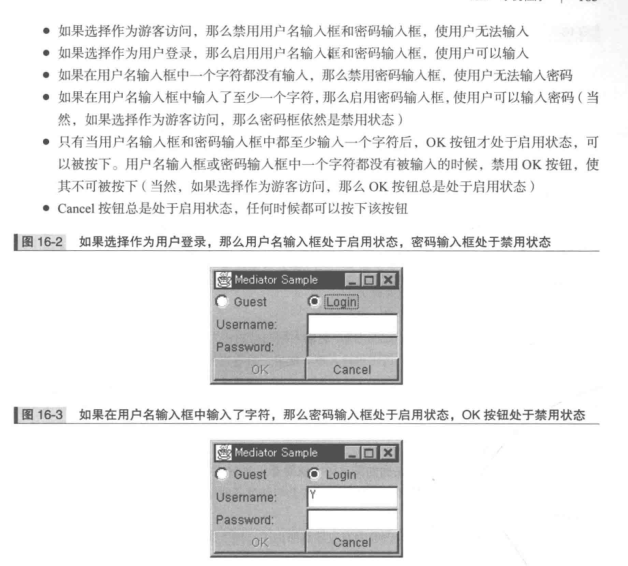
请大家想象一下一个乱槽槽的开发小组的工作状态。小组中的10个成员虽然一起协同工作，但是意见难以统一，总是互相指挥，导致工作始进度始终滞后。不仅如此，他们还都十分在意编码细节，经常为此争执不下。这时，我们就需要一个中立的仲裁者站出来说：“各位，请大家将情况报告给我，我来负责仲裁。我会从团队整体出发进行考虑，然后下达指示。但我不会评价大家的工作细节。”这样，当出现争执时大家就会找仲裁者进行商量，仲裁者会负责统一大家的意见。最后，整个团队的交流过程就变为了组员向仲裁者报告，仲裁者向组员下达指示。组员之间不再相互询问和相互指示。

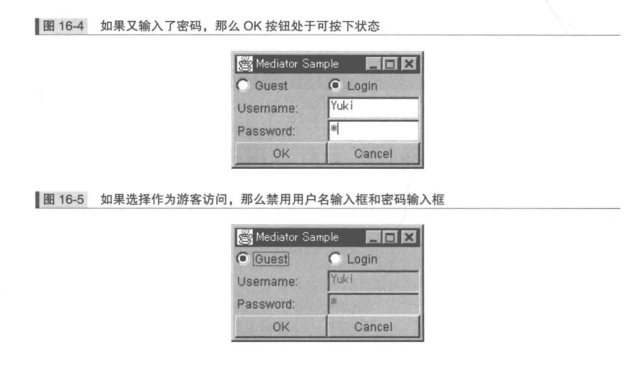
在本章中，我们将要学习Mediator模式。

Mediator的意思是“仲裁者”“中介者”。一方面，当发生麻烦事情的时候，通知仲裁者；当发生涉及全体组员的事情时，也通知仲裁者。当仲裁者下达指示时，组员会立即执行。团队组员之间不再互相沟通并私自做出决定，而是发生任何事情都向仲裁者报告。另一方面，仲裁者站在整个团队的角度上对组员上报的事情做出决定。这就是Mediator模式。

在Mediator模式中，“仲裁者”被称为Mediator，各组员被称为Colleague。Colleague这个单词很容易拼错，大家可能也不太明白这个英文单词的意思，但在GoF书（请参见附录E[GoF]）中就是这么记述的，因此本书中沿用该术语。









哎呀，上面列举的这些操作情况好复杂，用语言都很难表达清楚。不过大家在实际操作一下这个对话框后，应该能够很容易地理解设计者的意图。那么，我们应该如何编写程序呢？

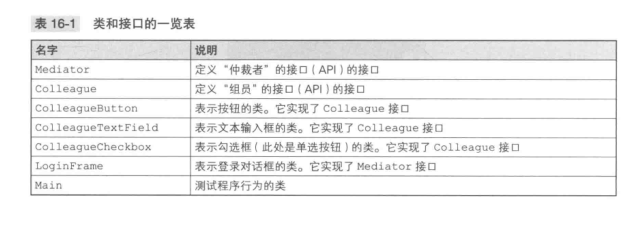
对话框中的单选按钮（radiobutton，用于选择作为游客访问还是作为用户登录）、文本输入框（用于输入用户名和密码）、按钮（OK和Cancel）都是单独的类。如果将上面的逻辑处理分散在各个类中，那么编码的工作量会变得非常大。这是因为所有的对象都互相关联、互相制约。

“如果选择作为用户登录，那么启用用户名输入框和密码输入框，但是如果在用户名输入框中一个字符都没有被输入，就必须禁用密码输入框。然后，只有当用户名输入框和密码输入框中都输人有文字后，才能启用OK按钮”……这段代码应该写在哪里才好呢？写在单选按钮的类里面吗？不过如果这样写代码，负责控制显示的代码就会散落在各个类里面了，导致不论是编写代码还是调试代码，都会变得非常麻烦。而且，一旦需求发生了变化，例如要“增加一个用于输入邮件地址的输入框”……想想都让人害怕。

像上面这样要调整多个对象之间的关系时，就需要用到Mediator模式了。即不让各个对象之间互相通信，而是增加一个仲裁者角色，让他们各自与仲裁者通信。然后，将控制显示的逻辑处理交给仲裁者负责。

前面讲得很多，该模式的大致内容大家应该都明白了。接下来，请注意结合上面这些内容来阅读示例程序。

示例程序的类和接口的一览表请参见图16-1，类图和时序图请分别参见图16-7和图16-8。



|16.3Mediator 模式中的登场角色

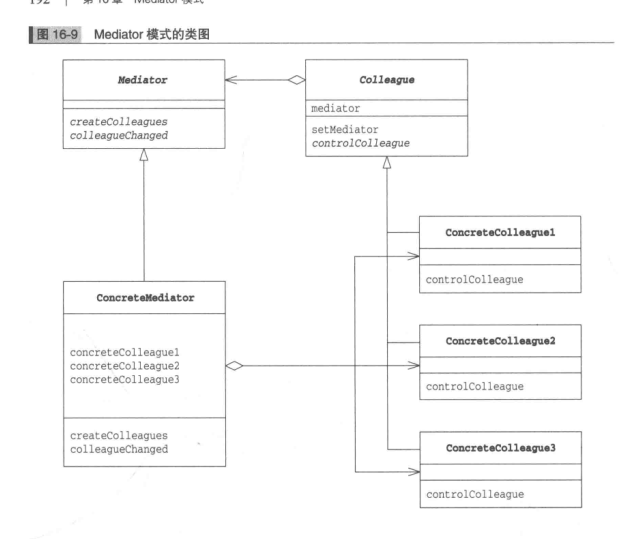
在Mediator模式中有以下登场角色。

·Mediator（仲裁者、中介者）Mediator角色负责定义与Colleague角色进行通信和做出决定的接口（APl）。在示例程序中，由Mediator 接口扮演此角色。

·ConcreteMediator（具体的仲裁者、中介者）ConcreteMediator角色负责实现Mediator角色的接口（API），负责实际做出决定。在示例程序中，由LoginErame类扮演此角色。

·Colleague（同事）Colleague角色负责定义与Mediator角色进行通信的接口（APl）。在示例程序中，由Colleague接口扮演此角色。

·ConcreteColleague（具体的同事）ConcreteColleague角色负责实现Colleague角色的接口（API）。在示例程序中，由ColleagueButton类、ColleagueTextField类和ColleagueCheckbox类扮演此角色。



|16.4拓展思路的要点

当发生分散灾难时

示例程序中的LoginFrame类的colleaguechanged方法（代码清单16-6）稍微有些复杂。如果发生需求变更，该方法中很容易发生Bug。不过这并不是什么问题。因为即使 colleagueChanged方法中发生了Bug，由于其他地方并没有控制控件的启用/禁用状态的逻辑处理，因此只要调试该方法就能很容易地找出Bug的原因。

请试想一下，如果这段逻辑分散在co1leagueButton类、ColleagueTextField类和Colleaguecheckbox类中，那么无论是编写代码还是调试代码和修改代码，都会非常困难。

通常情况下，面向对象编程可以帮助我们分散处理，避免处理过于集中，也就是说可以“分而治之”。但是在本章中的示例程序中，把处理分散在各个类中是不明智的。如果只是将应当分散的处理分散在各个类中，但是没有将应当集中的处理集中起来，那么这些分散的类最终只会导致灾难。

|通信线路的增加

假设现在有A和B这2个实例，它们之间互相通信（相互之间调用方法），那么通信线路有两条，即A→B和A-B。如果是有A、B和C这3个实例，那么就会有6条通信线路，即A→B、A-B、B→C、B-C、C→A和C-A。如果有4个实例，会有12条通信线路；5个实例就会有20条通信线路，而6个实例则会有30条通信线路。如果存在很多这样的互相通信的实例，那么程序结构会变得非常复杂。

可能会有读者认为，如果实例很少就不需要Mediator模式了。但是需要考虑到的是，即使最初实例很少，很可能随着需求变更实例数量会慢慢变多，迟早会暴露出问题。

|哪些角色可以复用

ConcreteColleague角色可以复用，但ConcreteMediator角色很难复用。

例如，假设我们现在需要制作另外一个对话框。这时，我们可将扮演ConcreteColleague角色的ColleagueButton类、ColleagueTextField类和ColleagueCheckbox类用于新的对话框中。这是因为在ConcreteColleague角色中并没有任何依赖于特定对话框的代码。在示例程序中，依赖于特定应用程序的部分都被封装在扮演ConcreteMediator角色的LoginFrame类中。依赖于特定应用程序就意味着难以复用。因此，LoginFrame类很难在其他对话框中被复用。

|16.5相关的设计模式

◆Facade模式（第15章）在Mediator模式中，Mediator 角色与Colleague角色进行交互。

而在Facade模式中，Facade角色单方面地使用其他角色来对外提供高层接口（APl）。因此，可以说Mediator模式是双向的，而Facade模式是单向的。

◆Observer模式（第17章）有时会使用Observer模式来实现Mediator角色与Colleague角色之间的通信。