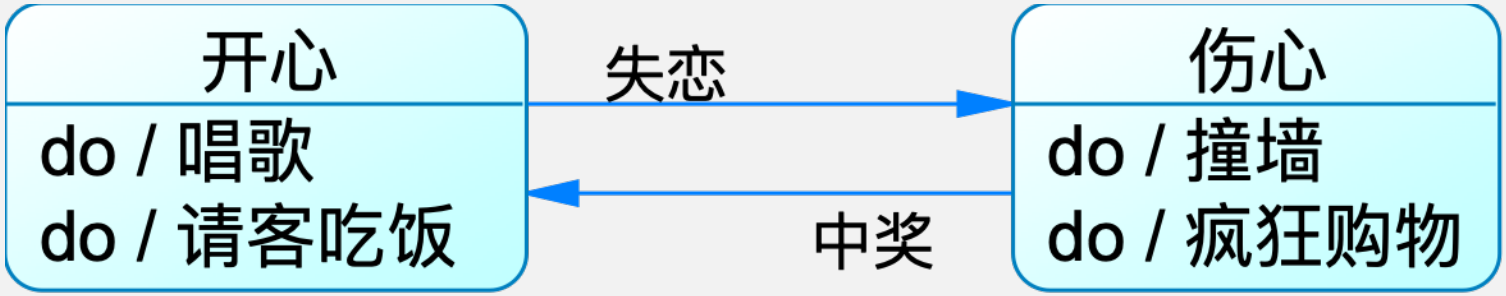
设计模式

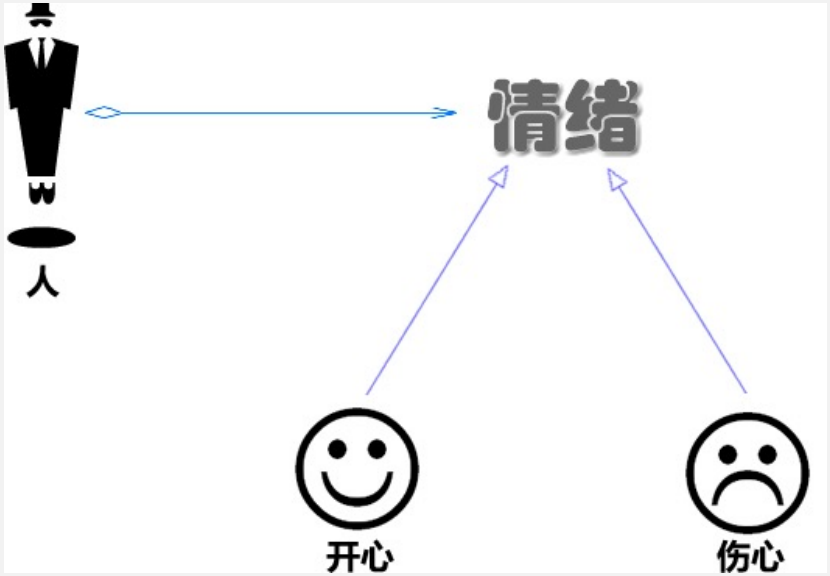
状态模式

- 模式动机
  - 在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化。
  - 在UML中可以使用状态图来描述对象状态的变化。

状态模式

- 模式动机



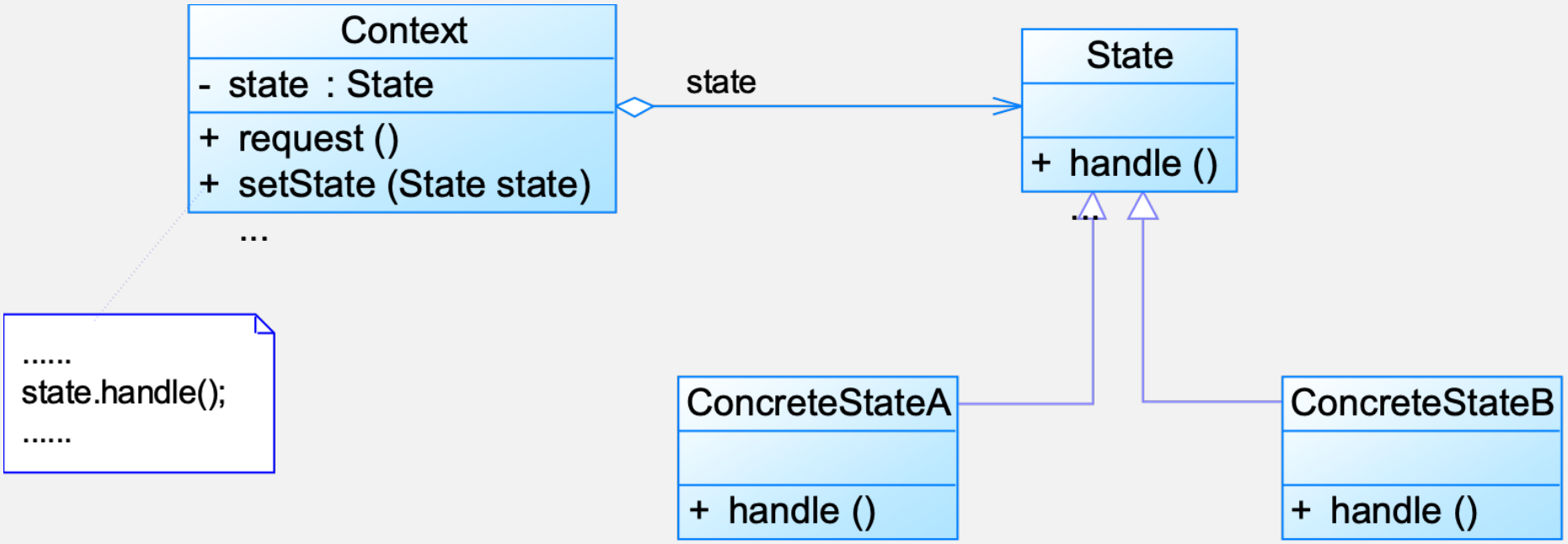


状态模式

- 模式定义
  - 状态模式(State Pattern)：允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象(Objects for States)，状态模式是一种对象行为型模式。
  - State Pattern: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

状态模式

- 模式结构


Context
- state : State
+ request ()
+ setState (State state)
...
state
State
+ handle ()
.....
state.handle();
.....
ConcreteStateA
+ handle ()
ConcreteStateB
+ handle ()

状态模式

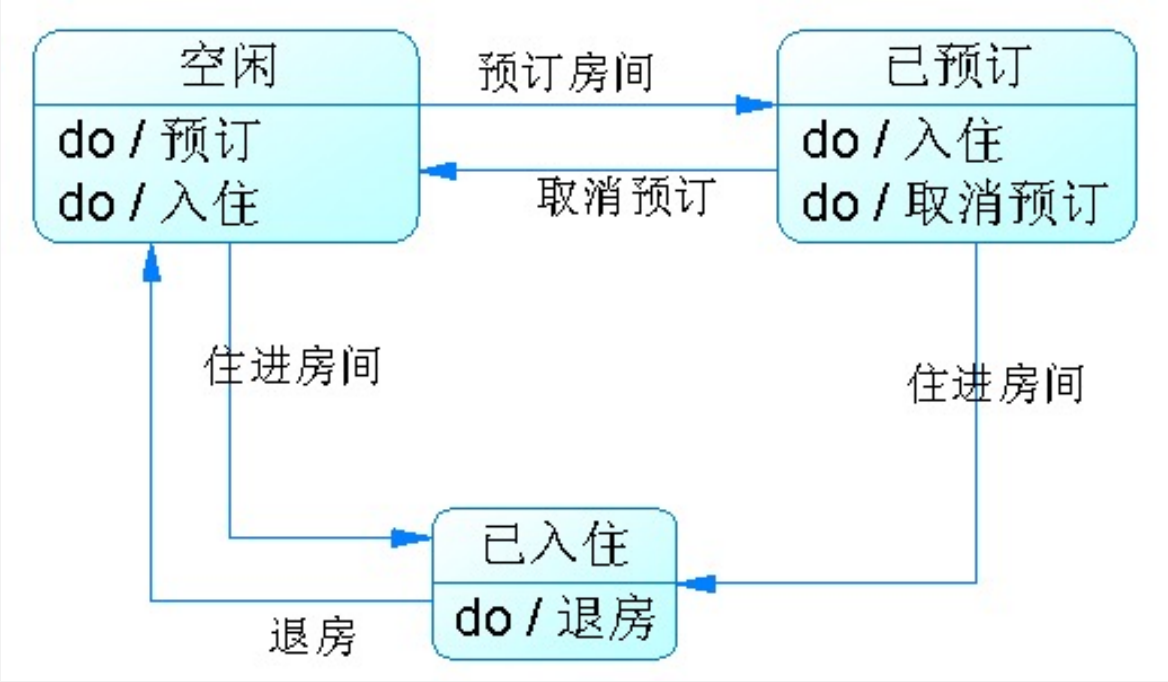
- 模式结构
  - 状态模式包含如下角色:
    - Context: 环境类
    - State: 抽象状态类
    - ConcreteState: 具体状态类

状态模式

- 模式分析
  - 状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。
  - 状态模式的关键是引入了一个抽象类来专门表示对象的状态，这个类我们叫做抽象状态类，而对象的每一种具体状态类都继承了该类，并在不同具体状态类中实现了不同状态的行为，包括各种状态之间的转换。

状态模式

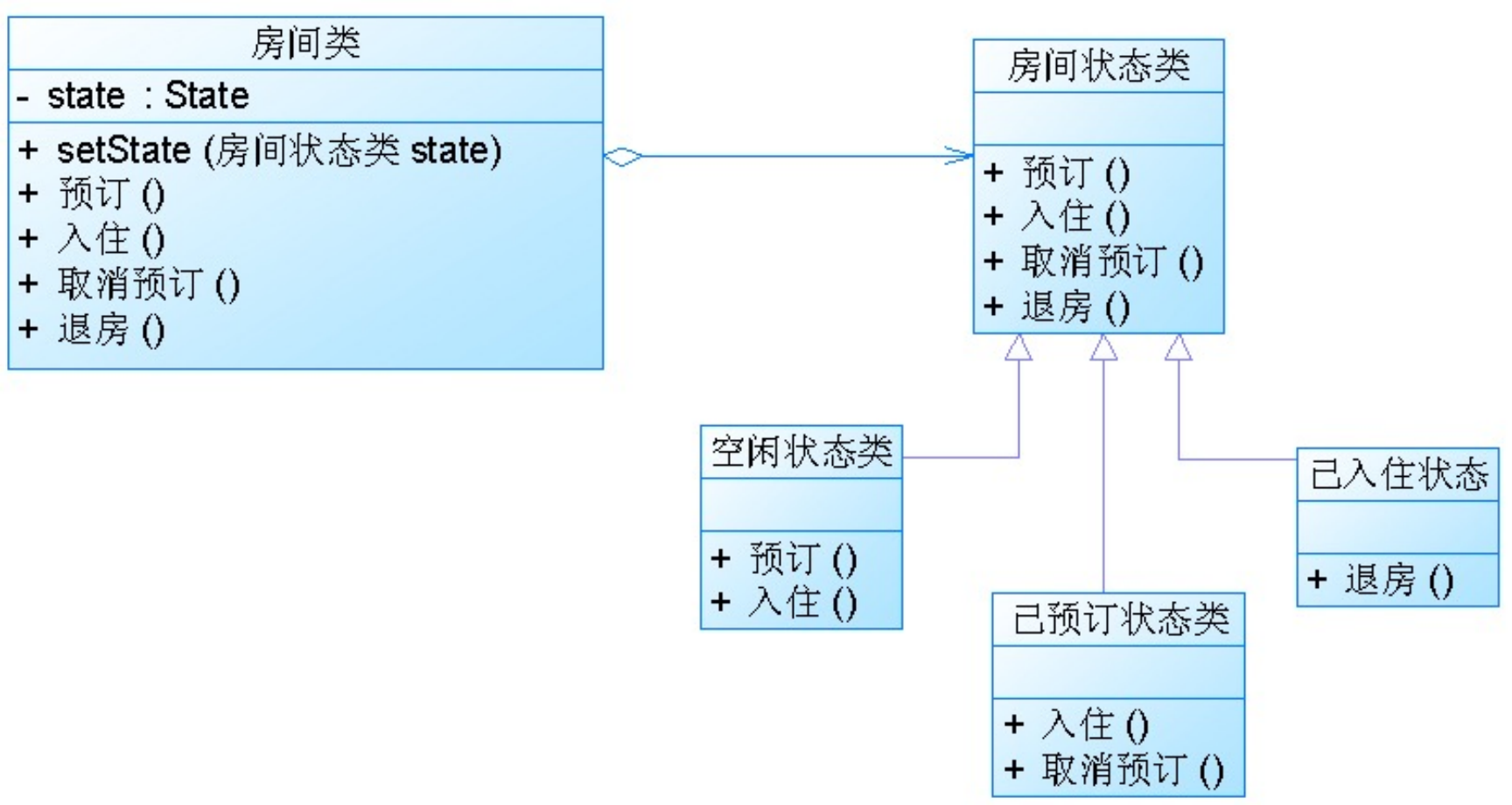
- 模式分析



```
……
if(state=="空闲")
{
        if(预订房间)
        {
                预订操作;
                state="已预订";
        }
        else if(住进房间)
        {
                入住操作;
                state="已入住";
        }
}
else if(state=="已预订")
{
        if(住进房间)
        {
                入住操作;
                state="已入住";
        }
        else if(取消预订)
        {
                取消操作;
                state="空闲";
        }
}
……
```

状态模式

- 模式分析


房间类
- state : State
+ setState (房间状态类 state)
+ 预订 ()
+ 入住 ()
+ 取消预订 ()
+ 退房 ()
房间状态类
+ 预订 ()
+ 入住 ()
+ 取消预订 ()
+ 退房 ()
空闲状态类
+ 预订 ()
+ 入住 ()
已预订状态类
+ 入住 ()
+ 取消预订 ()
已入住状态
+ 退房 ()

状态模式

- 模式分析
  - 使用状态模式重构之后的代码：

```
//重构之后的“空闲状态类”示例代码
……
if(预订房间)
{
        预订操作;
        context.setState(new 已预订状态类());
}
else if(住进房间)
{
        入住操作;
        context.setState(new 已入住状态类());
}
……
```

状态模式

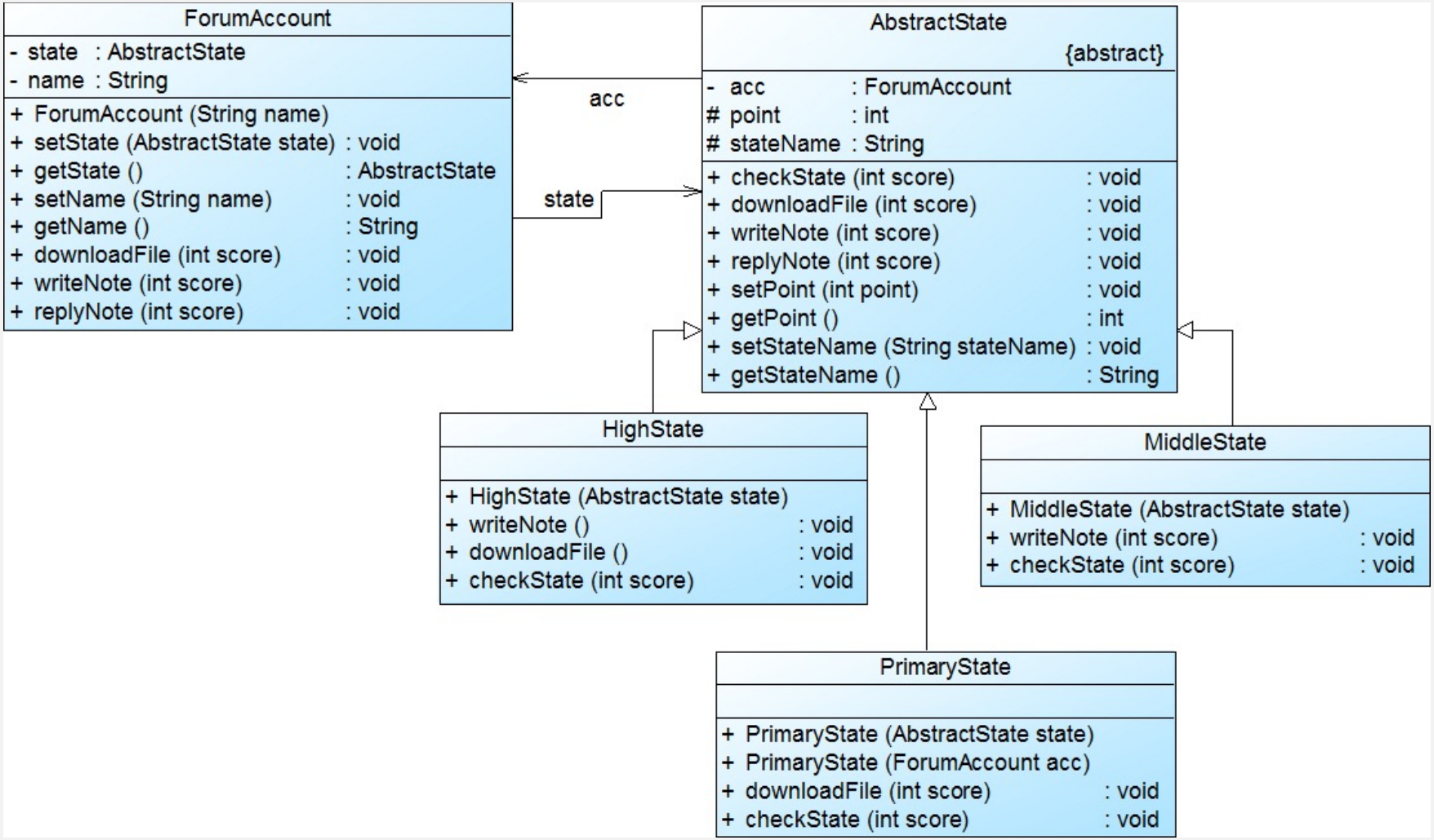
- 模式分析
  - 在状态模式结构中需要理解环境类与抽象状态类的作用：
    - 环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器(State Manager)的角色，可以在环境类中对状态进行切换操作。
    - 抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件：这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

状态模式

- 状态模式实例与解析
  - 实例：论坛用户等级
    - 在某论坛系统中，用户可以发表留言，发表留言将增加积分；用户也可以回复留言，回复留言也将增加积分；用户还可以下载文件，下载文件将扣除积分。该系统用户分为三个等级，分别是新手、高手和专家，这三个等级对应三种不同的状态，这三种状态分别定义如下：
    - (1) 如果积分小于100分，则为新手状态，用户可以发表留言、回复留言，但是不能下载文件。如果积分大于等于1000分，则转换为专家状态；如果积分大于等于100分，则转换为高手状态。
    - (2) 如果积分大于等于100分但小于1000分，则为高手状态，用户可以发表留言、回复留言，还可以下载文件，而且用户在发表留言时可以获取双倍积分。如果积分小于100分，则转换为新手状态；如果积分大于等于1000分，则转换为专家状态；如果下载文件后积分小于0，则不能下载该文件。
    - (3) 如果积分大于等于1000分，则为专家状态，用户可以发表留言、回复留言和下载文件，用户除了在发表留言时可以获取双倍积分外，下载文件只扣除所需积分的一半。如果积分小于100分，则转换为新手状态；如果积分小于1000分，但大于等于100，则转换为高手状态；如果下载文件后积分小于0，则不能下载该文件。

状态模式

- 状态模式实例与解析



状态模式

- 模式优缺点
  - 状态模式的优点
    - 封装了转换规则。
    - 枚举可能的状态，在枚举状态之前需要确定状态种类。
    - 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
    - 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
    - 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

状态模式

- 模式优缺点
  - 状态模式的缺点
    - 状态模式的使用必然会增加系统类和对象的个数。
    - 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
    - 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。

状态模式

- 模式适用环境
  - 在以下情况下可以使用状态模式：
    - 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
    - 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态。

状态模式

- 模式应用
  - (1) 状态模式在工作流或游戏等类型的软件中得以广泛使用，甚至可以用于这些系统的核心功能设计，如在政府OA办公系统中，一个批文的状态有多种：尚未办理；正在办理；正在批示；正在审核；已经完成等各种状态，而且批文状态不同时对批文的操作也有所差异。使用状态模式可以描述工作流对象（如批文）的状态转换以及不同状态下它所具有的行为。

状态模式

- 模式应用
  - (2) 在目前主流的RPG（Role Play Game，角色扮演游戏）中，使用状态模式可以对游戏角色进行控制，游戏角色的升级伴随着其状态的变化和行为的变化。对于游戏程序本身也可以通过状态模式进行总控，一个游戏活动包括开始、运行、结束等状态，通过对状态的控制可以控制系统的行为，决定游戏的各个方面，因此可以使用状态模式对整个游戏的架构进行设计与实现。

状态模式

- 模式扩展
  - 共享状态
    - 在有些情况下多个环境对象需要共享同一个状态，如果希望在系统中实现多个环境对象实例共享一个或多个状态对象，那么需要将这些状态对象定义为环境的静态成员对象。

状态模式

- 模式扩展
  - 简单状态模式与可切换状态的状态模式
    - (1) 简单状态模式：简单状态模式是指状态都相互独立，状态之间无须进行转换的状态模式，这是最简单的一种状态模式。对于这种状态模式，每个状态类都封装与状态相关的操作，而无须关心状态的切换，可以在客户端直接实例化状态类，然后将状态对象设置到环境类中。如果是这种简单的状态模式，它遵循“开闭原则”，在客户端可以针对抽象状态类进行编程，而将具体状态类写到配置文件中，同时增加新的状态类对原有系统也不造成任何影响。

状态模式

- 模式扩展
  - 简单状态模式与可切换状态的状态模式
    - (2) 可切换状态的状态模式：大多数的状态模式都是可以切换状态的状态模式，在实现状态切换时，在具体状态类内部需要调用环境类Context的setState()方法进行状态的转换操作，在具体状态类中可以调用到环境类的方法，因此状态类与环境类之间通常还存在关联关系或者依赖关系。通过在状态类中引用环境类的对象来回调环境类的setState()方法实现状态的切换。在这种可以切换状态的状态模式中，增加新的状态类可能需要修改其他某些状态类甚至环境类的源代码，否则系统无法切换到新增状态。

状态小结

- 状态模式允许一个对象在其内部状态改变时改变它的行为，对象看起来似乎修改了它的类。其别名为状态对象，状态模式是一种对象行为型模式。
- 状态模式包含三个角色：环境类又称为上下文类，它是拥有状态的对象，在环境类中维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时，它是一个State子类的对象，可以定义初始状态；抽象状态类用于定义一个接口以封装与环境类的一个特定状态相关的行为；具体状态类是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类其行为有所不同。
- 状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。

状态小结

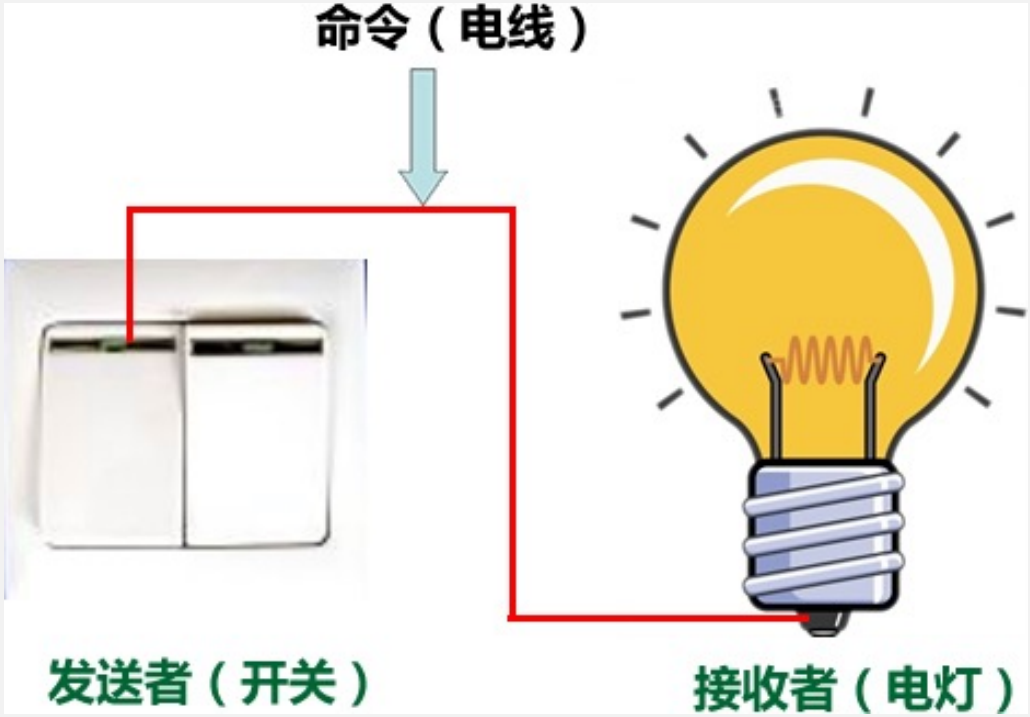
- 状态模式的主要优点在于封装了转换规则，并枚举可能的状态，它将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为，还可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数；其缺点在于使用状态模式会增加系统类和对象的个数，且状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，对于可以切换状态的状态模式不满足“开闭原则”的要求。
- 状态模式适用情况包括：对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为；代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。

命令模式

- 模式动机
  - 在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式

- 模式动机



命令模式

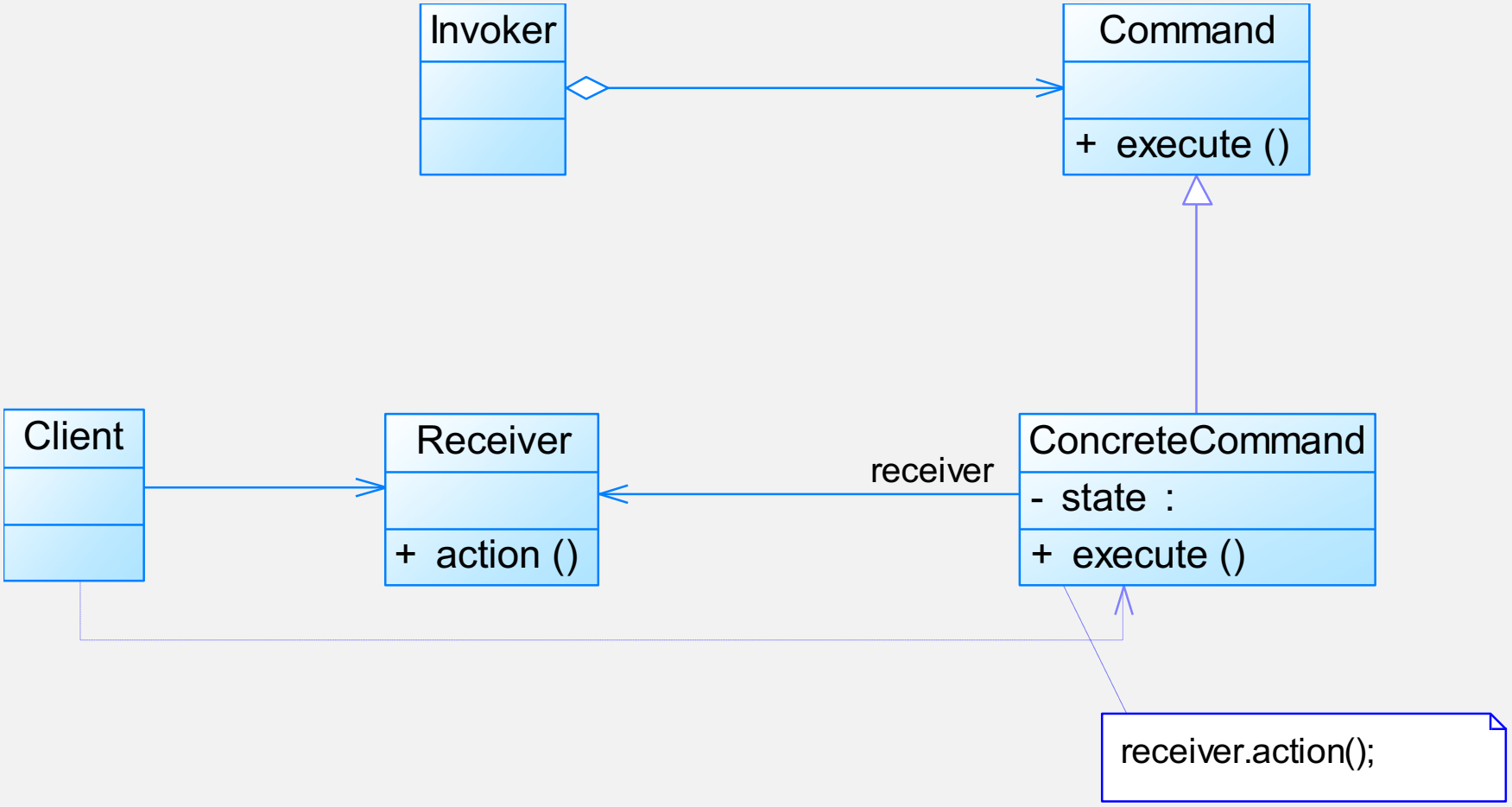
- 模式动机
  - 命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

命令模式

- 模式定义
  - 命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。
  - Command Pattern: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

命令模式

- 模式结构


Invoker
Command
+ execute ()
Client
Receiver
+ action ()
receiver
ConcreteCommand
- state :
+ execute ()
receiver.action();

命令模式

- 模式结构
  - 命令模式包含如下角色：
    - Command: 抽象命令类
    - ConcreteCommand: 具体命令类
    - Invoker: 调用者
    - Receiver: 接收者
    - Client:客户类

命令模式

- 模式分析
  - 命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。
  - 每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。
  - 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令模式

- 模式分析
  - 命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
  - 命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

命令模式

- 模式分析
  - 典型的抽象命令类代码：

```
public abstract class Command
{
    public abstract void execute();
}
```

命令模式

- 模式分析
  - 典型的调用者代码：

```
public class Invoker
{
        private Command command;

        public Invoker(Command command)
        {
                this.command=command;
        }

        public void setCommand(Command command)
        {
                this.command=command;
        }

        //业务方法，用于调用命令类的方法
        public void call()
        {
                command.execute();
        }
}
```

命令模式

- 模式分析
  - 典型的具体命令类代码：

```
public class ConcreteCommand extends Command
{
    private Receiver receiver;
    public void execute()
    {
        receiver.action();
    }
}
```

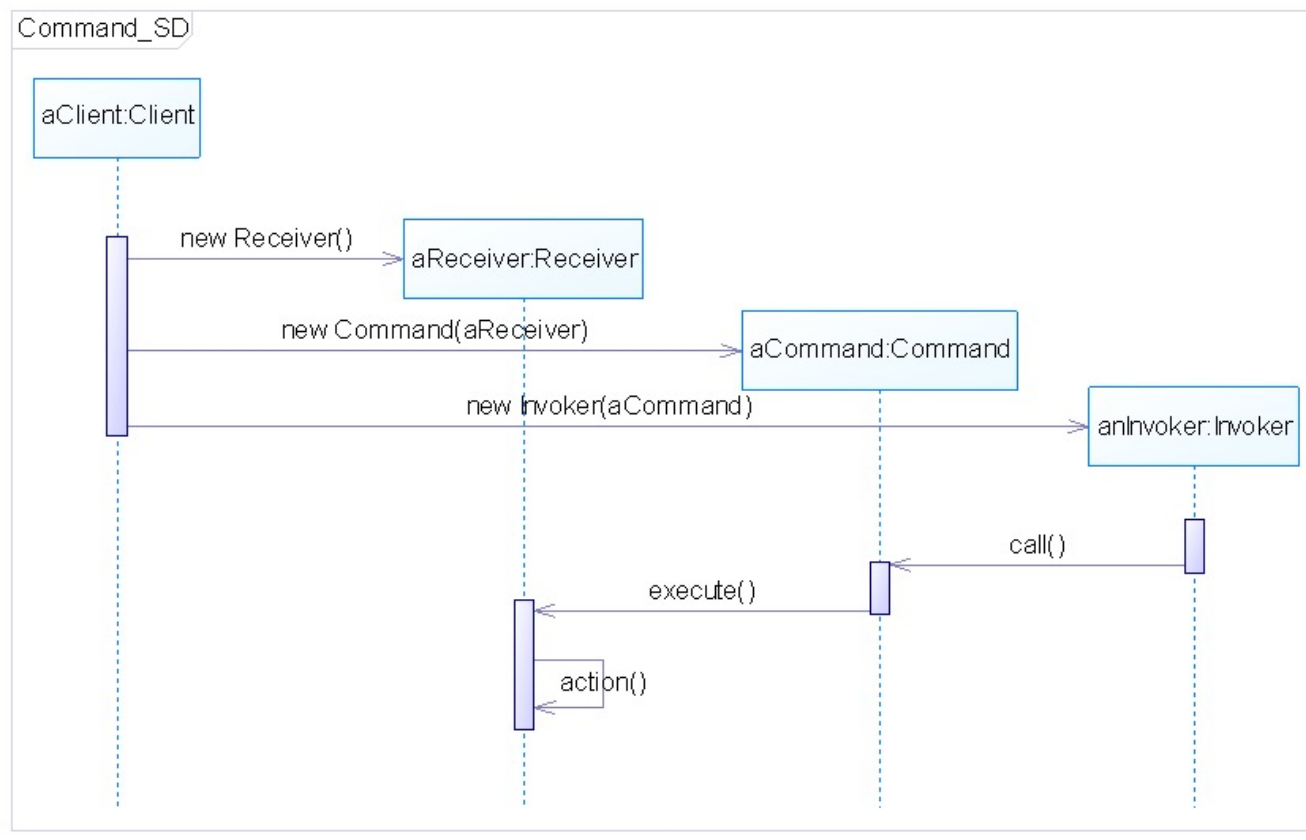
命令模式

- 模式分析
  - 典型的请求接收者代码：

```
public class Receiver
{
    public void action()
    {
        //具体操作
    }
}
```

命令模式

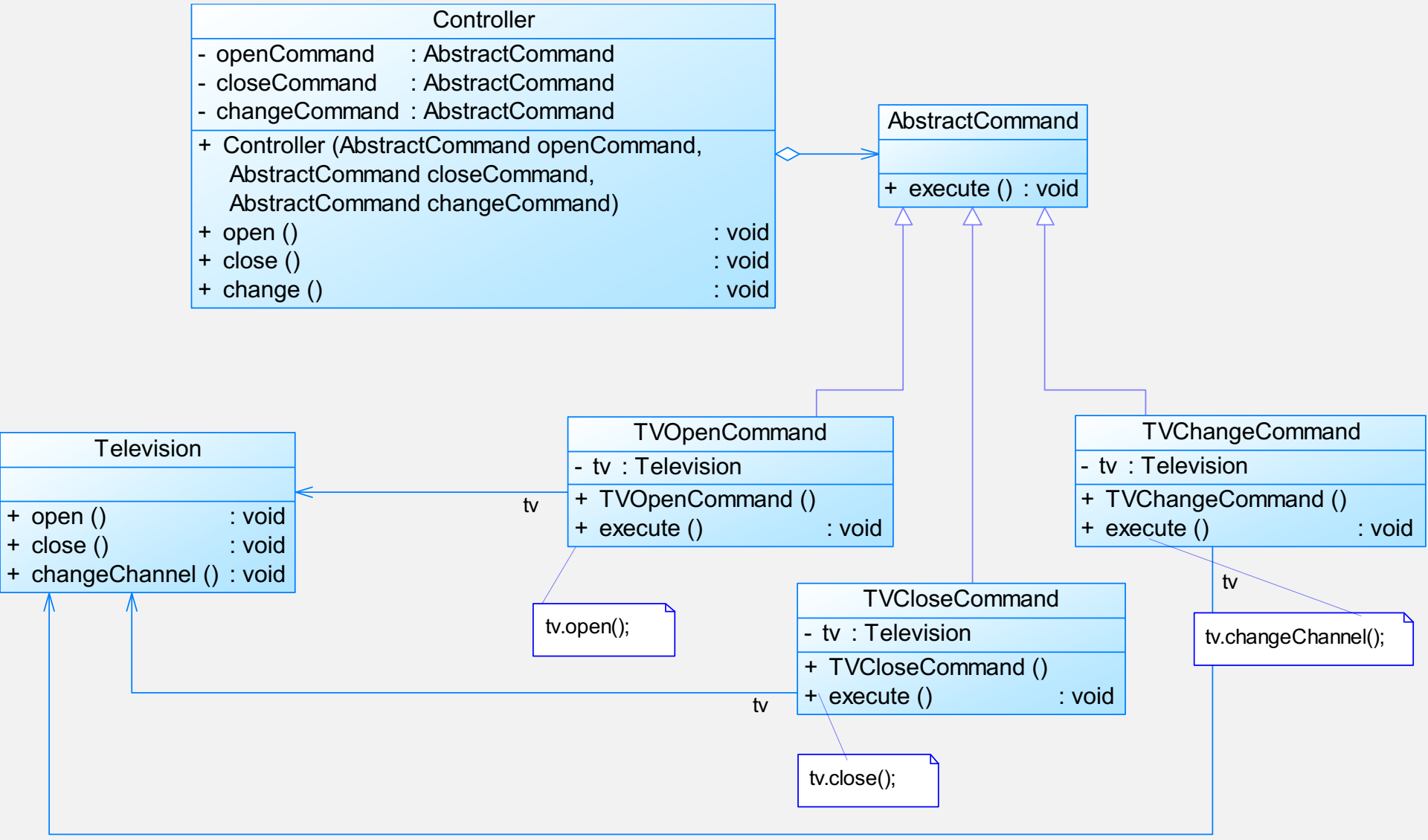
- 模式分析
  - 命令模式顺序图：



命令模式

- 命令模式实例与解析
  - 实例一：电视机遥控器
    - 电视机是请求的接收者，遥控器是请求的发送者，遥控器上有一些按钮，不同的按钮对应电视机的不同操作。抽象命令角色由一个命令接口来扮演，有三个具体的命令类实现了抽象命令接口，这三个具体命令类分别代表三种操作：打开电视机、关闭电视机和切换频道。显然，电视机遥控器就是一个典型的命令模式应用实例。

- 命令模式实例与解析
  - 实例一： 电视机遥控器

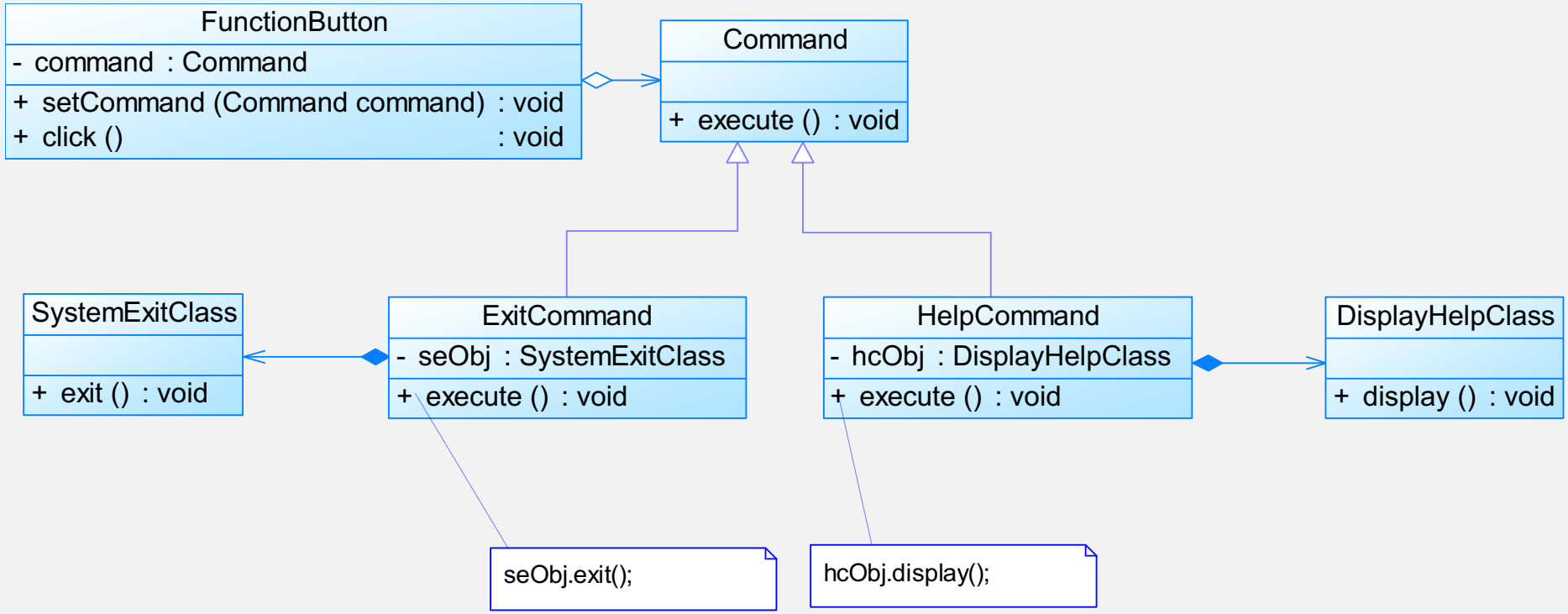


命令模式

- 命令模式实例与解析
  - 实例二：功能键设置
    - 为了用户使用方便，某系统提供了一系列功能键，用户可以自定义功能键的功能，如功能键FunctionButton可以用于退出系统(SystemExitClass)，也可以用于打开帮助界面(DisplayHelpClass)。用户可以通过修改配置文件来改变功能键的用途，现使用命令模式来设计该系统，使得功能键类与功能类之间解耦，相同的功能键可以对应不同的功能。

命令模式

- 命令模式实例与解析
  - 实例二：功能键设置



命令模式

- 模式优缺点
  - 命令模式的优点
    - 降低系统的耦合度。
    - 新的命令可以很容易地加入到系统中。
    - 可以比较容易地设计一个命令队列和宏命令（组合命令）。
    - 可以方便地实现对请求的Undo和Redo。

命令模式

- 模式优缺点
  - 命令模式的缺点
    - 使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

命令模式

- 模式适用环境
  - 在以下情况下可以使用命令模式：
    - 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。
    - 系统需要在不同的时间指定请求、将请求排队和执行请求。
    - 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。
    - 系统需要将一组操作组合在一起，即支持宏命令。

命令模式

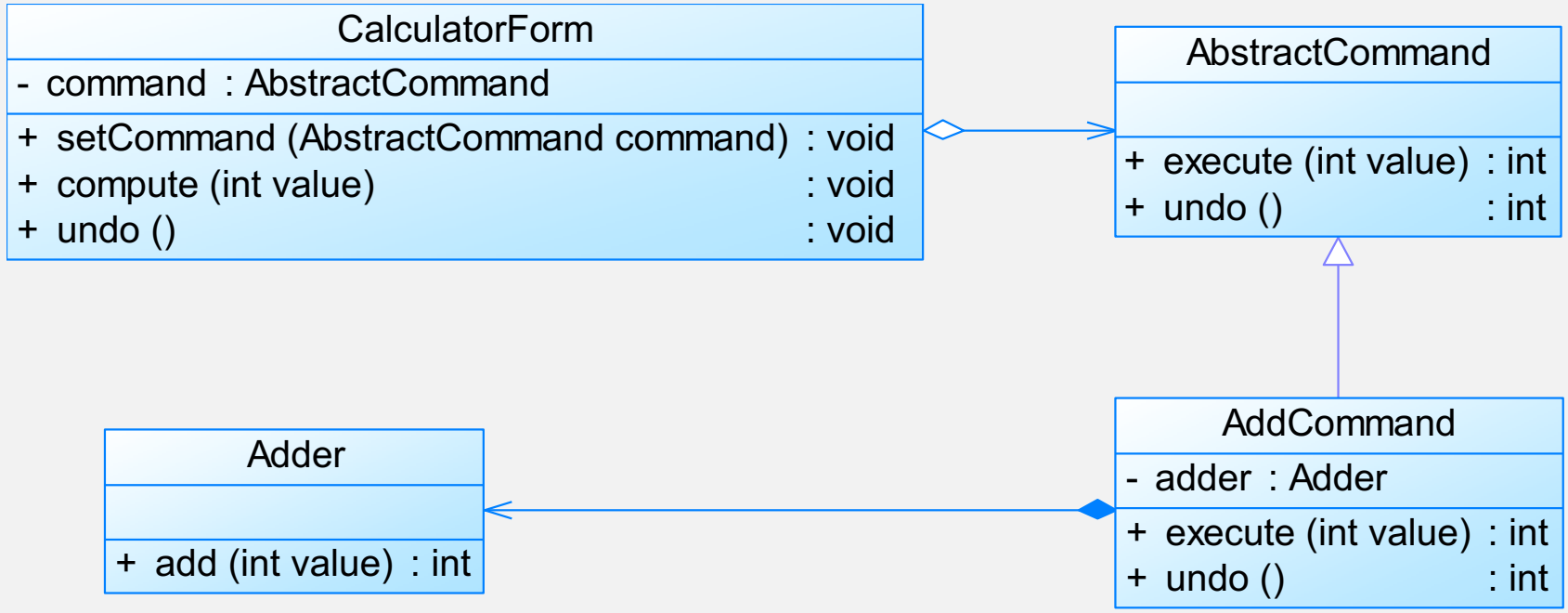
- 模式应用
  - (1) Java语言使用命令模式实现AWT/Swing GUI的委派事件模型 (Delegation Event Model, DEM) 。
    - 在AWT/Swing中，Frame、Button等界面组件是请求发送者，而AWT提供的事件监听器接口和事件适配器类是抽象命令接口，用户可以自己写抽象命令接口的子类来实现事件处理，即实现具体命令类，而在具体命令类中可以调用业务处理方法来实现该事件的处理。对于界面组件而言，只需要了解命令接口即可，无须关心接口的实现，组件类并不关心实际操作，而操作由用户来实现。

命令模式

- 模式应用
  - (2) 很多系统都提供了宏命令功能，如UNIX平台下的Shell编程，可以将多条命令封装在一个命令对象中，只需要一条简单的命令即可执行一个命令序列，这也是命令模式的应用实例之一。

命令模式

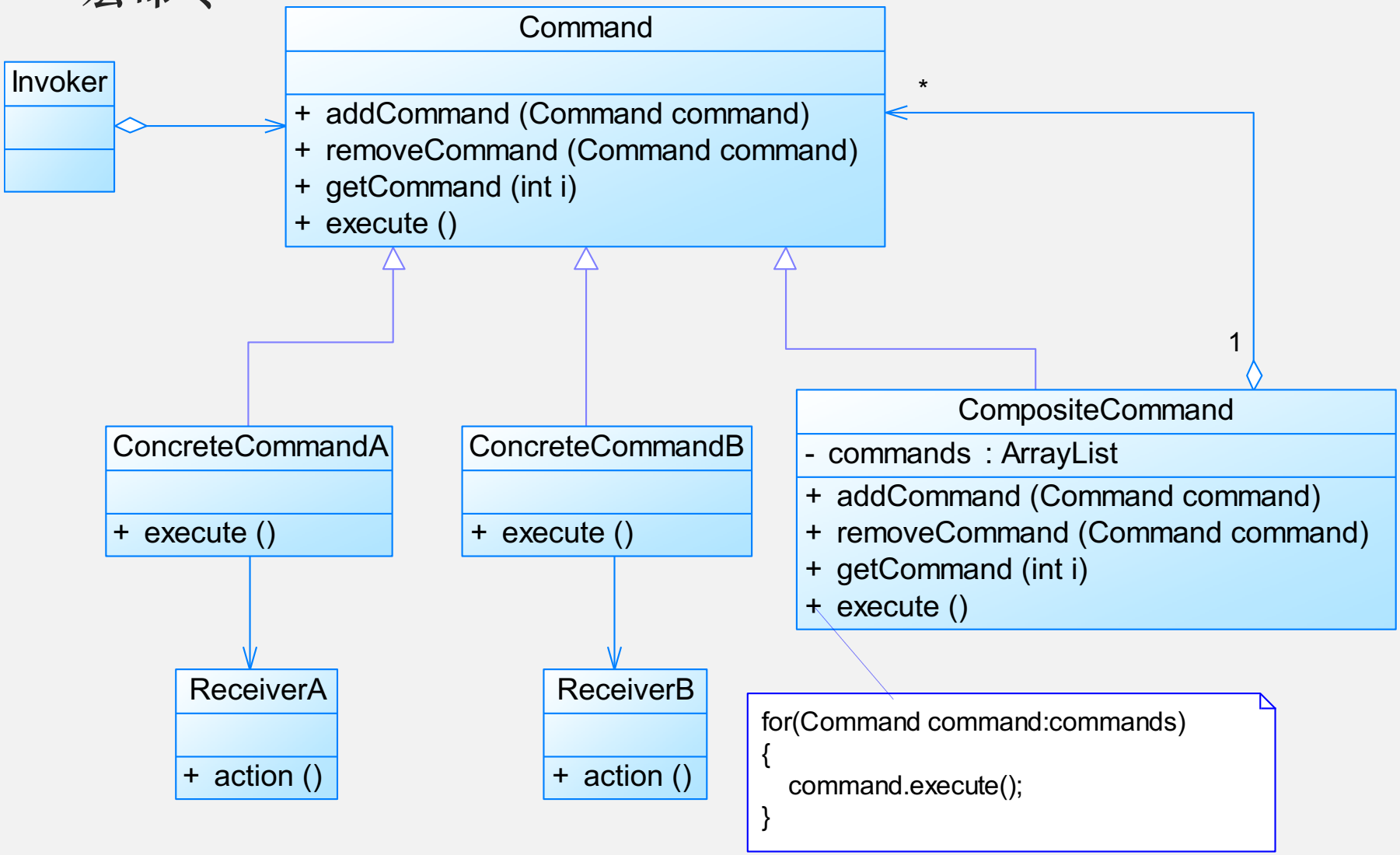
- 模式扩展
  - 撤销操作的实现



命令模式

- 模式扩展
  - 宏命令又称为组合命令，它是命令模式和组合模式联用的产物。
  - 宏命令也是一个具体命令，不过它包含了对其他命令对象的引用，在调用宏命令的execute()方法时，将递归调用它所包含的每个成员命令的execute()方法，一个宏命令的成员对象可以是简单命令，还可以继续是宏命令。执行一个宏命令将执行多个具体命令，从而实现对命令的批处理。

- 模式扩展
  - 宏命令


Command
+ addCommand (Command command)
+ removeCommand (Command command)
+ getCommand (int i)
+ execute ()
Invoker
*
1
ConcreteCommandA
+ execute ()
ConcreteCommandB
+ execute ()
CompositeCommand
- commands : ArrayList
+ addCommand (Command command)
+ removeCommand (Command command)
+ getCommand (int i)
+ execute ()
ReceiverA
+ action ()
ReceiverB
+ action ()
for(Command command:commands)
{
command.execute();
}

命令小结

- 在命令模式中，将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作模式或事务模式。
- 命令模式包含四个角色：抽象命令类中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作；具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中；调用者即请求的发送者，又称为请求者，它通过命令对象来执行请求；接收者执行与请求相关的操作，它具体实现对请求的业务处理。

命令小结

- 命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。
- 命令模式的主要优点在于降低系统的耦合度，增加新的命令很方便，而且可以比较容易地设计一个命令队列和宏命令，并方便地实现对请求的撤销和恢复；其主要缺点在于可能会导致某些系统有过多的具体命令类。
- 命令模式适用情况包括：需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互；需要在不同的时间指定请求、将请求排队和执行请求；需要支持命令的撤销操作和恢复操作；需要将一组操作组合在一起，即支持宏命令。