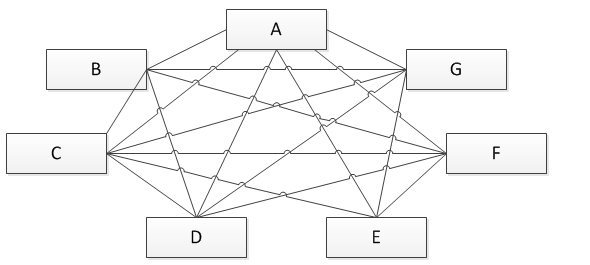
在阎宏博士的《JAVA与模式》一书中开头是这样描述调停者（Mediator）模式的：

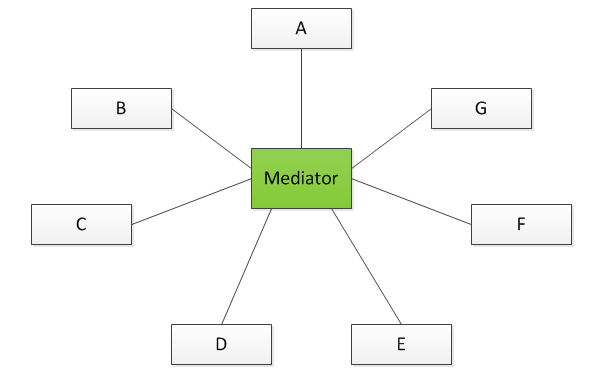
**调停者模式是对象的行为模式。调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显引用。从而使它们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立地变化。**

**为什么需要调停者**

如下图所示，这个示意图中有大量的对象，这些对象既会影响别的对象，又会被别的对象所影响，因此常常叫做同事(Colleague)对象。这些同事对象通过彼此的相互作用形成系统的行为。从图中可以看出，几乎每一个对象都需要与其他的对象发生相互作用，而这种相互作用表现为一个对象与另一个对象的直接耦合。这就是过度耦合的系统。



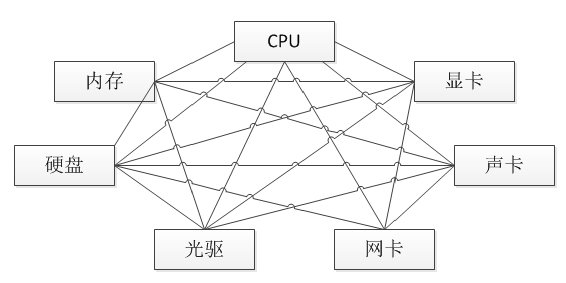
通过引入调停者对象(Mediator)，可以将系统的网状结构变成以中介者为中心的星形结构，如下图所示。在这个星形结构中，同事对象不再通过直接的联系与另一个对象发生相互作用；相反的，它通过调停者对象与另一个对象发生相互作用。调停者对象的存在保证了对象结构上的稳定，也就是说，系统的结构不会因为新对象的引入造成大量的修改工作。



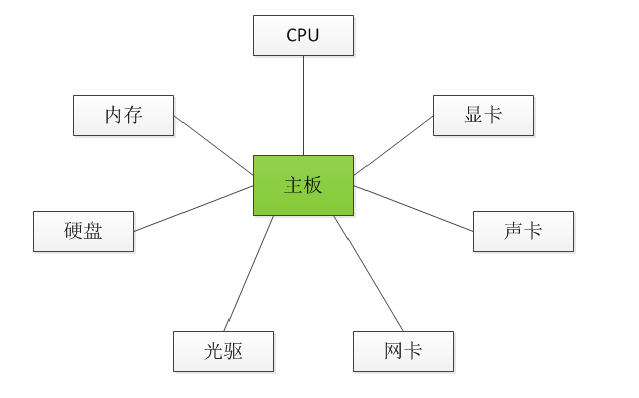
　　一个好的面向对象的设计可以使对象之间增加协作性(Collaboration)，减少耦合度(Couping)。一个深思熟虑的设计会把一个系统分解为一群相互协作的同事对象，然后给每一个同事对象以独特的责任，恰当的配置它们之间的协作关系，使它们可以在一起工作。

**如果没有主板**

　　大家都知道，电脑里面各个配件之间的交互，主要是通过主板来完成的。如果电脑里面没有了主板，那么各个配件之间就必须自行相互交互，以互相传送数据。而且由于各个配件的接口不同，相互之间交互时，还必须把数据接口进行转换才能匹配上。

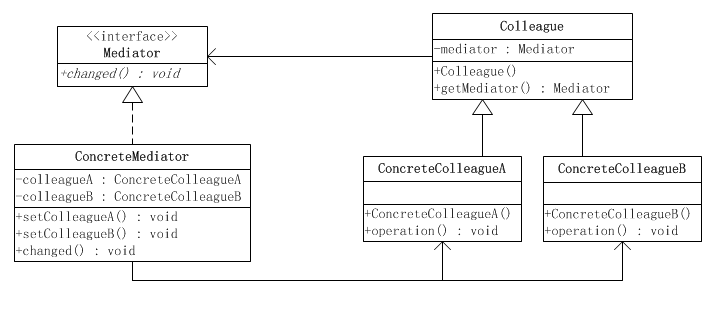


　　所幸是有了主板，各个配件的交互完全通过主板来完成，每个配件都只需要和主板交互，而主板知道如何跟所有的配件打交道，这样就简单多了。



**调停者模式的结构**

　　调停者模式的示意性类图如下所示：



　　调停者模式包括以下角色：

　　●　　**抽象调停者(Mediator)角色：**定义出同事对象到调停者对象的接口，其中主要方法是一个（或多个）事件方法。

　　●　　**具体调停者(ConcreteMediator)角色：**实现了抽象调停者所声明的事件方法。具体调停者知晓所有的具体同事类，并负责具体的协调各同事对象的交互关系。

　　●　　**抽象同事类(Colleague)角色：**定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。

　　●　　**具体同事类(ConcreteColleague)角色：**所有的具体同事类均从抽象同事类继承而来。实现自己的业务，在需要与其他同事通信的时候，就与持有的调停者通信，调停者会负责与其他的同事交互。

**源代码**

　　抽象调停者类

public interface Mediator {

/\*\*

\* 同事对象在自身改变的时候来通知调停者方法

\* 让调停者去负责相应的与其他同事对象的交互

\*/

public void changed(Colleague c);

}

　　具体调停者类

public class ConcreteMediator implements Mediator {

//持有并维护同事A

private ConcreteColleagueA colleagueA;

//持有并维护同事B

private ConcreteColleagueB colleagueB;

public void setColleagueA(ConcreteColleagueA colleagueA) {

this.colleagueA = colleagueA;

}

public void setColleagueB(ConcreteColleagueB colleagueB) {

this.colleagueB = colleagueB;

}

@Override

public void changed(Colleague c) {

/\*\*

\* 某一个同事类发生了变化，通常需要与其他同事交互

\* 具体协调相应的同事对象来实现协作行为

\*/

}

}

　　抽象同事类

public abstract class Colleague {

//持有一个调停者对象

private Mediator mediator;

/\*\*

\* 构造函数

\*/

public Colleague(Mediator mediator){

this.mediator = mediator;

}

/\*\*

\* 获取当前同事类对应的调停者对象

\*/

public Mediator getMediator() {

return mediator;

}

}

　　具体同事类

public class ConcreteColleagueA extends Colleague {

public ConcreteColleagueA(Mediator mediator) {

super(mediator);

}

/\*\*

\* 示意方法，执行某些操作

\*/

public void operation(){

//在需要跟其他同事通信的时候，通知调停者对象

getMediator().changed(this);

}

}

public class ConcreteColleagueB extends Colleague {

public ConcreteColleagueB(Mediator mediator) {

super(mediator);

}

/\*\*

\* 示意方法，执行某些操作

\*/

public void operation(){

//在需要跟其他同事通信的时候，通知调停者对象

getMediator().changed(this);

}

}

**使用电脑来看电影**

　　在日常生活中，我们经常使用电脑来看电影，把这个过程描述出来，简化后假定会有如下的交互过程：

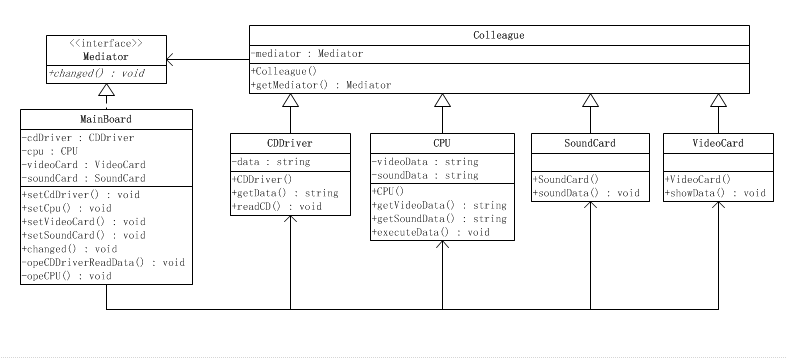
　　（1）首先是光驱要读取光盘上的数据，然后告诉主板，它的状态改变了。

　　（2）主板去得到光驱的数据，把这些数据交给CPU进行分析处理。

　　（3）CPU处理完后，把数据分成了视频数据和音频数据，通知主板，它处理完了。

　　（4）主板去得到CPU处理过后的数据，分别把数据交给显卡和声卡，去显示出视频和发出声音。

　　要使用调停者模式来实现示例，那就要区分出同事对象和调停者对象。很明显，主板是调停者，而光驱、声卡、CPU、显卡等配件，都是作为同事对象。



**源代码**

　　抽象同事类

public abstract class Colleague {

//持有一个调停者对象

private Mediator mediator;

/\*\*

\* 构造函数

\*/

public Colleague(Mediator mediator){

this.mediator = mediator;

}

/\*\*

\* 获取当前同事类对应的调停者对象

\*/

public Mediator getMediator() {

return mediator;

}

}

　　同事类——光驱

public class CDDriver extends Colleague{

//光驱读取出来的数据

private String data = "";

/\*\*

\* 构造函数

\*/

public CDDriver(Mediator mediator) {

super(mediator);

}

/\*\*

\* 获取光盘读取出来的数据

\*/

public String getData() {

return data;

}

/\*\*

\* 读取光盘

\*/

public void readCD(){

//逗号前是视频显示的数据，逗号后是声音

this.data = "One Piece,海贼王我当定了";

//通知主板，自己的状态发生了改变

getMediator().changed(this);

}

}

　　同事类——CPU

public class CPU extends Colleague {

//分解出来的视频数据

private String videoData = "";

//分解出来的声音数据

private String soundData = "";

/\*\*

\* 构造函数

\*/

public CPU(Mediator mediator) {

super(mediator);

}

/\*\*

\* 获取分解出来的视频数据

\*/

public String getVideoData() {

return videoData;

}

/\*\*

\* 获取分解出来的声音数据

\*/

public String getSoundData() {

return soundData;

}

/\*\*

\* 处理数据，把数据分成音频和视频的数据

\*/

public void executeData(String data){

//把数据分解开，前面是视频数据，后面是音频数据

String[] array = data.split(",");

this.videoData = array[0];

this.soundData = array[1];

//通知主板，CPU完成工作

getMediator().changed(this);

}

}

　　同事类——显卡

public class VideoCard extends Colleague {

/\*\*

\* 构造函数

\*/

public VideoCard(Mediator mediator) {

super(mediator);

}

/\*\*

\* 显示视频数据

\*/

public void showData(String data){

System.out.println("您正在观看的是：" + data);

}

}

　　同事类——声卡

public class SoundCard extends Colleague {

/\*\*

\* 构造函数

\*/

public SoundCard(Mediator mediator) {

super(mediator);

}

/\*\*

\* 按照声频数据发出声音

\*/

public void soundData(String data){

System.out.println("画外音：" + data);

}

}

　　抽象调停者类

public interface Mediator {

/\*\*

\* 同事对象在自身改变的时候来通知调停者方法

\* 让调停者去负责相应的与其他同事对象的交互

\*/

public void changed(Colleague c);

}

　　具体调停者类

public class MainBoard implements Mediator {

//需要知道要交互的同事类——光驱类

private CDDriver cdDriver = null;

//需要知道要交互的同事类——CPU类

private CPU cpu = null;

//需要知道要交互的同事类——显卡类

private VideoCard videoCard = null;

//需要知道要交互的同事类——声卡类

private SoundCard soundCard = null;

public void setCdDriver(CDDriver cdDriver) {

this.cdDriver = cdDriver;

}

public void setCpu(CPU cpu) {

this.cpu = cpu;

}

public void setVideoCard(VideoCard videoCard) {

this.videoCard = videoCard;

}

public void setSoundCard(SoundCard soundCard) {

this.soundCard = soundCard;

}

@Override

public void changed(Colleague c) {

if(c instanceof CDDriver){

//表示光驱读取数据了

this.opeCDDriverReadData((CDDriver)c);

}else if(c instanceof CPU){

this.opeCPU((CPU)c);

}

}

/\*\*

\* 处理光驱读取数据以后与其他对象的交互

\*/

private void opeCDDriverReadData(CDDriver cd){

//先获取光驱读取的数据

String data = cd.getData();

//把这些数据传递给CPU进行处理

cpu.executeData(data);

}

/\*\*

\* 处理CPU处理完数据后与其他对象的交互

\*/

private void opeCPU(CPU cpu){

//先获取CPU处理后的数据

String videoData = cpu.getVideoData();

String soundData = cpu.getSoundData();

//把这些数据传递给显卡和声卡展示出来

videoCard.showData(videoData);

soundCard.soundData(soundData);

}

}

　　客户端类

public class Client {

public static void main(String[] args) {

//创建调停者——主板

MainBoard mediator = new MainBoard();

//创建同事类

CDDriver cd = new CDDriver(mediator);

CPU cpu = new CPU(mediator);

VideoCard vc = new VideoCard(mediator);

SoundCard sc = new SoundCard(mediator);

//让调停者知道所有同事

mediator.setCdDriver(cd);

mediator.setCpu(cpu);

mediator.setVideoCard(vc);

mediator.setSoundCard(sc);

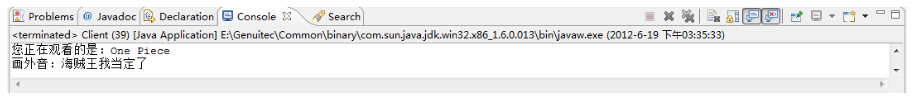
//开始看电影，把光盘放入光驱，光驱开始读盘

cd.readCD();

}

}

运行结果如下：



**调停者模式的优点**

　　●　　**松散耦合**

　　调停者模式通过把多个同事对象之间的交互封装到调停者对象里面，从而使得同事对象之间松散耦合，基本上可以做到互补依赖。这样一来，同事对象就可以独立地变化和复用，而不再像以前那样“牵一处而动全身”了。

　　●　**集中控制交互**

　　多个同事对象的交互，被封装在调停者对象里面集中管理，使得这些交互行为发生变化的时候，只需要修改调停者对象就可以了，当然如果是已经做好的系统，那么就扩展调停者对象，而各个同事类不需要做修改。

　　●　　**多对多变成一对多**

　　没有使用调停者模式的时候，同事对象之间的关系通常是多对多的，引入调停者对象以后，调停者对象和同事对象的关系通常变成双向的一对多，这会让对象的关系更容易理解和实现。

**调停者模式的缺点**

　　调停者模式的一个潜在缺点是，过度集中化。如果同事对象的交互非常多，而且比较复杂，当这些复杂性全部集中到调停者的时候，会导致调停者对象变得十分复杂，而且难于管理和维护。