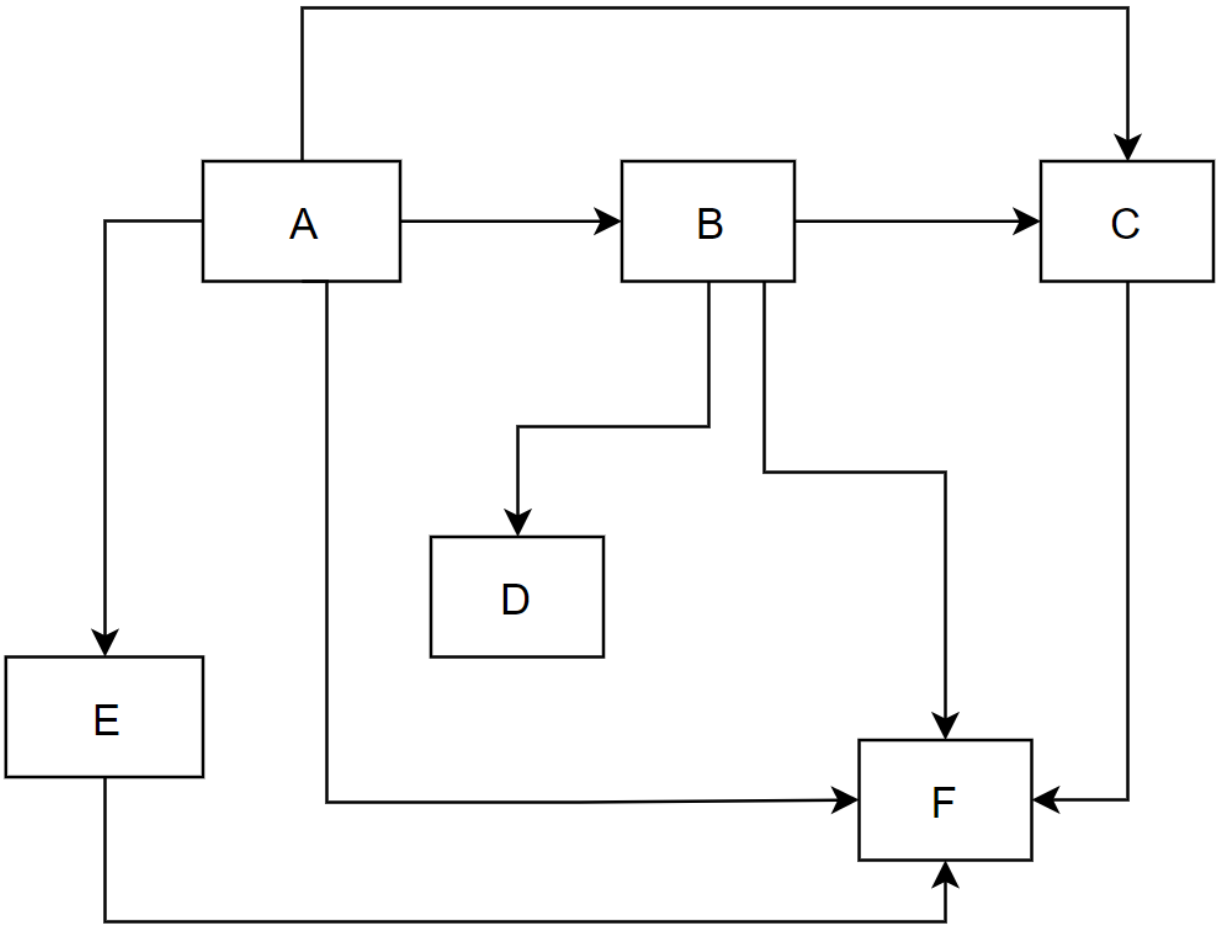
14 中介者模式

平时我们大概能记住10个朋友的电话或者10家餐馆的位置，程序中一个对象也许会和10个对象打交道，保持10个对象的引用。并且当程序的规模增大，对象会越来越多，他们的关系也会越来越复杂，很容易形成复杂的网状的交叉引用。

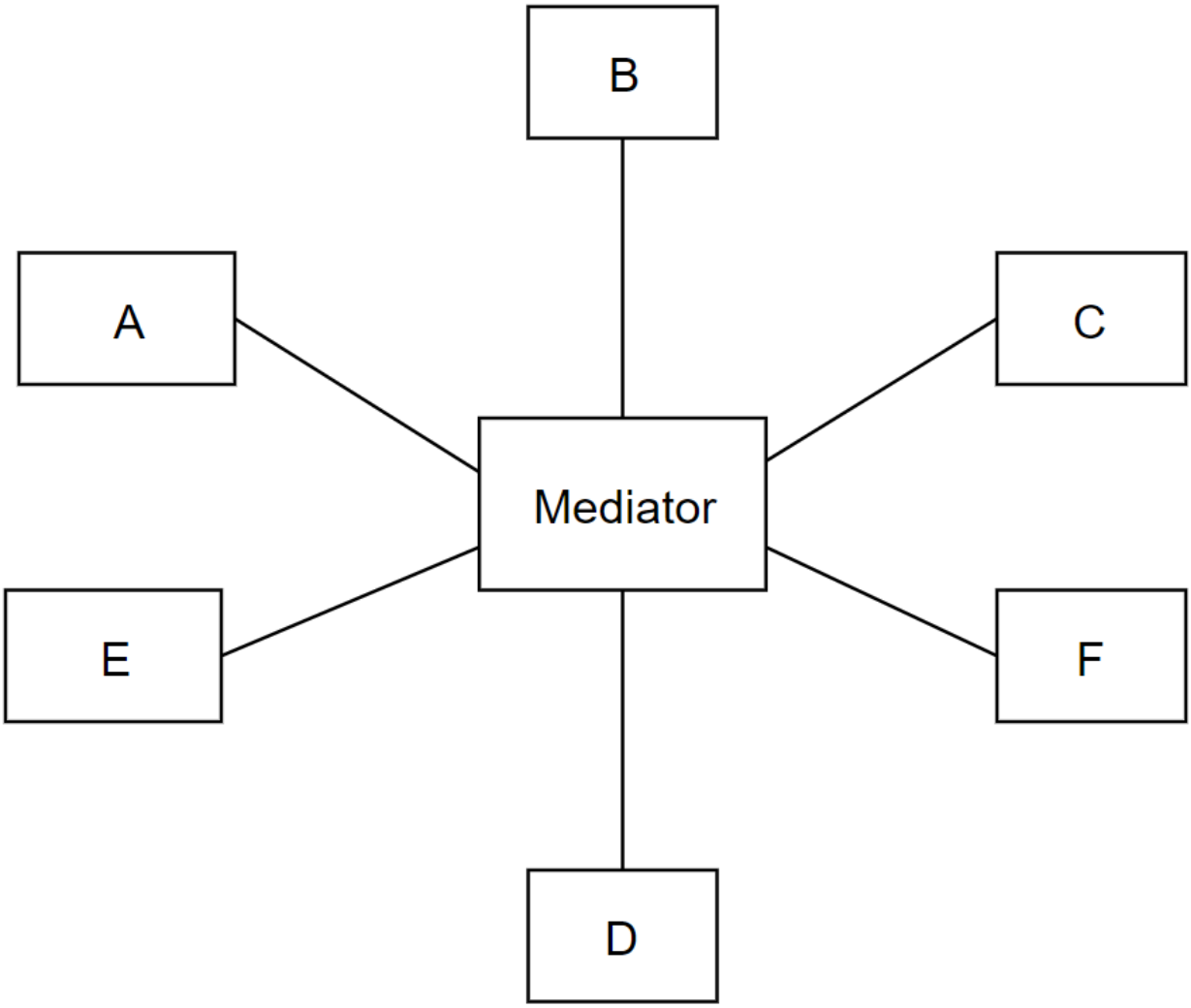
此时，当我们改变或者删除一个对象的时候很可能需要通知其他引用该对象的对象，需要十分小心。



虽然面向对象设计鼓励将行为分布到各个对象，以细粒度来组合，增强对象可复用性，但是如果拆分粒度过细，很可能导致对象之间的联系激增，反过来降低其复用性

中介者模式的作用就是用来解除对象和对象之间的紧耦合关系

增加中介者对象后，所有相关对象通过中介通信，而非直接引用，当对象改变时只需要通知中介处理即可，最终将网状的多对多关系变成了简单的一对多关系：



使用中介者模式改进之后，A发生改变时只需要通知中介者对象即可

现实中的中介者

1. 机场指挥塔

如果没有指挥塔的存在，每一架飞机都要和方圆百里的其他飞机直接通信才能确定航线和飞行情况，想想就复杂。现实中通过指挥塔每个飞机只需要和指挥塔通信，通过他作为调停者来知道其他飞机情况并合理安排起降时间

2. 博彩公司

在世界杯期间购买彩票，如果没有博彩公司介入，上千万人一起计算赔率和输赢是不可能实现的，有了之后每个人只需要和博彩公司联系，关注赔率和赌金即可

例子 - 泡泡堂游戏

我们通过一个自制的泡泡堂游戏来感受一下：

```
1 /**
```

```
 * 泡泡堂的例子
 */
// 定义玩家
class Player {
    constructor(name){
        this.name = name;
        this.enemy = null
    }

    win() {
        console.log(this.name + 'win');
    }

    lose(){
        console.log(this.name + 'lose');
    }

    die(){
        this.lose();
        this.enemy.win();
    }
}

let player1 = new Player('皮蛋');
let player2 = new Player('小怪');

// 相互为敌人
player1.enemy = player2;
player2.enemy = player1;

// 当player1被泡泡炸死的时候，只需要下面这一句代码完成了游戏
player1.die();
```

为游戏增加队伍

接下来我们改进游戏，考虑增加玩家到8个人，并可以设置队伍，但是还是用原来的方式非常麻烦：

```
player1.enemies = [player5,player6....]
```

这里我们需要定义数组players来保存所有的玩家，并新增一些属性来保存队友，敌人，状态，队伍等信息：

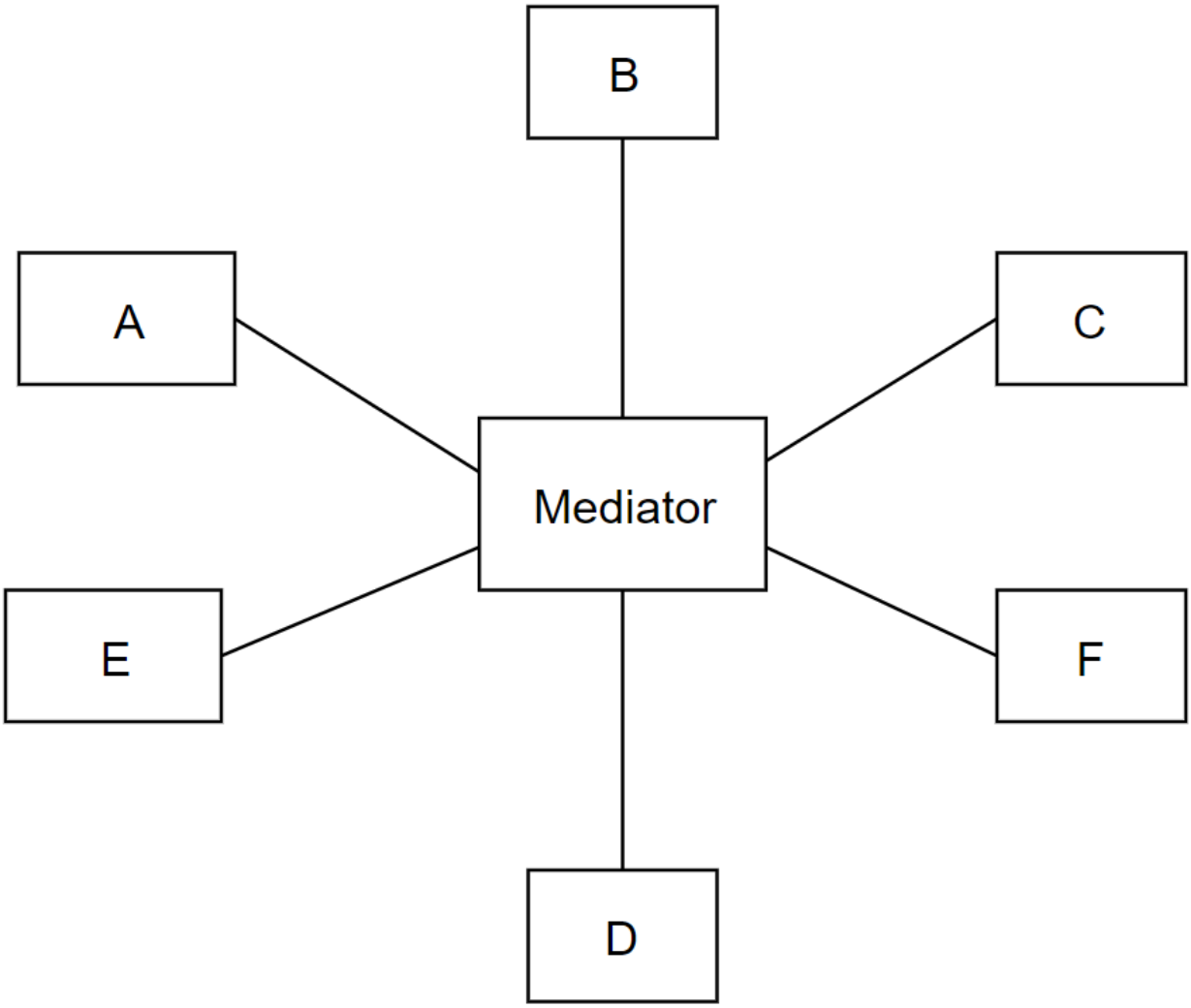
```
/**
 * 泡泡堂的例子
 */
// 保存所有玩家
const players = [];

// 定义玩家
class Player {
    constructor(name, teamColor) {
        this.name = name;
        this.partners = [];
        this.enemies = [];
        this.state = 'live';
        this.teamColor = teamColor;
    }

    win() {
        console.log(this.name + 'win');
    }

    lose() {
        console.log(this.name + 'lose');
    }

    die() {
        // 需要遍历其他队友情况，判断队伍获胜
        let all_dead = true;
        this.state = 'dead';
        for (let partner in this.partners) {
            if (partner.state !== 'dead') {
                all_dead = false;
                break;
            }
        }
        if (all_dead) {
            this.lose();
            for (let partner in this.partners) {
                partner.lose();
            }
            for (let enemy in this.enemies) {
                enemy.win();
            }
        }
    }
}

const playerFactor = function (name, teamColor) {
```

```
    // 定义工厂方法
    const newPlayer = new Player(name, teamColor);
    for (let player in players) {
        if (player.teamColor === newPlayer.teamColor) {
            player.partners.push(newPlayer);
            newPlayer.partners.push(player);
        } else {
            player.enemies.push(newPlayer);
            newPlayer.enemies.push(player);
        }
    }
    players.push(newPlayer);

    return newPlayer;
}

let player1 = playerFactor('玩家1', 'red');
let player2 = playerFactor('玩家2', 'red');
let player3 = playerFactor('玩家3', 'blue');
let player4 = playerFactor('玩家4', 'blue');

player1.die();
player2.die();
```

但是我们发现这里其实每个玩家和别的玩家对象都是耦合在一起的，通过this.partners/enemies保存的引用，当每个对象的状态发生改变的时候必须显式地遍历通知其他对象才行。

如果是在一个大型网游中，那么每个玩家都必须保存其他玩家的所有信息，当队伍改变时候更麻烦。所以上面的代码很难支撑大体量的逻辑

用中介者模式改造



我们需要一个playerDirector来管理玩家之间的交互，而Player对象也不再需要耦合其他的Player，只需要将自身状态变更向playerDirector发送消息即可

```
/**
 * 引入中介模式
 */
// Player无需关注其他Player
class Player {
    constructor(name, teamColor) {
        this.name = name;
        this.state = 'live';
        this.teamColor = teamColor;
    }

    win() {
        console.log(this.name + 'win');
    }

    lose() {
        console.log(this.name + 'lose');
```

```
    }

    die() {
        this.state = 'dead';
        // 通知中介者-玩家死亡
        playerDirector.ReceiveMessage('playerDead', this);
    }

    remove() {
        playerDirector.ReceiveMessage('removePlayer', this);
    }
}

const playerFactor2 = function (name, teamColor) {
    // 这里工厂只需要当个壳子
    const newPlayer = new Player(name, teamColor);
    playerDirector.ReceiveMessage('addPlayer', newPlayer);

    return newPlayer;
}

// 实现中介者 - 两种实现本质上没区别 - demo是第二种
// 1. 通过发布-订阅模式。Director为订阅者，各Player发布者
// 2. 在playerDirctor中开放一些接收消息的接口，player主动向director发送消息，director

const playDirector = (function(){
    // 保存所有玩家
    let players = {};
    // 可以执行的操作
    let operations = {};

    operations.addPlayer = function(player){
        const teamColor = player.teamColor;
        players[teamColor] = players[teamColor] || [];
        players[teamColor].push(player);
    }

    operations.removePlayer = function(player){
        //...删除逻辑
    }

    operations.changeTeam = function(player){
        //...变队逻辑
    }

    operations.playerDead = function(player){
        //...玩家死亡逻辑 - 需要判断是否队伍全部死亡，是的话另一对获胜
```

```
    }

    const ReceiveMessage = function(){
        // 获取第一个参数为消息名称
        const msg = Array.prototype.shift.call(arguments);
        operations[msg].apply(this, arguments);
    }

    return {
        ReceiveMessage
    }
})();
```

这里除了中介者，没有一个玩家需要知道其他玩家的存在，解除了不必要的耦合，某个玩家的操作只需要给中介者发送一个消息，中介者处理完消息之后会把处理结果反馈给其他的玩家对象。同时我们扩展功能只需要给中介者扩展

小结

中介者模式是迎合迪米特法则的实现，即最少知识原则。指一个对象应该尽可能少地了解另外的对象（不和陌生人说话）。如果对象之间耦合太高，那么一个对象的改变难免会影响其他对象。而在中介者模式中，对象不需要知道彼此存在，只通过中介在影响对方

将原来对象之间的多对多网状关系改为了一对多关系

不过注意中介者模式也存在缺点：会新增一个较为庞大的中介者对象，本身会较为复杂和难以维护，相对也比较消耗性能

注意，对象之间不一定非要解耦，项目之中对象或者模块有一些依赖是很正常的，毕竟写程序是为了交付而非堆砌模式和过度设计，所以关键是要衡量对象之间的耦合程序与业务的关系，如果耦合度随着业务迭代程序指数增长，那么我们就由必要使用中介者模式等方法来重构了