软件架构与设计模式
OsCourseProject

1452692 尹屹凡

设计模式

适配器 Adapter

案例：动态内存分配中的内存表管理

做法：原先的设计中，空闲内存和内存使用表都是自行设计编写完成的；实际上，大部分功能都可用标准库中的ArrayList容器实现。因此重新封装了ArrayList容器，用于维护这两张内存表。

作用：这样可以很好的利用标准库中已经十分完善的代码实现自己的功能，结构清晰，不易出错也便于维护。

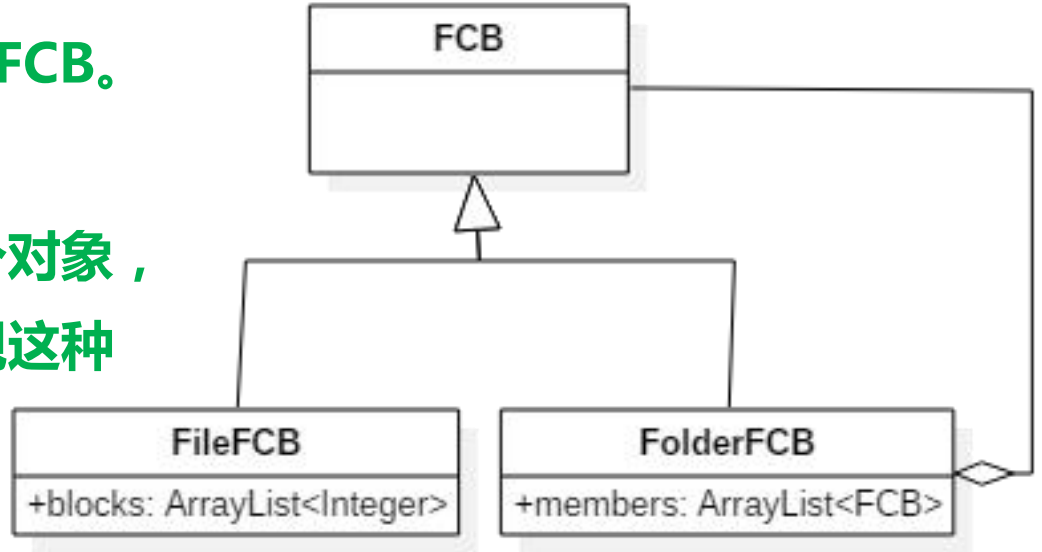
设计模式

组合模式 Composite

案例：文件管理中的文件和目录

做法：设计两个子类FileFCB和FolderFCB，都继承自FCB。其中FolderFCB可以包含FCB类型的子节点。

作用：文件系统是典型的组合模式，其中文件作为单个对象，文件夹作为复合对象，利用组合模式可以很简单地实现这种文件系统结构。



设计模式

工厂方法 Factory Method

案例：文件和目录的创建、页式内存管理的指令等

做法：设计两个子类FileFCB和FolderFCB，都继承自FCB。其中FolderFCB可以包含FCB类型的子节点。

作用：减少写死的代码逻辑，便于拓展代码，在新增子类的时候减少修改的工作量。使代码结构更加清晰，增强可读性和可维护性。

```
//factory method
static FCB createFCB(String n, int t) {
    switch(t) {
    case FILE:
        return new FileFCB(n);
    case FOLDER:
        return new FolderFCB(n);
    default:
        return null;
    }
}
```

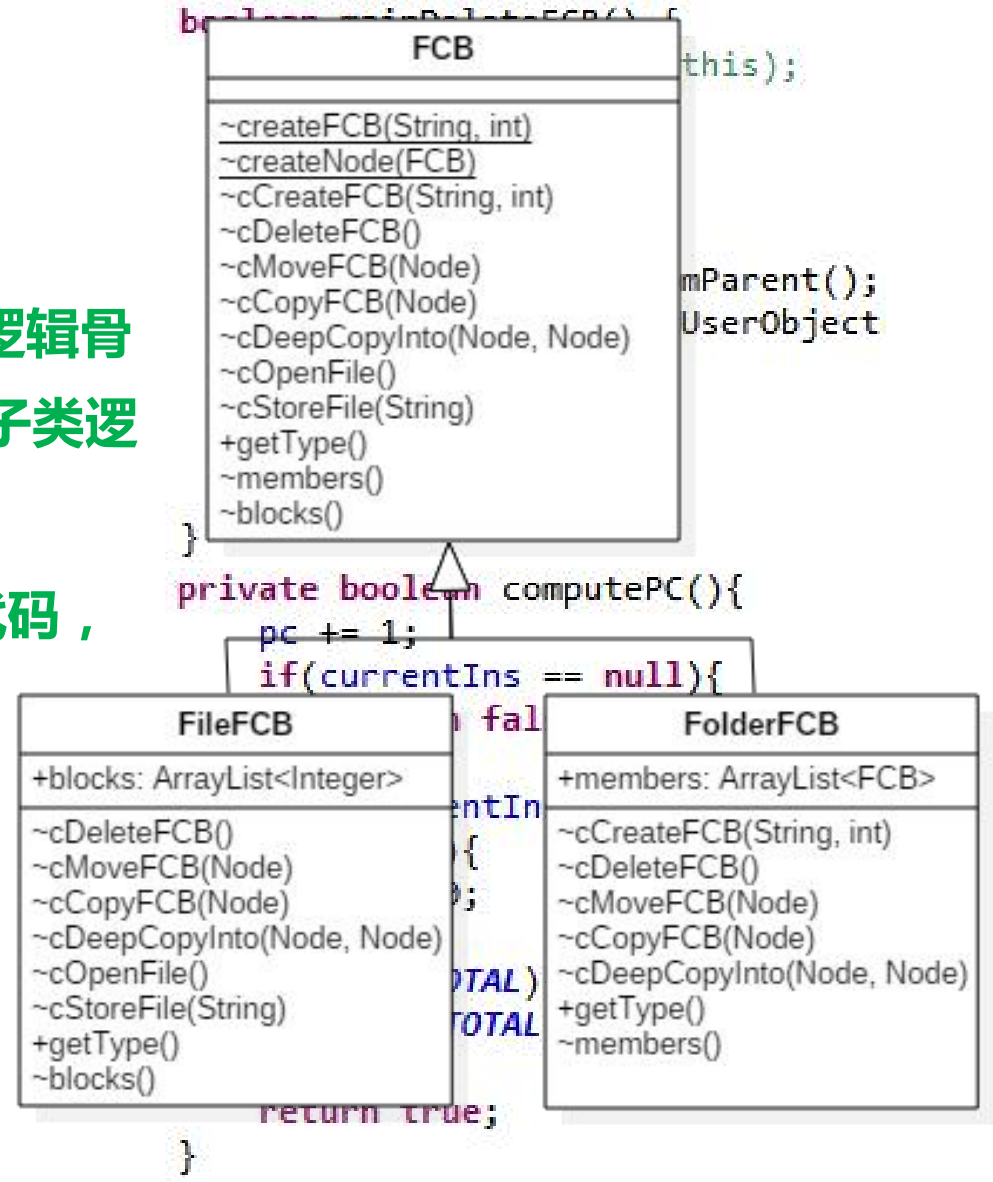
设计模式

模板方法 Template Method

案例：FCB、Instruction、FunctionItem等

做法：在FCB类中的mainDeleteFCB()方法中设计好逻辑骨架，而逻辑细节则调用cDeleteFCB()。后者对于不同子类逻辑不同，由FileFCB和FolderFCB实现。

作用：共用逻辑骨架，减少大量重复代码。便于理解代码，便于维护和拓展。



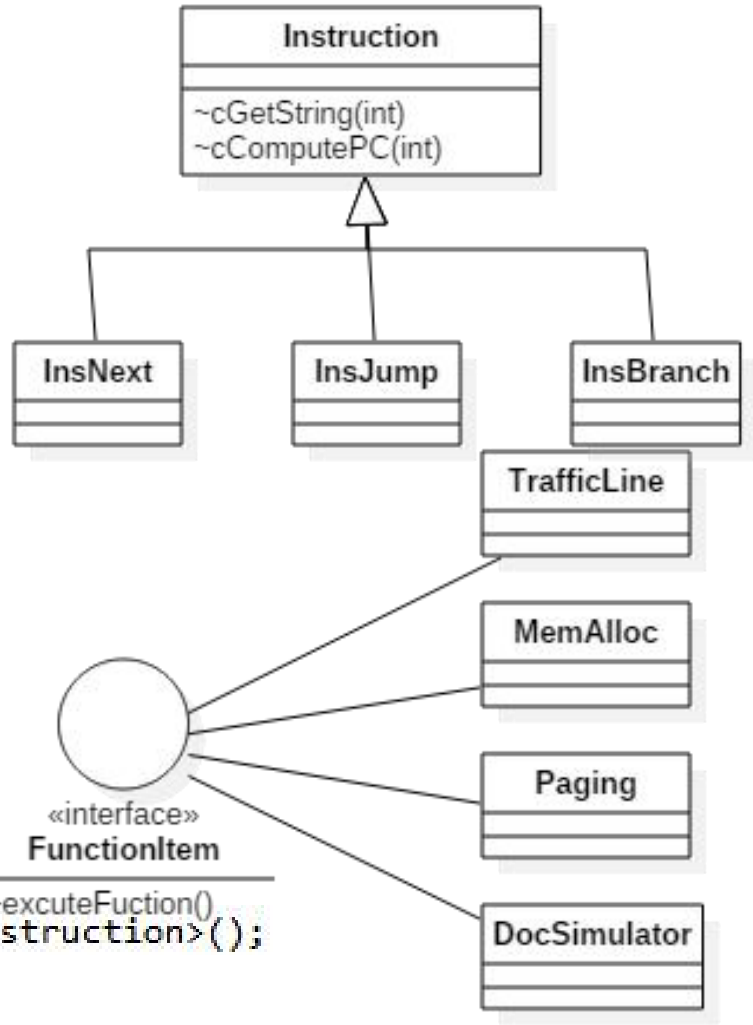
设计模式

原型模式 Prototype

案例：页式内存管理的指令创建、模拟器功能块的创建

做法：Paging类为创建Instruction的子类，对每一种子类建立一个原型对象并保存起来，每次需要相应子类时，只需要调用该子类原型的create函数便可创建一个新的对象并对其初始化。

作用：这种方式可以减少其他类对Instruction子类实现细节的关注，只需要简单地调用create函数即可创建对象。




```
private ArrayList<Instruction> insTemplates = new ArrayList<Instruction>();
private Instruction createIns(int index) {
    if(index > -1 && index < insTemplates.size()) {
        return insTemplates.get(index).create();
    }
    return null;
}
```

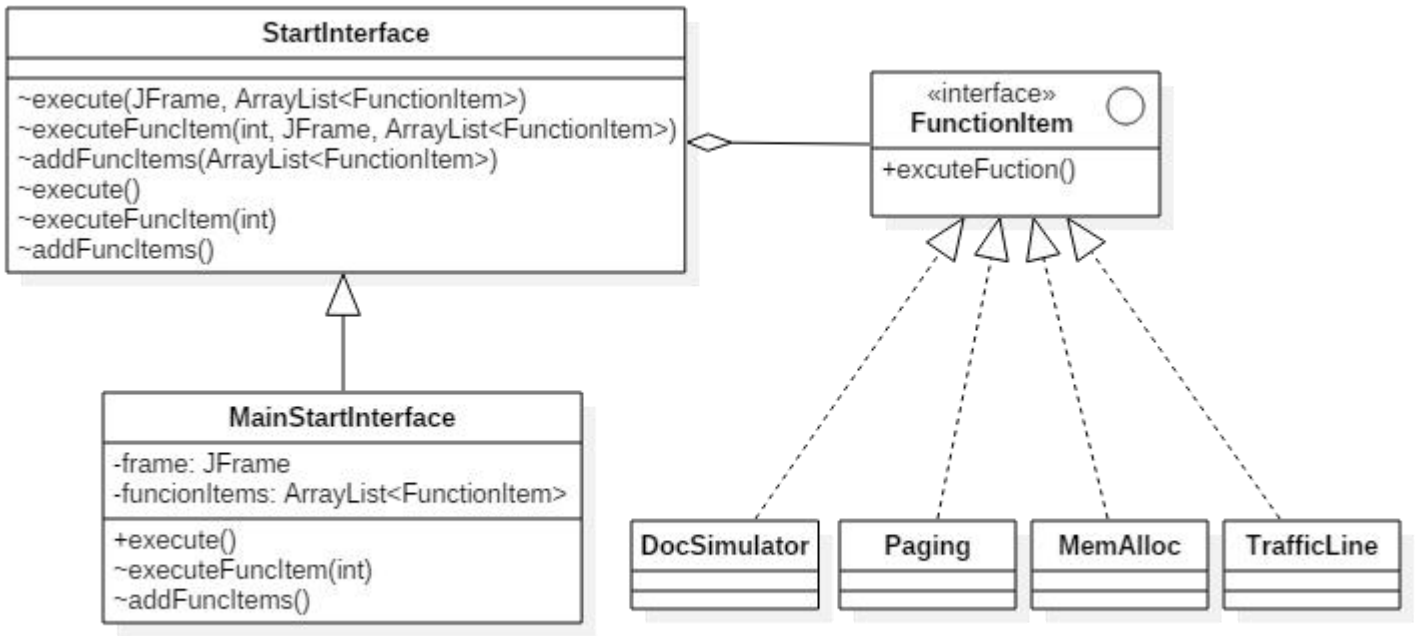
设计模式

抽象工厂 Abstract Factory

案例：模拟器主界面调用各个模拟器

做法：设计一个基本的主界面类StartInterface，自定义的主界面类可以继承该父类，修改创建部分方法和成员变量即可完成主界面的设计。

作用：可以创建设计不同但基本功能一致的主界面，便于拓展和维护。



其他重构

多线程

案例：交通管理系统

作用：重构后，每一个功能可以相对独立地执行，以信号量的方式进行互斥竞争或协调同步。控制逻辑不需要严格写死，改由线程的互斥和同步来控制程序。

```
static public void startRunning()
{
    TrafficLine tl1 = new TrafficLine(0);
    TrafficLine tl2 = new TrafficLine(1);
    drawInit(tl1, tl2);
    Thread thTurn = new Thread(()->{TrafficLine.turnSignal();});
    Thread thSet1 = new Thread(()->{TrafficLine.sRandomSetCar(tl1);});
    Thread thPopC1 = new Thread(()->{TrafficLine.sPopCarOnCross(tl1, tl2);});
    Thread thPopL1 = new Thread(()->{TrafficLine.sPopCarOnLine(tl1);});
    Thread thSet2 = new Thread(()->{TrafficLine.sRandomSetCar(tl2);});
    Thread thPopC2 = new Thread(()->{TrafficLine.sPopCarOnCross(tl2, tl1);});
    Thread thPopL2 = new Thread(()->{TrafficLine.sPopCarOnLine(tl2);});
    Thread thDrawFrame = new Thread(()->{TrafficLine.drawFrame(tl1, tl2);});
    thTurn.start();
    thSet1.start();
    thPopC1.start();
    thPopL1.start();
    thSet2.start();
    thPopC2.start();
    thPopL2.start();
    thDrawFrame.start();
}
```

其他重构

函数定义链

案例：文件系统节点设置

作用：统一相同的函数逻辑，避免不一致。尤其是在修改函数逻辑的时候，只需要修改函数定义链中最底端的函数逻辑，便可应用到整条定义链上。

```
static DefaultMutableTreeNode setNode(DefaultMutableTreeNode root, String p, String n, int t){
    DefaultMutableTreeNode parent = getNodeByPath(root, p);
    return createFCB(parent, n ,t);
}


static DefaultMutableTreeNode createFCB(DefaultMutableTreeNode parent, String n, int t){
    if(parent == null){
        return null;
    }
    return ((FCB)parent.getUserObject()).cCreateFCB(n, t);
}
```

其他重构

λ表达式

案例：整体

作用：利用该方式，可以把重点放在这些代码中函数的设计而非类和对象的设计上；同时使代码更加简洁。

```
buttons[0].addActionListener((ActionEvent e)->{executeFuncItem(0);});
buttons[1].addActionListener((ActionEvent e)->{executeFuncItem(1);});
buttons[2].addActionListener((ActionEvent e)->{executeFuncItem(2);});
buttons[3].addActionListener((ActionEvent e)->{executeFuncItem(3);});
```

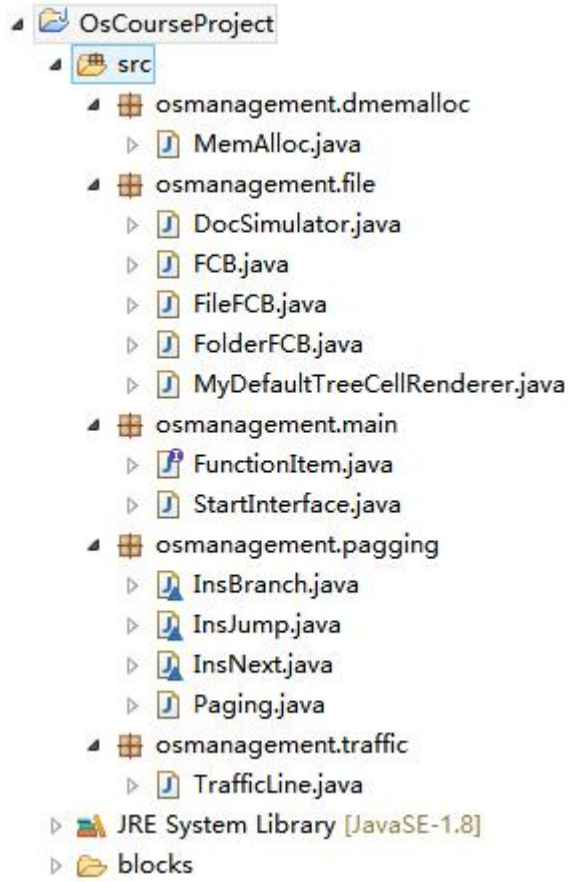
```
Thread thTurn = new Thread(()->{TrafficLine.turnSignal();});
Thread thSet1 = new Thread(()->{TrafficLine.sRandomSetCar(tl1);});
Thread thPopC1 = new Thread(()->{TrafficLine.sPopCarOnCross(tl1, tl2);});
Thread thPopL1 = new Thread(()->{TrafficLine.sPopCarOnLine(tl1);});
Thread thSet2 = new Thread(()->{TrafficLine.sRandomSetCar(tl2);});
Thread thPopC2 = new Thread(()->{TrafficLine.sPopCarOnCross(tl2, tl1);});
Thread thPopL2 = new Thread(()->{TrafficLine.sPopCarOnLine(tl2);});
Thread thDrawFrame = new Thread(()->{TrafficLine.drawFrame(tl1, tl2);});
```

其他重构

包和类划分

案例：整体

作用：使代码结构更加清晰，增强代码可读性。



项目管理

https://github.com/CNyyf/OsCourseProject/commits/master