面向对象第三次报告 – 高级设计意图

杨泽超 2019K8009929040

这次完成的内容主要是前两次的代码实现，在实现的过程中还是发现了一些考虑不周的问题。首先整体介绍一下最终确定的设计。

抽象方面，主要的一个设计是参考了2D游戏设计理念中的“精灵”或“游戏实体”概念。它能感受游戏运行中时间的流逝，也即可以随游戏进展而发生实时的变化。这样一个抽象可以良好封装其内部图像绘制、碰撞检测等逻辑，供外部的调度器统一调度、查询、重绘。例如产生动作时可以直接调用统一的接口：

ArrayList<Entity.ActionMessage> eMsg;

**for** (Entity entity : GameResources.*entityList*) {

eMsg = entity.generateAction();

**if** (eMsg != **null**)

msgList.addAll(eMsg);

}

或者交互时、绘制时：

msgList.forEach(m -> interact(m));

GameResources.*spiriteList*.forEach(e -> e.display(g));

事实上，NPC、玩家、普通方块之间以及各个NPC之间都会有不同的代码逻辑，例如NPC需要根据阵营绘制名称颜色，友方是绿色而敌方是红色；普通方块不能交互；碰撞检测方块就是长方形，而NPC和玩家根据行为不同可能会有圆形范围等。添加动画也变的简单，只需要每次行动的时候更改计时器，在绘制时绘制不同的图片即可，而这些游戏的调度器都不需要知道，调度器应当只关心游戏处于什么界面什么状态，将要跳转到什么位置，以及在这个状态下我应该做什么。同样的，各个游戏实体也无需知道现在究竟是什么状态——调度器让干什么就干什么就好了。

这样在实体设计上就完成了高内聚低耦合，调度器也只需要面向他们的父类“精灵”（为了节省时间开销又加了一层“实体”设计，来减少枚举可交互物体的复杂度）编程，依赖于抽象。这样在某一类物体需要修改逻辑时只需要添加/修改子类代码即可，只是添加物体、删除物体、作弊修改的话只需要改改文件数据即可。同时，对于共有的代码例如广义的碰撞检测（即不考虑细节粗略判断）、没有细节的显示等代码可以直接复用。

如此将实体与实体之间解耦合后，需要继续处理他们之间的依赖，而他们之间的依赖非常的广泛。玩家产生的动作可能作用到很多其他实体上，所有实体的状态由决定了NPC的AI来产生动作。这样就可以采取简单的观察者模式来处理这种情况，即之前所提到的调度器。

在游戏中，状态的改变和观察者是简单的。状态只依赖于外部时间或者外部玩家的输入，在一秒30帧的情况下，用户的输入仍然是相对稀疏的、并且只对（目前）唯一的Player类生效，所以可以附在时间变化上，这样状态变化简单；观察者也是简单的，游戏窗口是唯一的，时间流动是统一的，所以只需要一个观察者统筹所有目标即可。在这种情况下可以简化实现为外部事件（时间流动）触发调度器运作，调度器告诉所有目标时间、输入信息，目标以消息（具体目标）的队列传递协作通知。再由调度器委派到相关目标上（这里具体上是通过实体提供的crashDetect方法）。核心代码也如上所示，其中interact是为每个目标处理目标“兴趣”的Dispatcher内部的方法。

如上提到了窗体和调度器在游戏中是唯一的，那么可以使用单例模式。窗体关了程序就结束了，不过内部游戏结束后重启、切换游戏模式时可能需要单例的调度器，只需要重新添加键盘监听器。

**private** **static** Dispatcher *dispatcher* = **null**;

**private** Dispatcher(GameBoard board) {

//init all resources

……

}

**public** **static** **synchronized** Dispatcher getDispatcher(GameBoard board) {

**if** (*dispatcher* == **null**)

*dispatcher* = **new** Dispatcher(board);

**else** board.addKeyListener(*dispatcher*.**new** KeyMoniter());

**return** *dispatcher*;

}

\*顺便，在调试加载图片的一个bug时候进入Toolkit代码也意外发现了单例模式的应用。

最后，最终的类图在同目录下的图片中。（部分方法与代码可能存在一对多或者功能类似但名称有出入）

依赖上，核心逻辑是调度器，其主要依赖于抽象的精灵类或实体类（以及其他如菜单-条目类），而所依赖的类只关心自己的问题不依赖于其他。

另外，游戏还有一些其他多样性设计，例如多样的技能、菜单等，考虑到需求和设计的频繁变更，采用了load方法按照规则将文件中的数据加载后解释性呈现，并且维护菜单表实现自定义的菜单跳转，来进一步加强扩展性和修改的封闭性。不过由于其他课作业也特别多写完框架和逻辑已经没有时间来设计关卡、图片、物品数据等等了，所以在最后的代码里只有简单逻辑的展示orz

总的来说，在实际编写代码的时候还是发现了大量初次设计中没有考虑到的地方，结构也调整过很多次（第一次写确实比我想象中的要困难的多），完全不是了解了概念、设计方法就可以很好完成的。毕竟最终的目标是降低系统的复杂性，用统一一套方法解决一个问题，在考虑、修改一个问题时尽量不要牵扯到其他的问题，同时能少写新代码就尽量少写新代码，所以对我这样的来说最终还是要落到实际需求、可能边界的处理以及代码上。

整个课程总结下来还是学到了很多内容，亲眼也亲手知道了许多设计以及思想，在写操作系统的C代码的时候想着要是C也能直接用这些特性多好，方便多了~同时也理解了同期选的软件工程里面MVC框架和ruby更加彻底的一切皆对象。另外，选了设计实现恰恰又发现经常容易自己把自己绕坑里也恨不熟练，感觉先去看看别人的代码怎么写的或许更好呢？