# 设计模式课程设计说明文档

## 1：课程设计要求：

将在课堂上学到的三个设计模式，包括Composite，Observe，UndoRedo（即Command），以C++的形式给出，通过代码实践这三种设计模式，加深自己对这三个乃至其他经典的设计模式的理解，为以后再真正的大项目中运用这些设计模式打下基础，能够运用它们写出灵活，可复用的软件，提高代码的健壮性。

## 2：课程设计目标：

用C++语言分别实现三个设计模式是毋庸置疑的，其实实现它们并不困难，关键是能够在深刻理解它们，在自己的项目中能够运用它们。

所以，自己的课程设计目标就是程序中不仅限于这三个孤立的设计模式，而是穿插其他的设计模式，当然了，不是为了运用设计模式而用设计模式，而是以合适的方式在合适的地方使用合适的设计模式，并作出一定的解释。

同时，还有一个问题是怎么判断这三个设计模式是正确实现了的而没有错误呢？这肯定是需要一个个完整程序来进行说明，所以自己的课程设计项目中就是三个可检测的程序。

总之，自己的课程设计的最终目的就是分别以三个设计模式为基础，向周围扩散，应用其他的设计模式，实现三个简单的，具有一定可视化的界面的程序。

## 3：课程设计

### A：Composite设计模式实现

1：Composite设计模式说明

将对象组合成树形结构以表示“部分-整体”的层次结构，该设计模式使得用户对单个对象和组合对象的使用具有一致性。Composite模式描述了如何使用递归组合，使得用户可以不必对这些类进行区分。

其适用于：当你想表示部分-整体层次结构；当你希望用户忽略组合对象和单个对象的不同，方便用户统一地使用组合结构中的对象。

其定义了包含基本对象和组合对象的类层次结构；简化了客户代码；使得更容易地增加新类型的组件，使设计更加一般化。

2：程序实现及说明

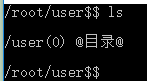
考虑到目录和文件的关系最符合Composite所定义的类的体系结构，所以考虑在此基础在实现了一个简单的基于命令行的模拟文件系统（当然了，只是模拟的，并没有真正的存储文件）。同时在此基础上，采用了其他的设计模式，最终实现的是可以撤销以及重做的模拟文件系统

3：程序截图演示以及说明

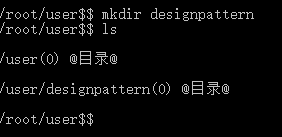
为了方便展示，一开始程序中就添加了一些文件或目录，如下图所示



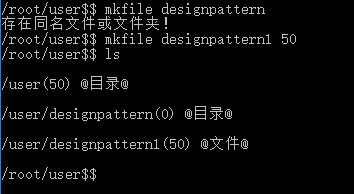
ls:列出当前目录下的（包括本目录）所有的文件和目录



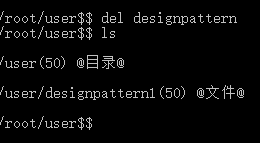
mkdir：在当前目录下创建目录



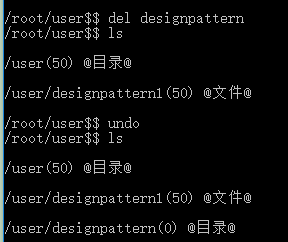
mkfile:在当前目录下创建文件



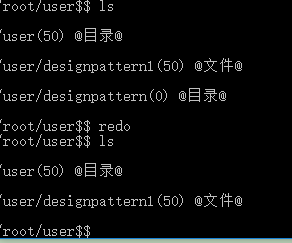
del:删除文件或目录:



undo:撤销上一次的操作



redo:重做上一次的操作



cd: 从当前目录跳转到某个目录

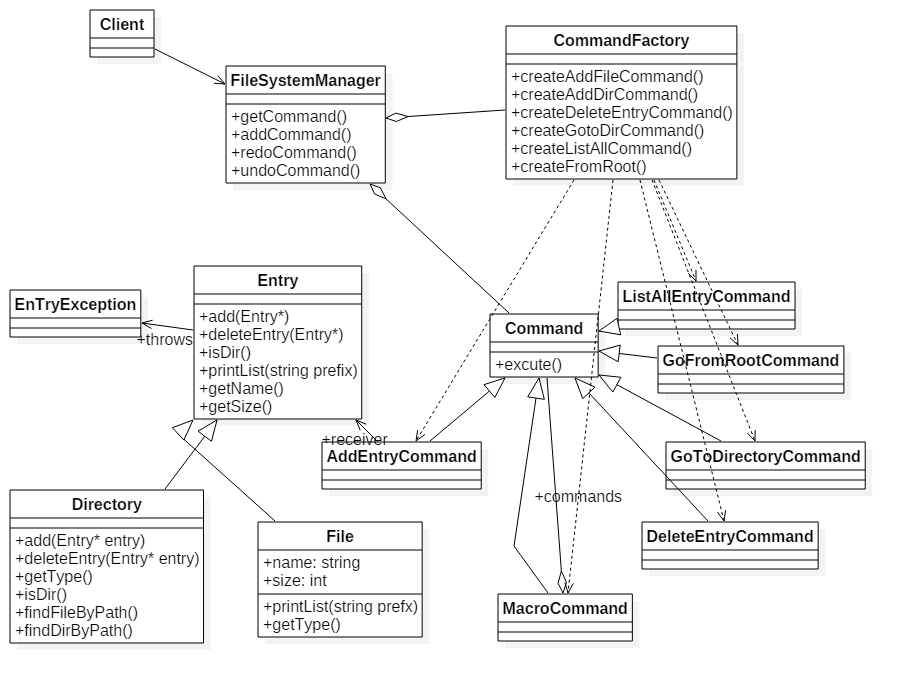


home:从主目录跳转到某个目录



4:程序UML图

类图

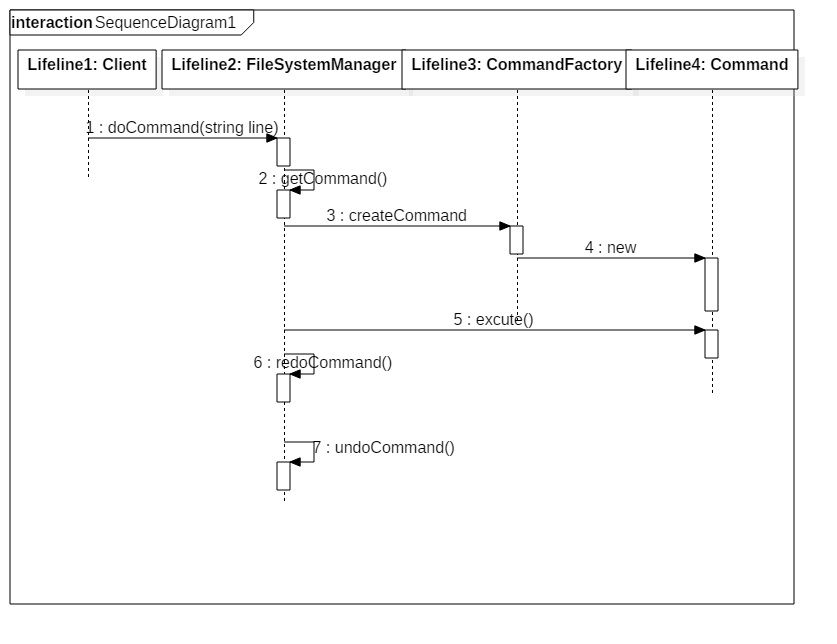


这里首先定义了一个Entry基类，FIle,Directory句继承自该基类，Entry，File在执行add(Entry\* e)将抛出异常

同时，这里定义了一个命令基类，Command，将对于文件的操作封装成为一个个的Command，方便调用，以支持Undo，Redo操作。可以看到，每一个对文件系统的操作均对于一个派生Command，整个设计的体系结构也更加清晰。考虑到命令的繁多以及相似性，单独创建了一个CommandFactory类来负责创建一个个的Command。

并且，对下层的整个系统抽离出一个门面，FileSystemManager类来处理下层负责的逻辑，Client只需要与FileSystemManager进行交换，根本不需要了解底层的逻辑。

时序图：



整个程序的逻辑如图所示：

客户端输入命令，FileSystemManager类负责处理命令，解析命令，生成对应的Command，然后执行就可以了。

5：涉及的设计模式

1：Composite设计模式：

2：Abstract Factory 设计模式 ：通过该设计模式，分离了具体的类，客户端不需要了解Command类的具体细节，不需要了解Command类的继承体系，只需要通过一个Factory就可以获得Command。

3：Command设计模式：将对文件，目录的操作封装成了一个个具体的命令，能够支持UndoRedo操作，将调用操作的对象与知道如何实现该操作的对象解耦。同时，增加新的Command很容易，因为无需改变已有的类。

4：Facade(外观）模式，定义了一个FileSystemManager，为底层子系统的一组接口提供了一个一致的界面，客户端只需要调用该门面类的接口即可。通过该外观模式，对客户端屏蔽了底层文件系统的逻辑。减少了客户端处理的对象的数目。实现了子系统与客户之间的松耦合关系。

### B：Observer设计模式实现

1：observer设计模式说明：

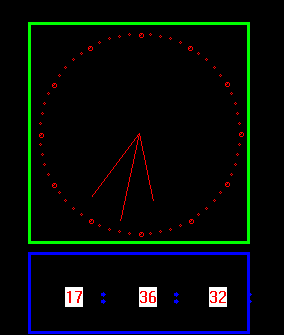
定义了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并被自动更新。Observer设计模式适用于：当一个抽象某些有两个方面，其中一个方面依赖于另一方面；当一个对象的改变需要同时改变其他对象；当一个对象必须通知其他对象，而又不能假定其他对象时谁。

Observer设计模式使得：目标和观察者间的抽象耦合，支持广播通信，以及意外的更新。

2：程序实现以及说明

在这个程序里，有一个被观察者（或者说消息发送者），名字是ClockTimer，用于存储和维护时间的流逝，它每秒通知一次它的观察者。同时，这里有两个观察者，一个是数字时钟，一个是普通的机械时钟，将接收到的时间显示出来

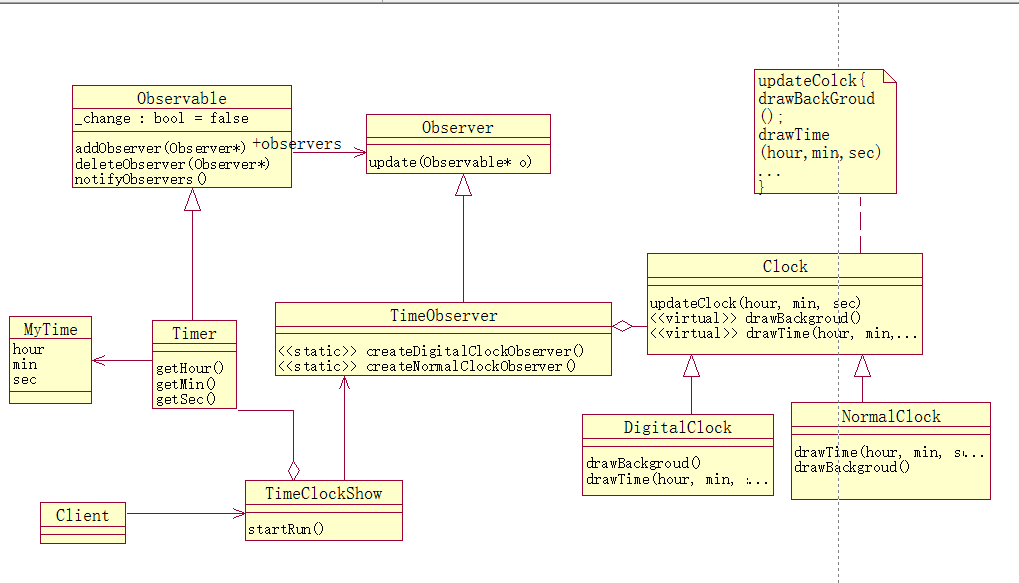
3：程序截图以及演示



如图可以看出分别有两种时钟来显示时间，

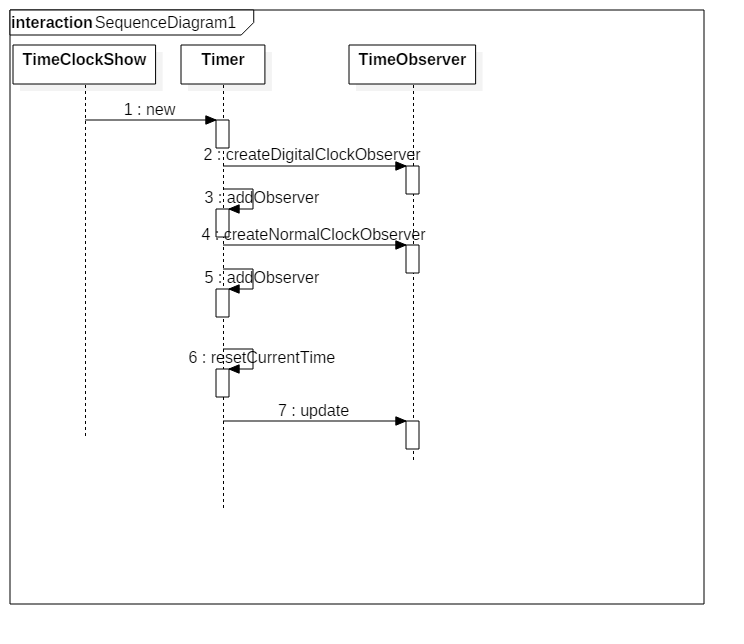
4：程序UML图

类图：



如图所示，定义了一个Observable类，被观察者继承自该类，在这个程序中，Timer 继承自该类用以向监听者发布时间信息，Observable维护一组Observer，每当对应的事件发生时，便通知其监听者，在这里定义了一个时间监听者TimerObserver，其监听到时间的变化，用不同的显示时间的算法（数字时钟对应一种算法，机械时钟对应一种算法）来对当前时间进行显示。

时序图：



如图所示，先创建一个Timer（发布事件），然后创建两个Clock（观察者），分别添加到Timer的观察者容器中，每秒重置当前时间，然后通知每个观察者更新时间。

5：涉及的设计模式

1：观察者模式

2：Strategy(策略）模式，在这里，尽管有两个观察者，以不同的方式相应时间变化事件，但我这里只是定义了一个观察者类，不同是我定义了两种算法，不同的观察者以不同的算法来响应时间变化事件。这样的话就不必定义不同的观察者类了，只需要定义不同的算法即可，同时也能实现算法的互相替换。两个不同的时间观察者仅仅是行为上有所区别，并没有其他本质的区别，“策略”就提供了一种用多个行为中的一个行为来配置一个类的方法。

策略模式实际上也算是一种替代继承的方法。通过这种方式，就可以避免产生一大堆相关的类，而这些类之间的唯一差别仅仅是使用的算法或行为。将算法封装在独立的Strategy类中就可以使它更易于切换，易于理解，易于扩展。

3：template（模板）模式，在Clock类中，有一个函数是UpdateClock()函数，这个函数实际上就定义了一套模板，先是drawBackGround()，然后设置时间，最后就是绘制时间drawTime（），这里drawBackGround()和drawTime（）都是纯虚函数，子类负责实现这些函数。

4：Facade(外观）模式，定义了一个TimeClockShow类，向Client屏蔽了时间显示的有关细节，Client端只需要调用TimeClockShow提供的接口即可，不用深入到交错繁杂的底层当中去。

### C：UndoRedo（command模式）设计模式实现

1：UndoRedo（command）设计模式说明

UndoRedo实际上是通过Command模式来实现的，Command设计模式将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销操作，这里重点关注的是通过Command设计模式，实现可撤销操作。

Command模式适用于：

1：抽象出待执行的动作以参数化某对象，可用过程语言中的回调函数表达这种参数机制；

2：:在不同的时刻指定、排列和执行请求。

3：支持取消操作。Command的Excute操作可在实施操作前将该状态存储起来，在取消操作时做过状态用来消除该操作的影响。Command接口要添加一个Unexcute操作，该操作取消上一次Excute调用的效果。

4：支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。

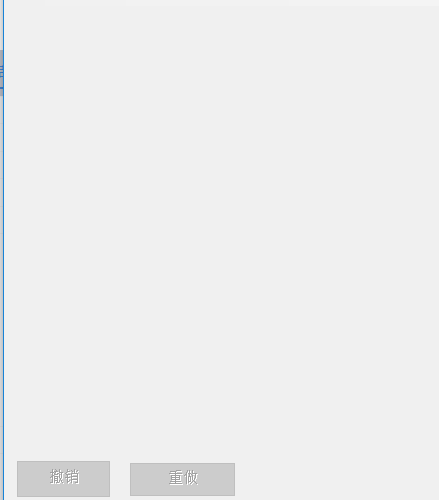
5：用构建在原语操作上的高层操作构造一个系统。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。

2：程序实现以及说明

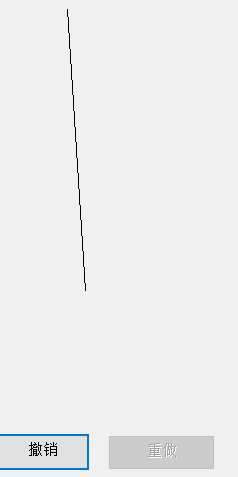
由于需要实现的是UndoRedo操作，而最适合展示UndoRedo操作的程序就属于画图或者文本编辑程序了，所以自己的以C++语言实现UndoRedo的程序就是一个画图的小程序，不过暂时只能画线段，不过这里只是为了验证实现的正确性，并没有做得太复杂。在UndoRedo的逻辑实现这一方面上，用的是纯C++语言（非托管的），而在界面显示方面用的是.net环境下的C++（托管了的），这里特此说明

3：程序截图以及演示

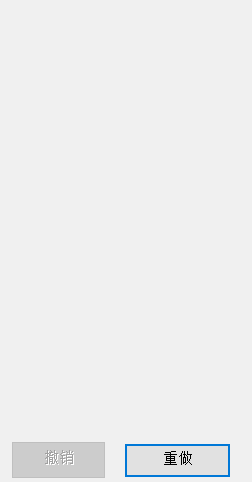
一开始，什么操作都没有做，所以既不能undo，也不能redo，如图



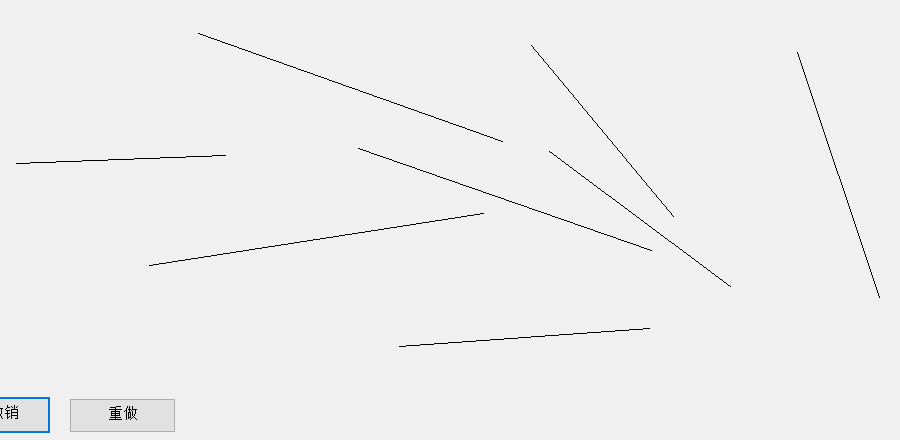
画一条线：



撤销之后，不能再撤销了，但可以重做，如图

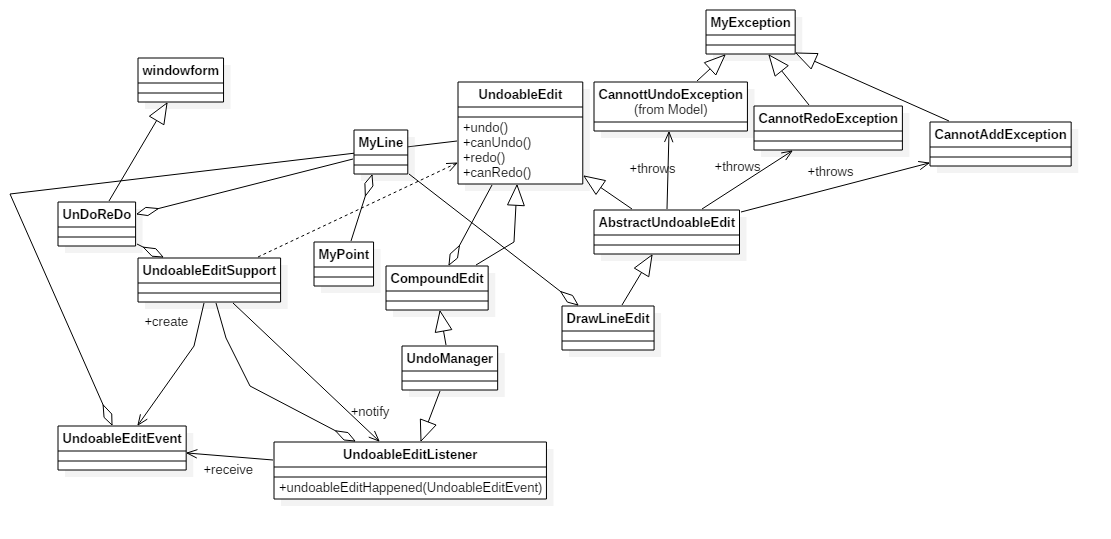


经过一系列操作后



4：程序UML图

类图：



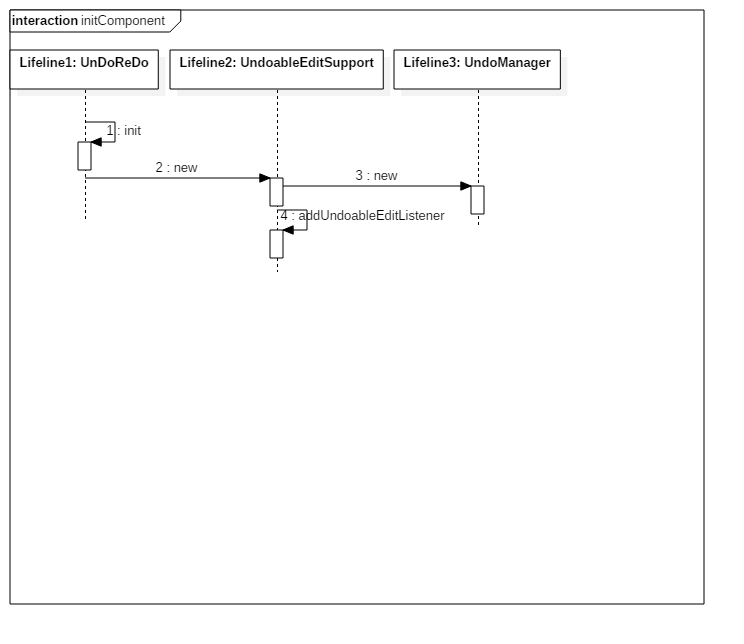
如图所示：

这里定义了一个基类UndoableEdit,它的undo，redo，canundo，canredo，都没有实现，AbstractUndoableEdit实现了这些函数，但对于undo，redo函数，该类只是实现了基本的标识位的置换而已。具体的操作要由具体的Edit来定义。同时定义了一个CompoundEdit类继承自UndoableEdit，同时也包含UndoableEdit，表示这是一个组合的编辑操作。即多个编辑操作组合为一个。UndoMananger类继承自CompoundEdit，实际上这才是真正管理UndoRedo操作的容器。同时，它也继承UndableEditListener，监听UndableEdit编辑操作的发生。

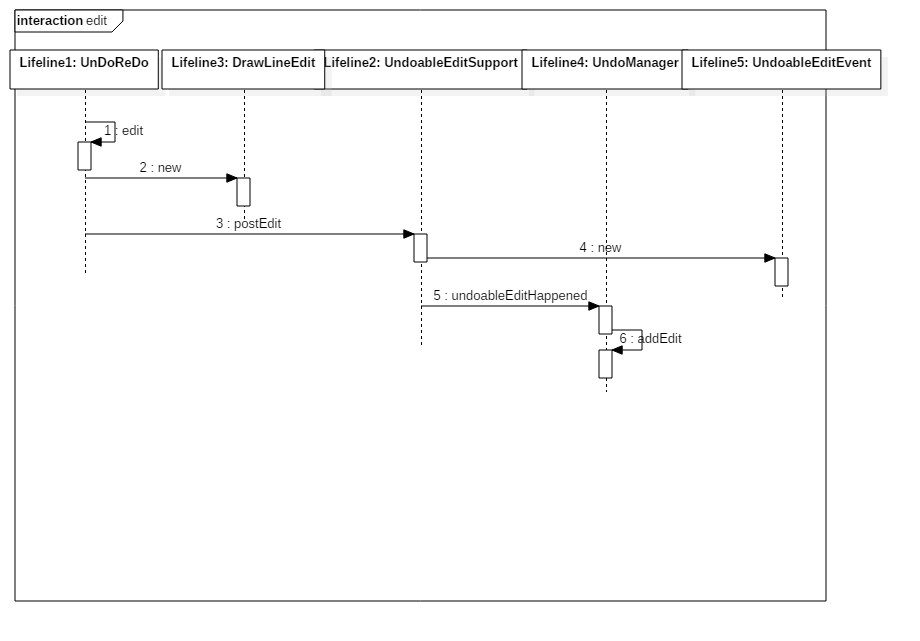
具体的界面类继承自.net的windowsform，其包含一个UndoableEditSupport,以支持UndoRedo操作，当发生一个UndoRedoEdit操作时，UndoableEditSupport,会将该操作包装成一个事件发送给它的监听者(这里实际上只是UndoMananger这个类)。

自定义的UndoRedoEdit编辑操作为DrawLineEdit，即画线操作，为方便画线，定义了两个类,MyPoint,MyLine，在它的Undo，redo函数中定义了具体的操作。

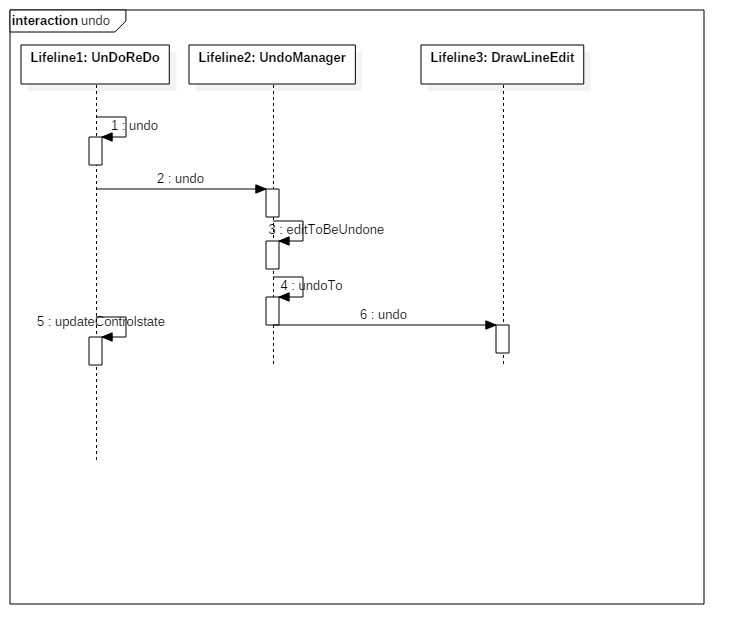
程序时序图：



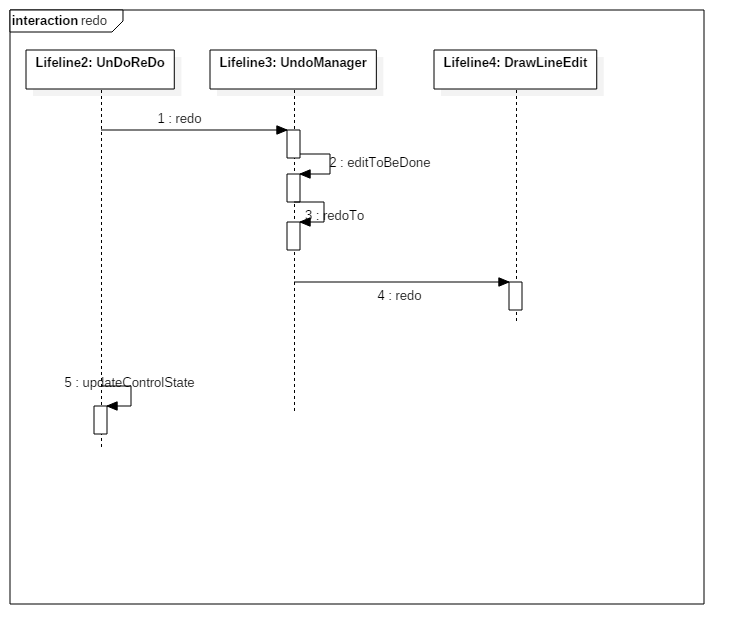
这里就是界面初始化相应组件，新建一个UndoableEditSupport，UndoMananger，将UndoMananger加入到UndoableEditSuppor观察者列表中。



当发生编辑操作时，新建一个DrawLineEdit，然后UndoaleEditSupport将其包装成一个UndoableEditEvent，发送给它的监听者，实际上就是UndoManager，然后UndoManager再将该操作记录起来



当触发undo操作时，undoableManager调用其undo函数，该函数先找到第一个可以被undo的edit，然后从当前指针指向的edit开始，一直到第一个可以被undo的edit，均执行undo操作，然后更新其状态（即置undo，redo按钮的enable属性为true，false）即可。



当触发redo操作时，undoableManager调用其redo函数，该函数先找到第一个可以被redo的edit，然后从当前指针指向的edit开始，一直到第一个可以被redo的edit，均执行redo操作，然后更新其状态（即置undo，redo按钮的enable属性为true，false）即可。

5：涉及的设计模式

1：Command命令模式

2：observer 模式：UndoaleEditSupport将Edit编辑事件发生给其所有的监听者，

鼠标点击事件的监听也属于观察者模式