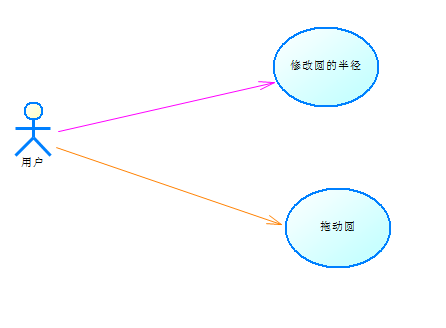
## 需求分析

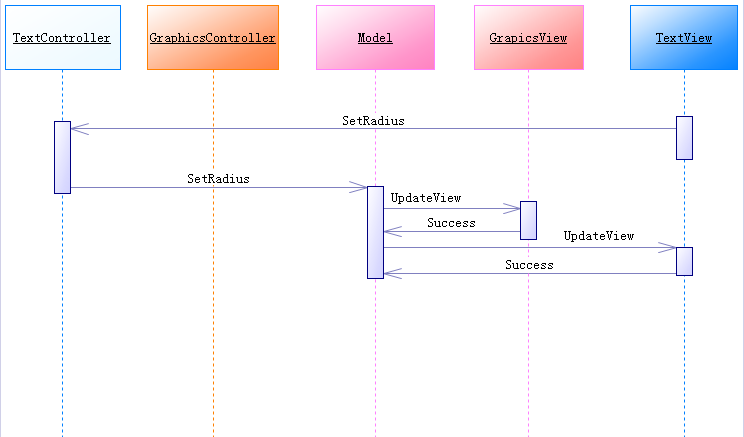
### 用例图

下图表示的是该应用的用例图，主要的用例如下图所示，第一个是要求能够修改圆的半径，第二个用例是要求能够在视图上直接拖动圆来改变圆的半径。

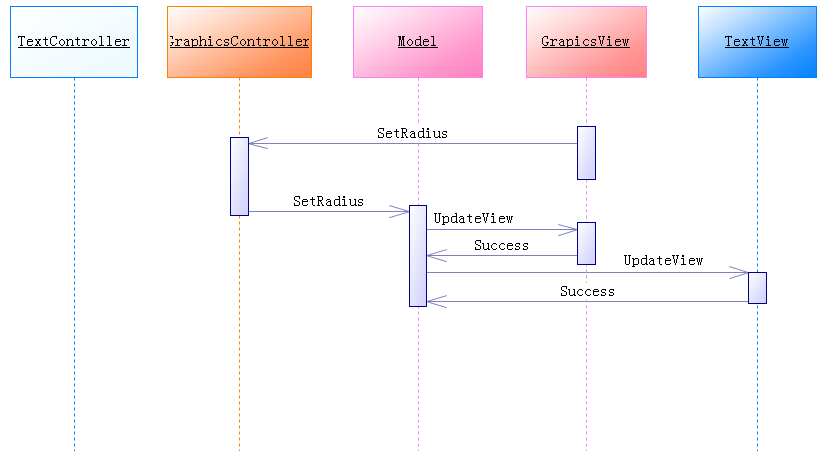


### 顺序图

下面的图表示的是用户改变了TextView里的值之后产生的顺序图：

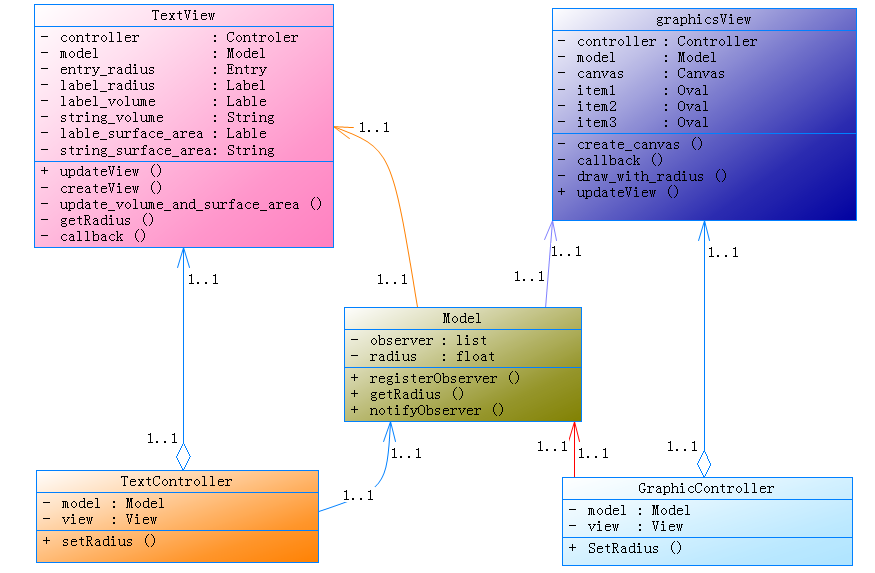


TextView调用TextController的SetRadius方法，然后TextView调用Model的SetRadius方法改变Model的radius，然后Model调用视图的UpdateView方法更新视图。与上面类似的，我们可以得到下面这张图，下面的图是用户改变了GraphicView之后产生的顺序图。



### 类图

根据上面的分析，我们可以很轻松的得到下面这张类图：



如图所示，这个应用一共分为五个类，第一个类是TextView类，这个类主要用于显示应用左半边的输入框，球的表面积，体积等信息，有一个公开的方法，叫做updateView(),第二个view类是graphicsView，这个类和第一个类很相似，只不过表现的是另外的一个东西，用于画球。同样也只有一个公开的方法updateView()。

接下来的controller类，第一个是TextController类，这个类主要是用于调节Model和TextView的关系，减小Model和View之间的耦合度，它只有一个公开的方法setRadius，GraphicsController也很类似。

最后的一个类是Model类，这个类用于记录应用的一些信息，在这个应用里面，有价值的东西只有radius啦，这个类有registerObserver()方法，这个方法主要给别的类注册成为Model的观察者，notifyObserver()这个方法在Model发生变化的时候调用，主要用于通知观察者Model发生了变化。

### 流程

下面我要讲述一下代码的工作流程。

在主函数里面，先调用：

|  |
| --- |
| s\_model **=** Model() **# 创建一个Model** |

然后创建一个TextController，将s\_model传递过去：

|  |
| --- |
| s\_text\_control **=** TextController(s\_model) |

在TextController的构造函数里面发生了什么呢？

|  |
| --- |
| **class TextController:  def \_\_init\_\_**(self, model)**:** self.\_\_model **=** model  self.\_\_text\_view **=** TextView(self, model) **# 初始化的时候创建view** self.\_\_text\_view.updateView() |

我们可以看到，在构造函数里面，TextController将model记录下来，然后创建了一个TextView，TextView里面也很自然的记录了model和controller的引用。

然后TextController调用TextView的函数来更新View，也就是视图。

|  |
| --- |
| **class TextView:  def \_\_init\_\_**(self, controller, model)**:** self.\_\_controller **=** controller  self.\_\_model **=** model  self.\_\_model.registerObserver(self) **# 将这个view注册成为观察者** self.createView() |

上面是TextView的构造函数，在TextView里面记录下了model和controller，然后这个TextView注册成为了model的观察者，然后构建了一个视图。

对于GraphicsController，代码也非常类似：

|  |
| --- |
| **class GraphicsController:  def \_\_init\_\_**(self, model)**: # controller的话，只需要model就行了** self.\_\_model **=** model  self.\_\_graphics\_view **=** GraphicsView(self, model)  self.\_\_graphics\_view.updateView() **# 初始化圆** |

**GraphicsController记录下model，然后构建一个GraphicsView，这个View记录下了对应的model和Controller，然后更新view。**

|  |
| --- |
| **class GraphicsView:**  **def \_\_init\_\_**(self, controller, model)**:** self.\_\_controller **=** controller  self.\_\_model **=** model  self.\_\_model.registerObserver(self) **# 注册成为观察者** self.create\_canvas()  self.create\_ovals() |

代码和前面类似，GraphicsView也要注册成为model的观察者。

值得注意的是，像向输入框输入，拖动椭圆的事件，我们已经注册好了对应的函数，当相应的事件发生的时候，会调用对应的函数。

如在GraphicsView里面，我们注册了这样的函数：

|  |
| --- |
| self.\_\_canvas.bind("<B1-Motion>",  functools.partial(GraphicsView.callback, view**=**self)) **# 注册回调函数** |

注册了feedback函数：

|  |
| --- |
| **@**staticmethod **def callback**(event, view)**:  """回调函数,当鼠标在画布上移动的时候会调用这个函数,我们调用controller的setRadius方法"""** x **=** event.x  y **=** event.y  **# 下面是用于计算出半径** radius **=** round(math.sqrt((x **-** GraphicsView.center[0]) **\*** (x **-** GraphicsView.center[0]) **+** (y **-** GraphicsView.center[1]) **\*** (y **-** GraphicsView.center[1])), 2)  **# 也就是说，现在半径发生了改变,我们应该怎么玩呢？** view.\_\_controller.setRadius(radius) |

代码里面很明显，那就是获取对应的坐标，求出相应的半径，然后调用对应的controller的setRadius函数：

|  |
| --- |
| **class GraphicsController:**  **def setRadius**(self, radius)**:** self.\_\_model.setRadius(radius) |

然后GraphicsController调用model的setRadius函数：

|  |
| --- |
| **class Model:**  **def setRadius**(self, radius)**:** self.\_\_radius **=** radius  self.notifyObserver() |

model改变了radius的大小之后，立刻通知它的观察者们：

|  |
| --- |
| **def notifyObserver**(self)**:  for** eachObserver **in** self.\_\_observer**:** eachObserver.updateView() **# 我们要通知每一个观察者** |

GraphicsView接收到了通知，调用updateView:

|  |
| --- |
| **def updateView**(self)**:  # 获取半径和圆心的消息** radius **=** self.\_\_model.getRadius()  **# 重新绘制图形** self.draw\_with\_radius(radius) |

重新绘制，达到了更新的效果。

更新半径的代码非常类似，这里就不再叙述了。

## 测试

根据需求，我们对做出来的软件进行测试。鼠标拖动画中的圆，左边部分的半径以及表面积，体积等信息发生了相应的变化，同样的，在左边输入半径的信息，右面的圆也会发生相应的变化，这达到了我们的需求。

