MINICAD

目录

实现功能与操作说明

界面介绍

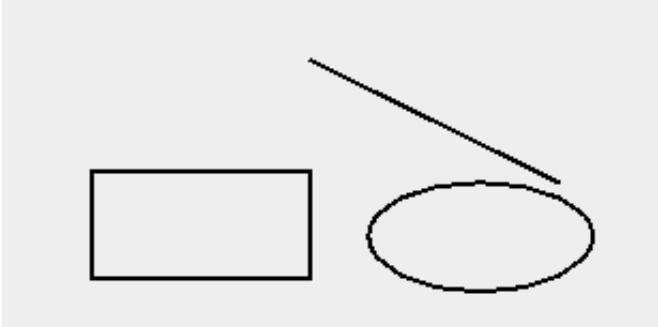
如图，界面共有三块：

- 左侧为模式栏
- 中间为绘图区
- 右侧为操作栏

基本图形绘制

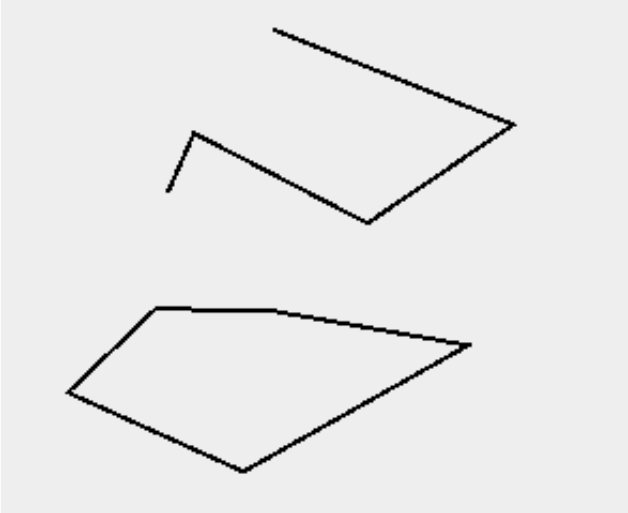
直线，矩形，椭圆

选择模式栏的 `line/rect/oval` 项，拖动鼠标绘制直线/矩形/椭圆。放开鼠标结束绘制：



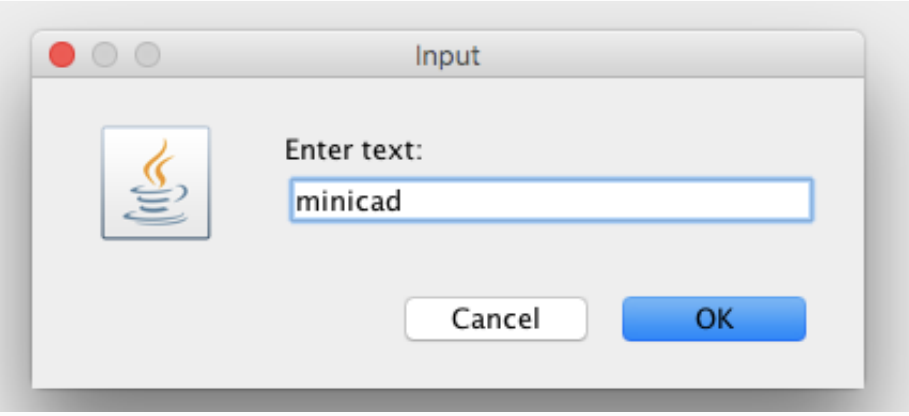
多边形和折线

选择模式栏的`polygon/polyline`。每次单击屏幕选择多边形的一个点。全部点选完后，点击鼠标右键结束绘制：



文本

选择模式栏的`text`，会弹出输入窗口，输入文字：



之后，在绘图区拖动鼠标绘制：

minicad

选中功能

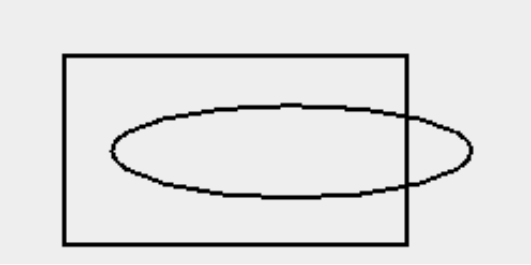
点击模式栏的`select`按钮，进入选择模式。在这个状态下，点击任意图形可以将其选中。判定规则为：

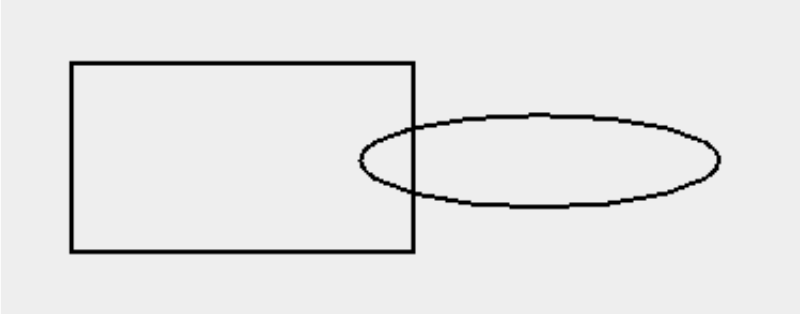
- 矩形，椭圆，文本：图形所在方框内
- 直线，折线，多边形：点击任何一条边，选中该图形。

以下操作必须在选择模式下才有效：

拖动

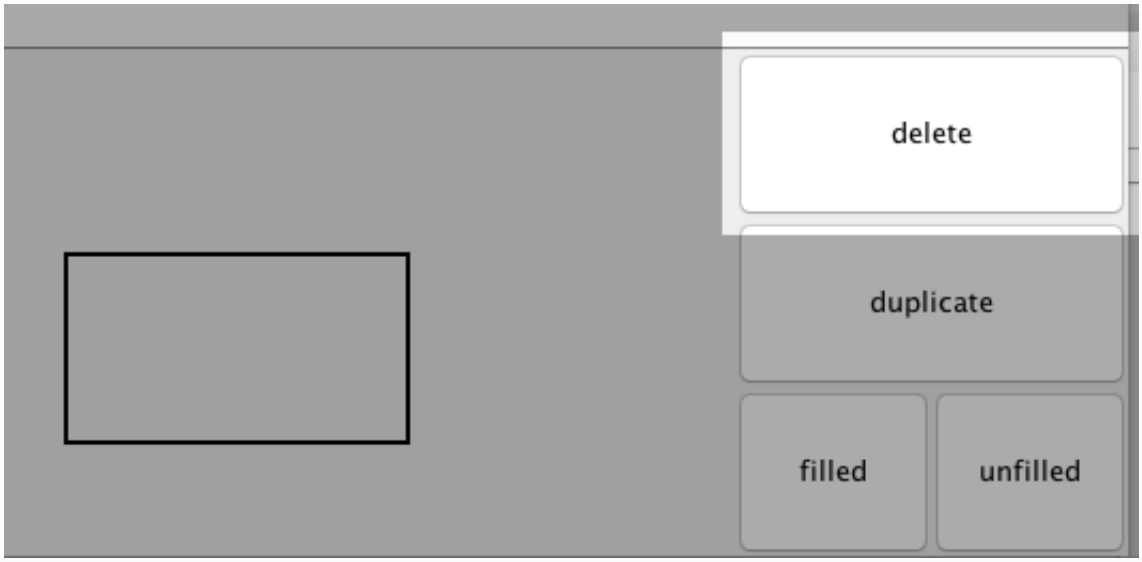
在选择模式下，可以随意拖动图形。拖动时，鼠标必须在要拖动的图形的判定范围内。





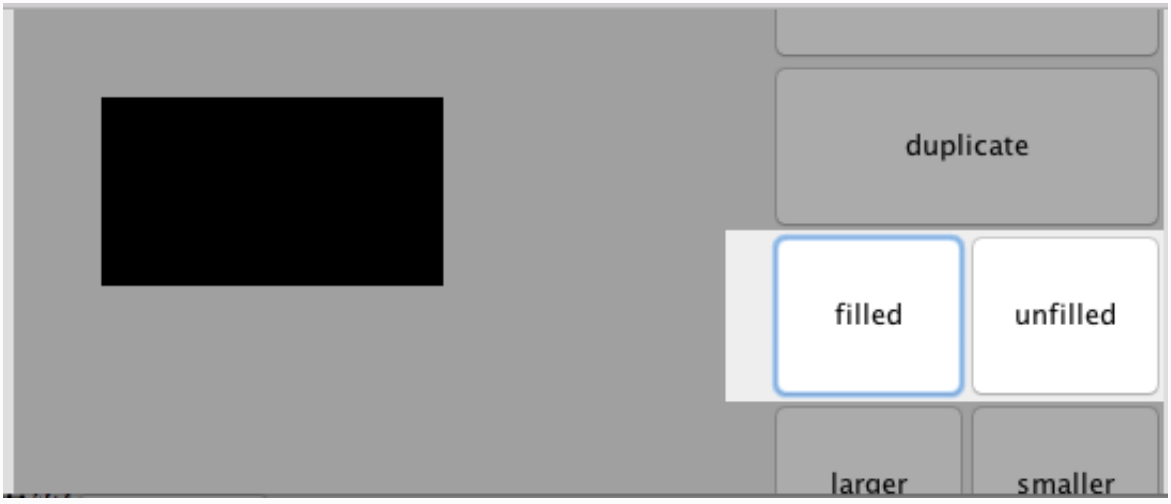
删除

在选择模式下，点击选中要删除的图形，点击右侧操作栏的 `delete` 键：



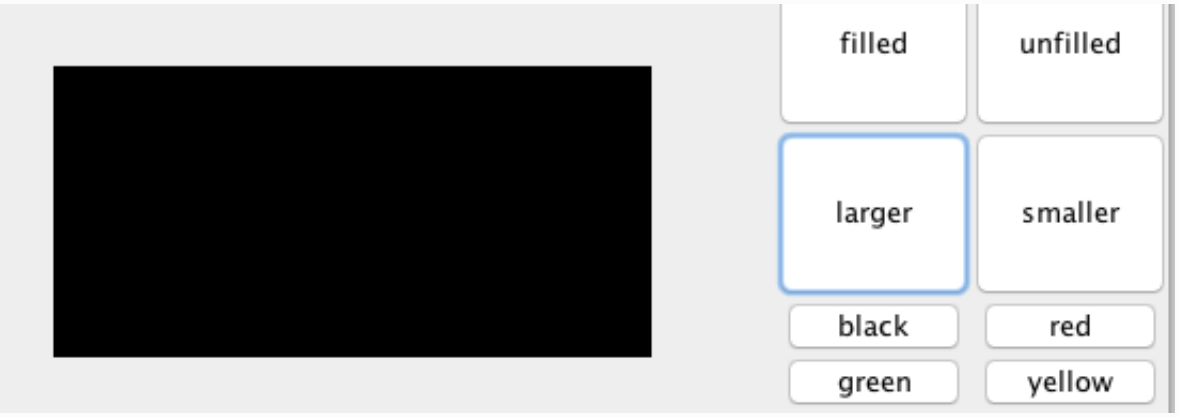
填充/不填充

在选择模式下，点击选中要修改填充的图形，点击右侧操作栏的 `filled/unfilled` 键，可以改变图形填充状态：



大小

在选择模式下，点击选中要修改大小的图形，点击右侧操作栏的 `larger/smaller` 键，可以改变图形填充状态:



颜色

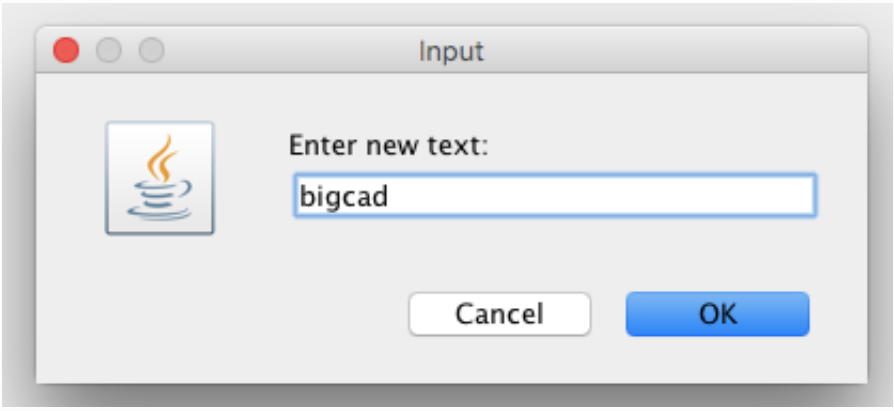
在选择模式下，点击选中要修改颜色的图形，点击右侧操作栏的 `black/red/green/yellow/blue` 键，可以改变图形的颜色:

线条

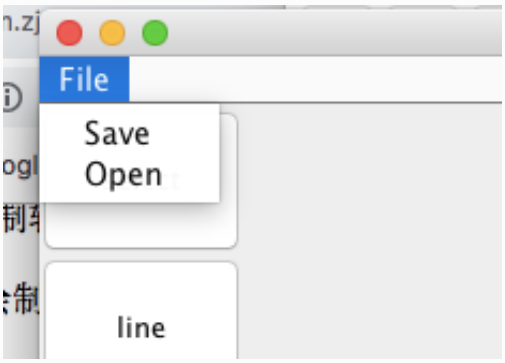
在选择模式下，点击选中要修改线条粗细的图形，在右侧操作栏的`stroke`框中输入所需的粗细（数字），可以改变图形的线条粗细:

修改文字

在选择模式下，点击选中要修改内容的文本，点击右侧操作栏的框中的`new text`键，在弹出的窗口中输入文本，可以改变文字:



保存和加载

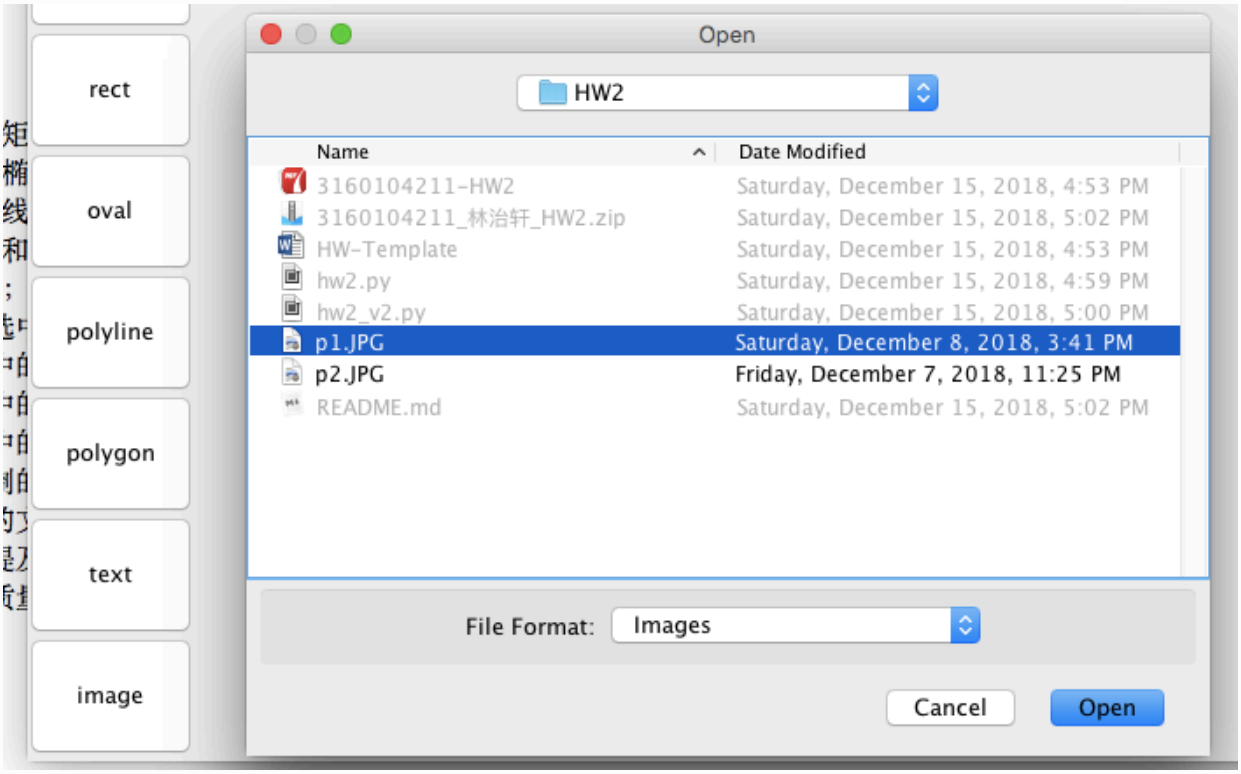


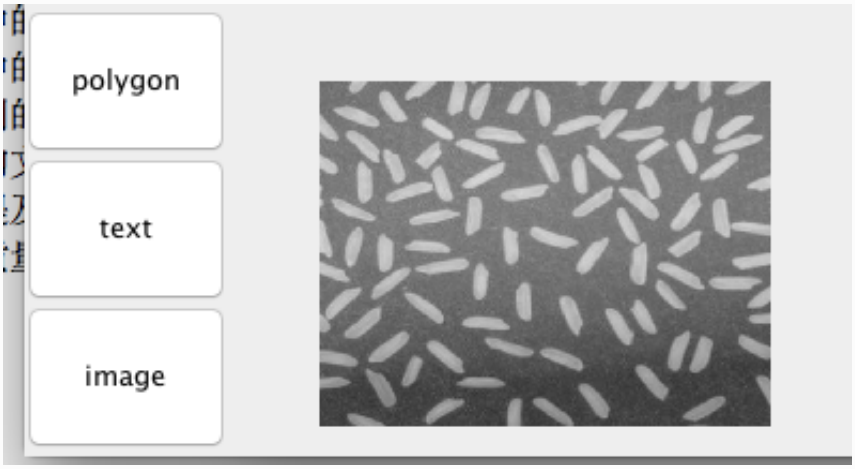
界面左上角可以保存和加载文件。文件会自动加上`.cad`后缀。

其他功能

加载图片

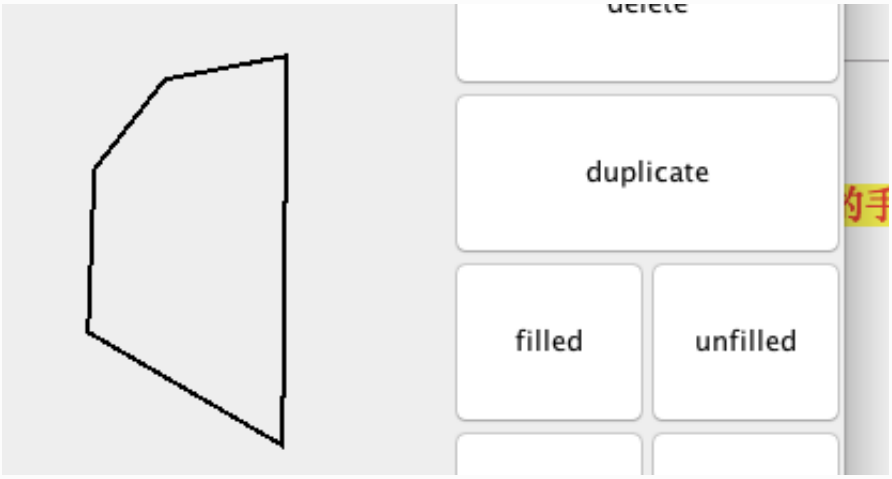
点击模式栏的`image`键，选择图片，然后在绘图区拖动即可加入图片：

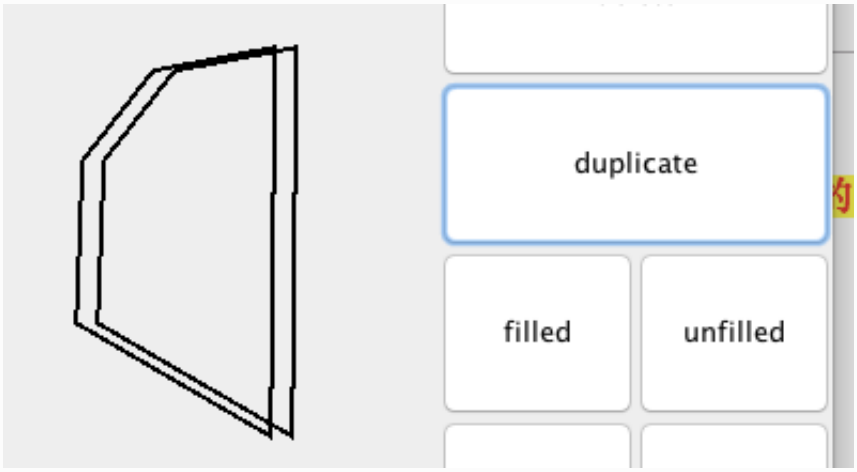




复制

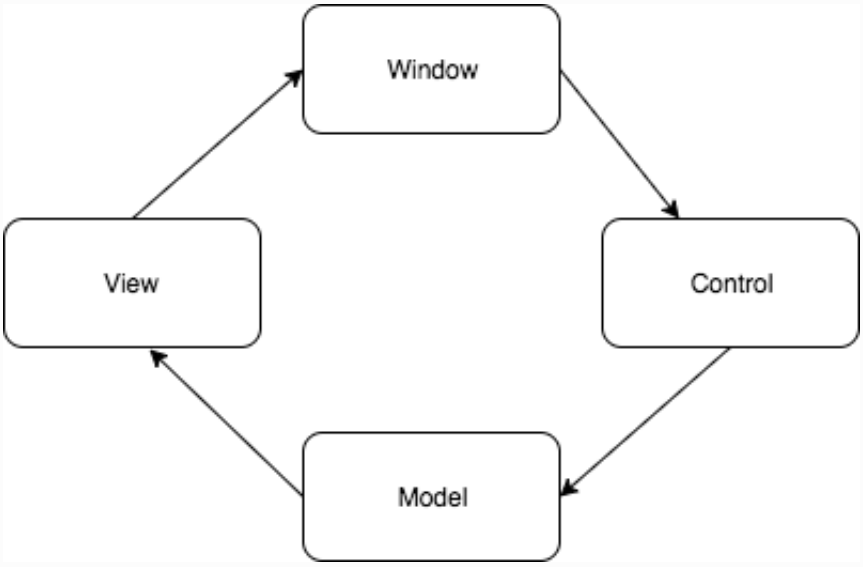
在选择模式下选中图形，点击操作栏的 `duplicate` ，即可复制图形。之后可以拖动将图形分开。





设计与实现

基本框架



程序的基本框架如上图所示。四个模块的基本功能：

- `Window`：该类继承了`JFrame`，即与用户直接交互界面。界面中的绘图板来自`View`。`Window`注册了`Control`的诸多listener，发生事件时会调用这些listener。
- `View`：该类继承了`JPanel`，是实际的绘图板。每当`Model`的数据发生变化时，`View`从`Model`取图形并绘制。
- `Model`：该类保存了所有的图形。
- `Control`：`Control`是框架的核心。有两大功能：
  - 维持一个内在状态`state`。
  - 接受`Window`传来的事件，做状态转换。

Shape

`Shape` 抽象类类是所有图形的子类。公共接口主要有：

```
abstract public class Shape implements
Serializable, Cloneable {
    public void render(Graphics2D g);
    abstract public boolean
fallsWithin(Point point);
    public void makeSmaller(int offset);
    public void makeLarger(int offset);
    public void move(int dx, int dy);
    // set point
    public void setPoint(int index, int x,
int y);
    public void setColor(Color color);
    public void setStroke(float width);
    public void setFilled(boolean filled);

    @Override
    abstract protected Object clone() throws
CloneNotSupportedException;
```

所有的 `Shape` 的位置信息都认为是由一组点唯一确定。这是后面设计状态机的关键。

每个 `Shape` 都必须实现三个方法： `render` ， `fallsWithin` 和 `clone` 。例如， `Rectangle` 的实现：

```
class Rectangle extends Shape {
```

```
    public Rectangle(int x1, int y1, int x2,
int y2) {
        points.add(new Point(x1, y1));
        points.add(new Point(x2, y2));
    }

    @Override
    public void render(Graphics2D g) {
        super.render(g);
        int x = getMinX(), y = getMinY(), w
= getMaxX() - x , h = getMaxY() - y;
        if (filled) {
            g.fillRect(x, y, w, h);
        }
        else {
            g.drawRect(x, y, w, h);
        }
    }
    @Override
    public boolean fallsWithin(Point p) {
        return fallsWithinBoundingBox(p);
    }

    @Override
    protected Object clone() throws
CloneNotSupportedException {
        Rectangle newRect = new Rectangle(0,
0, 0, 0);
        for (int i = 0; i < points.size();
i++) {
            Point p = points.get(i);
            newRect.setPoint(i, p.x +
OFFSET, p.y);
```

```
            }
            return newRect;
        }
    }
```

Model

`Model` 类十分简单。主要内容就是作为所有 `Shape` 的容器。在自己被修改时，会通知 `View` 自己被修改。

```
public class Model {
    public ArrayList<Shape> getShapeList() {
        return shapeList;
    }
    public void modified() {
        view.modified();
    }
    // ...other methods
}
```

View

`View` 类继承了 `JPanel` 。其主要功能就是每次 `Model` 被修改时，遍历其图形并重画：

```
public class View extends JPanel {

    public void modified() {
        repaint();
    }

    @Override
    protected void paintComponent(Graphics
g) {
        super.paintComponent(g);
        ArrayList<Shape> shapeList =
model.getShapeList();
        for (Shape shape : shapeList) {
            shape.render((Graphics2D) g);
        }
    }
    // ...other methods
}
```

Control

Control由两大块组成： `Listener` 和 `State` 。

事件处理

`Control` 内实现了诸多的 `Listener` 。这些 `Listener` 的行为十分类似：调用当前 `state` 的对应函数，并得到新的 `state` 。例如改变大小：

```
public class SizeChangeListener implements
ActionListener {

        @Override
        public void
actionPerformed(ActionEvent e) {
            String str =
e.getActionCommand();
            if (str.equals("larger"))
                state = state.makeLarger();
            else
                state = state.makeSmaller();
        }
    }
```

状态

状态的设计是整个程序的逻辑核心。`State` 是所有状态的父类。`State` 对象的每个方法都代表了这个 `State` 下发生的一个事件。每个事件都返回一个新的状态。

`State` 抽象类定义了所有可能的事件。`State` 的子类根据自己的需要重载，定义每个事件下自己的行为。

```
abstract public class State {
    // constants
    public static final int LINE = 0;
    public static final int RECT = 1;
    public static final int OVAL = 2;
    // ...
```

```
    // all possible events
    public State makeLarger() { return this;
}
    public State makeSmaller() { return
this; }
    public State mouseMoved(MouseEvent e) {
return this; }
    public State mouseDragged(MouseEvent e)
{ return this; }
    // ...other methods

}
```

所有的 `State` 子类语义如下：

- `DrawImage`：选择图片第一个点
- `DrawLine`：选择直线第一个点
- `DrawRect`：选择矩形第一个点
- `DrawOval`：选择椭圆第一个点
- `DrawPolygon`：选择多边形的第一个点
- `DrawPolyline`：选择折线的第一个点
- `DrawText`：选择文本的第一个点
- `Drawing`：选择所有两点图形的第二个点（图像，直线，矩形等）
- `DrawingPoly`：选择所有多点图形的所有点（多边形，折线）
- `Select:`：选择模式

绘制

以画矩形为例，设计的状态有 `DrawRect` 和 `Drawing`。当用户点击模式栏的 `rect` 时，我们进入 `DrawRect` 状态。在该状态下按下鼠标会设置矩形的第一个点，并进入 `Drawing` 状态：

```
class DrawRect extends State {
    public DrawRect(State state) {
        super(state);
    }

    @Override
    public State mousePressed(MouseEvent e)
{
        super.mousePressed(e);
        // add a rectangle
        if (e.getButton() ==
MouseEvent.BUTTON1) {
            shapeList.add(new
Rectangle(e.getX(), e.getY(), e.getX(),
e.getY()));
        }
        model.modified();
        // go into drawing state
        return new Drawing(this);
    }

}
```

在 `Drawing` 状态下，拖动鼠标并放开会确定矩形的第二个点，并返回上一个状态。这样我们可以继续绘制图形：

```
class Drawing extends State {

    public Drawing(State state) {
        super(state);
        last = state;
```

```
            shape =
shapeList.get(shapeList.size() - 1);
        }

        @Override
        public State mouseDragged(MouseEvent e)
{
            super.mouseMoved(e);
            // change second point
            shape.setPoint(1, e.getX(),
e.getY());
            model.modified();
            return this;
        }

        @Override
        public State mouseReleased(MouseEvent e)
{
            // go back to last state
            return last;
        }
        State last;
        Shape shape;
}
```

折线和多变形的绘制于此类似。不同的是，这两个图形可以有多个点，只有在点击鼠标右键时才算完成绘制。

选中

选中功能是所有图形操作的前提。选中状态为 `Select` 。实现非常简单，核心只需判定鼠标点击位置是否落在图形判定范围内即可：

```
// in class State
@Override
public State mousePressed(MouseEvent e) {
    boolean flag = false;
    lastX = e.getX();
    lastY = e.getY();
    if (e.getButton() == MouseEvent.BUTTON1)
{
        for (int i = shapeList.size() - 1; i
>= 0; i--) {
            Shape shape = shapeList.get(i);
            if
(shape.fallsWithin(e.getPoint())) {
                flag = true;
                current = shape;
                currentI = i;
                break;
            }
        }
    }
    if (!flag)
        current = null;
    return this;
}
```

保存和加载

所有的 `Shape` 类都实现了 `Serializable` 接口。在保存时，依次将所有 `Shape` 写入文件：

```
ObjectOutputStream output = new
ObjectOutputStream((new
FileOutputStream(file)));
for (Shape s : model.getShapeList()) {
    output.writeObject(s);
}
output.close();
```

在加载时，依次将所有 `Shape` 读入 `Model` ：

```
ObjectInputStream input = new
ObjectInputStream(new
FileInputStream(file));
ArrayList<Shape> shapeList = new ArrayList<>
();
while (true) {
    Shape s = (Shape)input.readObject();
    shapeList.add(s);
}
```