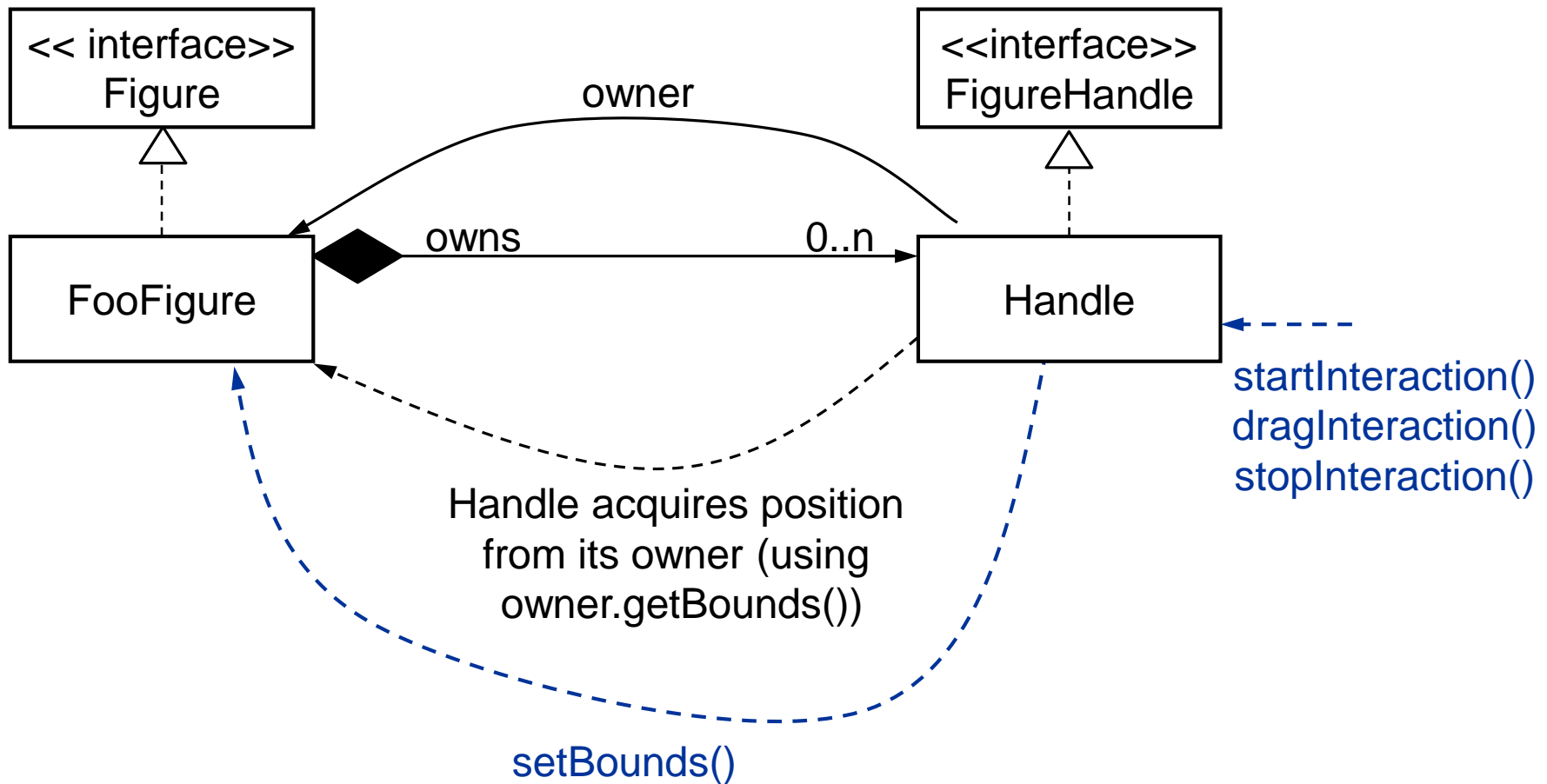
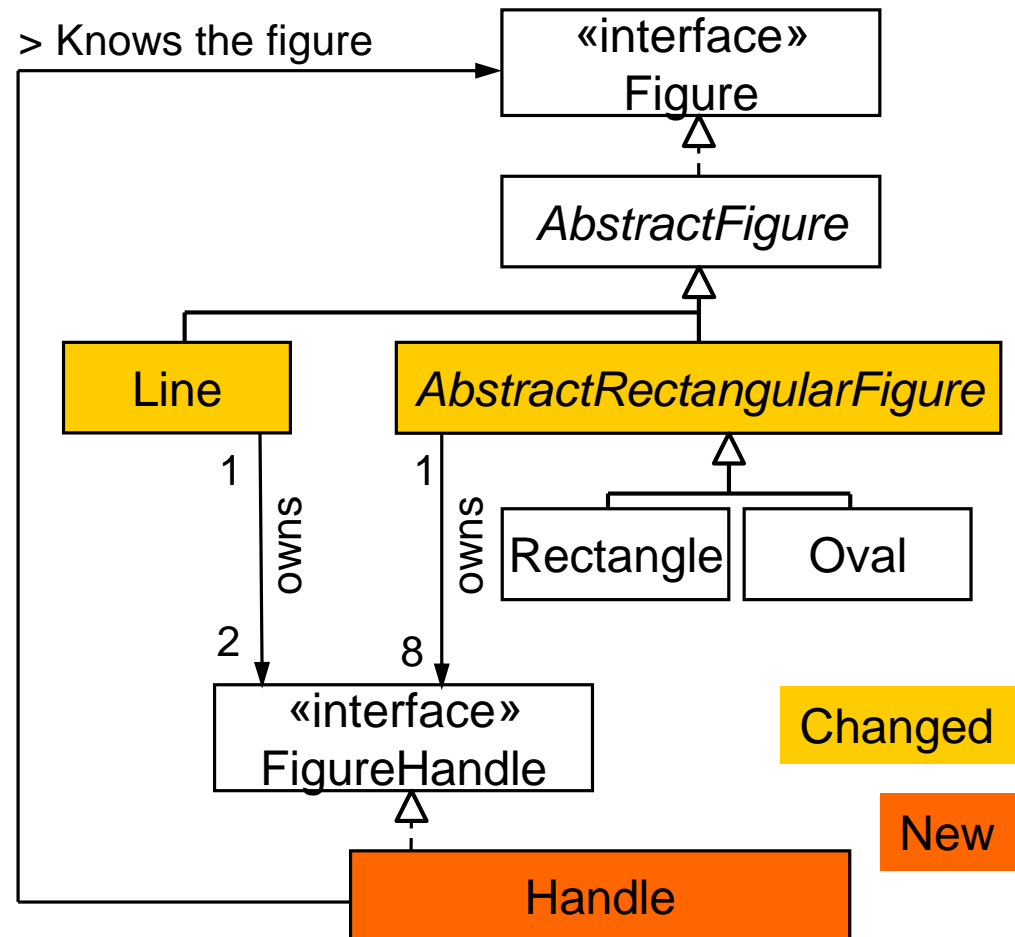


Figure and Handle : Overview



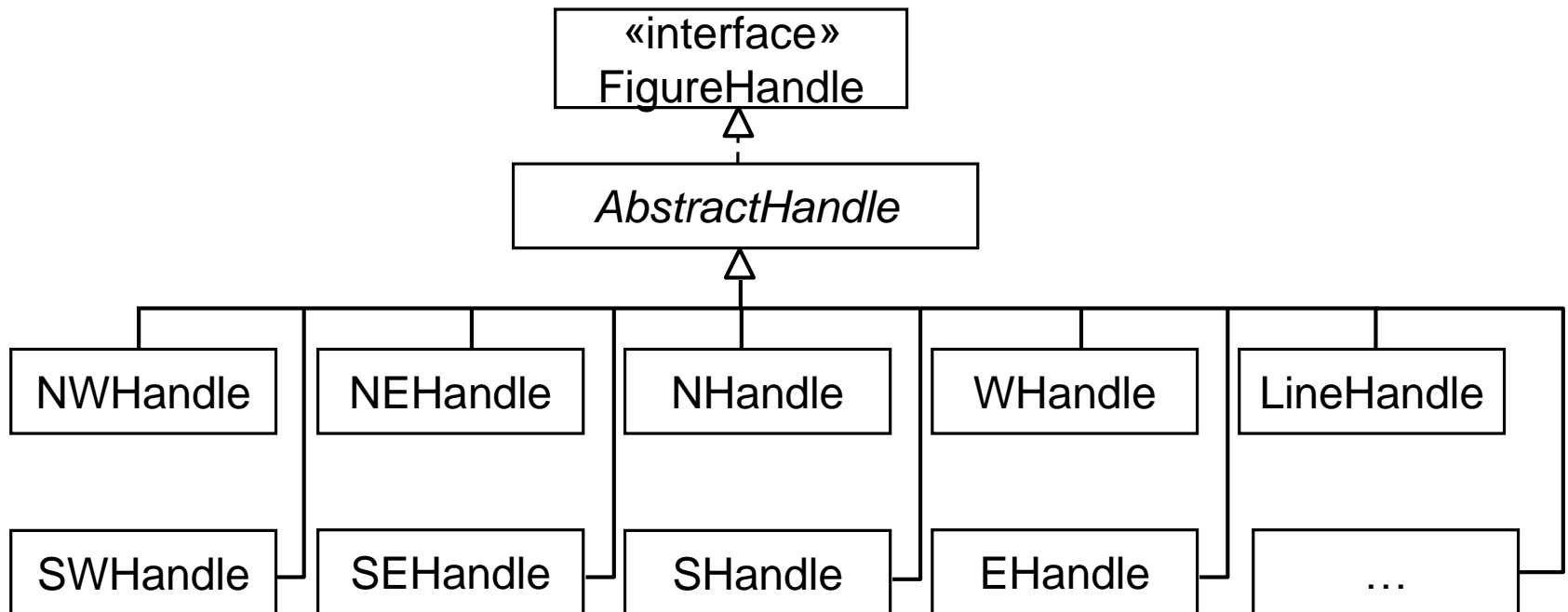
Single Handle Class

- **Single Handle-Class**
 - Behavior of the handle is determined with the arguments passed to the constructor
- => Many switch / if-else statements in the Handle-Class
- **Variants**
 1. Specialization in subclasses
 2. State-Pattern to out-source behavior



Variant 1: Specialization

- Outsourcing of state-dependent behavior in separate classes



Variant 1: Specialization: AbstractHandle

```
public abstract class AbstractHandle implements FigureHandle {  
    private static final int HANDLE_SIZE = 6;  
    private final Figure owner;  
  
    public AbstractHandle(Figure owner) { this.owner = owner; }  
  
    @Override  
    public Figure getOwner() { return this.owner; }  
  
    @Override  
    public boolean contains(int x, int y) {  
        Point loc = getLocation();  
        return Math.abs(x - loc.x) < HANDLE_SIZE / 2  
            && Math.abs(y - loc.y) < HANDLE_SIZE / 2;  
    }  
  
    @Override public void draw(Graphics g) { ... }  
    ...  
}
```

Many methods are the same for all handles if the handle location is given

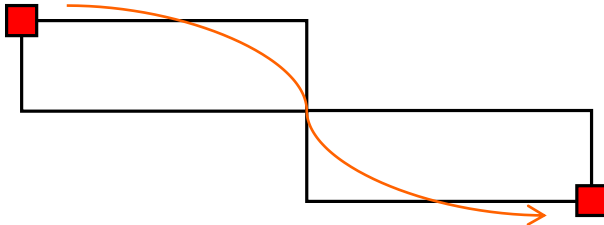
Variant 1: Specialization: NordWestHandle

```
public class NorthWestHandle extends AbstractHandle {  
    @Override  
    public Point getLocation() {  
        Rectangle r = getOwner().getBounds();  
        return new Point(r.x, r.y);  
    }  
    @Override  
    public Cursor getCursor() {  
        return Cursor.getPredefinedCursor(Cursor.NW_RESIZE_CURSOR);  
    }  
    @Override  
    public void dragInteraction(int x, int y, ... ) {  
        Rectangle r = getOwner().getBounds();  
        getOwner().setBounds(new Point(x, y),  
            new Point(r.x + r.width, r.y + r.height));  
    }  
    ...  
}
```

Variant 1: Specialization: NordWestHandle

```
@Override
public void dragInteraction(int x, int y, ...) {
    Rectangle r = getOwner().getBounds();
    getOwner().setBounds(new Point(x,y),
        new Point(r.x+r.width, r.y+r.height));
}
...
```

- **Problem**



- Now $x == r.x + r.width$
and $y == r.y + r.height$

Drag Interaction Problem: Solution A

- **Prevent to move beyond opposite side**
 - Visio uses this approach

```
@Override
public void dragInteraction(int x, int y, ...) {
    Rectangle r = getOwner().getBounds();
    getOwner().setBounds(
        new Point(
            Math.min(x, r.x + r.width),
            Math.min(y, r.y + r.height)),
        new Point(r.x + r.width, r.y + r.height)
    );
}
```

Drag Interaction Problem: Solution B

- **Store opposite corner (or original bounds) in startInteraction**
 - Use the fixed point (or the original bounds) for setBounds calls

```
public abstract class AbstractHandle implements FigureHandle {  
    private Point fixedCorner;  
  
    @Override public void startInteraction(int x, int y, ...) {  
        fixedCorner = getFixedCorner();  
    }  
  
    @Override public void dragInteraction(int x, int y, ...) {  
        owner.setBounds(getVariableCorner(x, y), fixedCorner);  
    }  
  
    @Override public void stopInteraction(int x, int y, ...) {  
        fixedCorner = null;  
    }  
}
```


Drag Interaction Problem: Solution B

- **Store opposite corner (or original bounds) in startInteraction**
 - Use the fixed point (or the original bounds) for setBounds calls

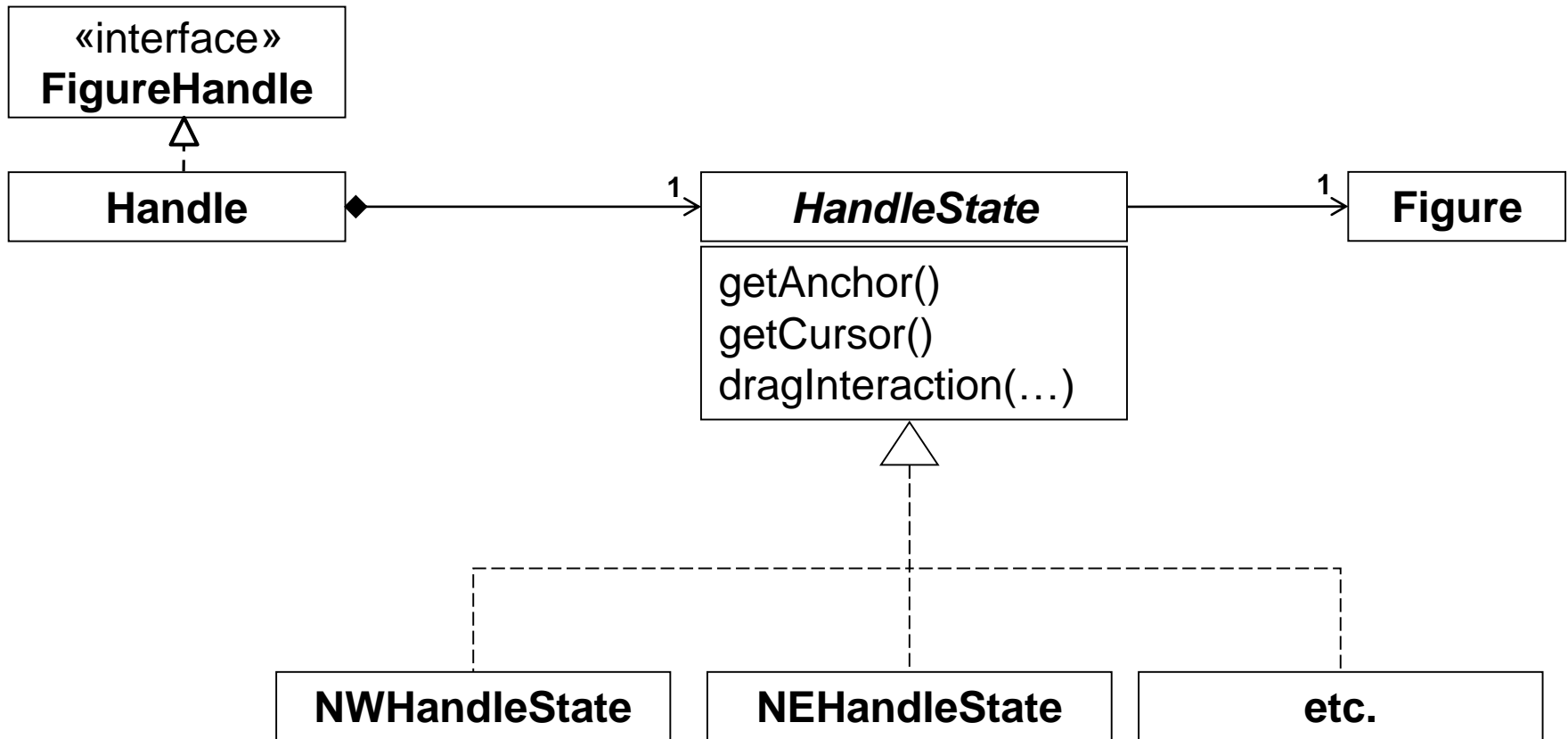
```
public class NorthWestHandle extends AbstractHandle {  
    ...  
  
    @Override  
    public Point getFixedCorner() {  
        Rectangle r = getBounds();  
        return new Point(r.x + r.width, r.y + r.height);  
    }  
  
    @Override  
    public Point getVariableCorner(int x, int y) {  
        return new Point(x, y);  
    }  
}
```

Drag Interaction Problem: Solution B

- **Store opposite corner (or original bounds) in startInteraction**
 - Use the fixed point (or the original bounds) for setBounds calls

```
public class NorthHandle extends AbstractHandle {  
    ...  
  
    @Override  
    public Point getFixedCorner() {  
        Rectangle r = getBounds();  
        return new Point(r.x + r.width, r.y + r.height);  
    }  
  
    @Override  
    public Point getVariableCorner(int x, int y) {  
        // the x-coordinate of the mouse is ignored  
        Rectangle r = getBounds();  
        return new Point(r.x, y);  
    }  
}
```

Variant 2: State Pattern



Variant 2: State Pattern: Handle

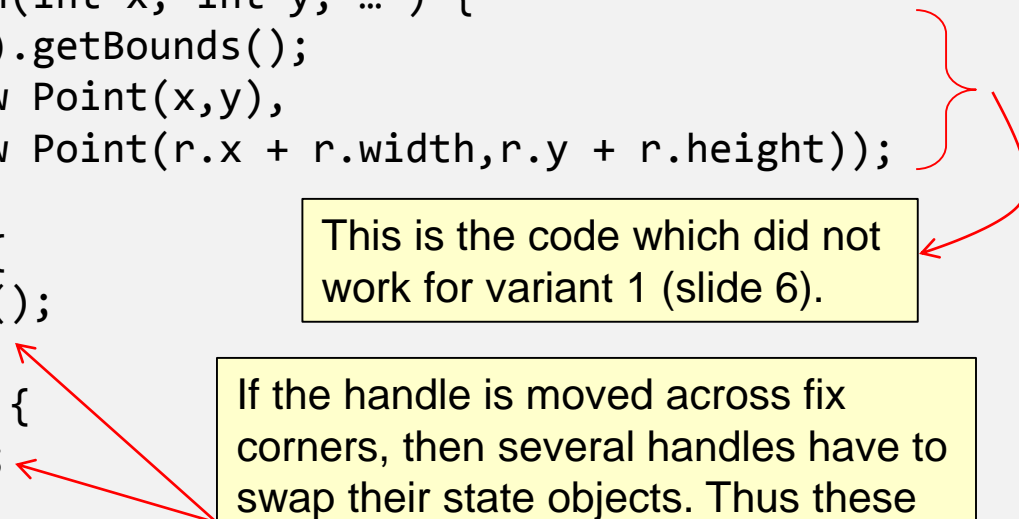
```
public class Handle implements FigureHandle {  
    private HandleState state;  
    public Handle(HandleState state) { this.state = state; }  
  
    public void setState(HandleState state) {  
        this.state = state;   
    }  
    public HandleState getState() { return state; }  
  
    @Override  
    public void dragInteraction(int x, int y, MouseEvent e, DrawView v) {  
        state.dragInteraction(x, y, e, v);  
    }  
  
    @Override  
    public void draw(Graphics g) { ... }  
    ...  
}
```

State of a handle
can be changed
at run-time

dragInteraction() is
delegated to the state
object.

Variant 2: State Pattern: NorthWestState

```
public class NorthWestState extends AbstractHandleState {  
    public NorthWestState(Figure owner) { super(owner); }  
  
    @Override  
    public void dragInteraction(int x, int y, ... ) {  
        Rectangle r = getOwner().getBounds();  
        getOwner().setBounds(new Point(x,y),  
                                new Point(r.x + r.width,r.y + r.height));  
  
        if (x > r.x + r.width) {  
            owner.swapHorizontal();  
        }  
        if (y > r.y + r.height) {  
            owner.swapVertical();  
        }  
    }  
}
```



This is the code which did not work for variant 1 (slide 6).

If the handle is moved across fix corners, then several handles have to swap their state objects. Thus these swapXXX methods have to be implemented at a common place, e.g. in class Figure.

Variant 2: State Pattern: State Exchanger

```
public class Rect extends AbstractRectangularFigure {  
    Handle NW = new Handle(new NorthWestState());  
    ...  
    public void swapHorizontal() {  
        HandleState NWstate = NW.getState();  
        HandleState NEstate = NE.getState();  
        HandleState SWstate = SW.getState();  
        HandleState SEstate = SE.getState();  
        HandleState WState = W.getState();  
        HandleState EState = E.getState();  
        NW.setState(NEstate);  
        NE.setState(NWstate);  
        SW.setState(SEstate);  
        SE.setState(SWstate);  
        W.setState(EState);  
        E.setState(WState);  
    }  
}
```

All horizontally opposing
handles have to be
swapped.

swapVertical analog.

Variant 2: State Pattern: State Exchanger

```
public class Rect extends AbstractRectangularFigure {  
    ...  
    private static void swapHorizontal(Figure owner) {  
        for(FigureHandle fh : owner.getHandles()) {  
            Handle h = (Handle)fh;  
            h.setState(h.getState().swapHorizontal());  
        };  
    }  
  
    public static class NW extends AbstractHandleState {  
        ...  
        @Override public HandleState swapHorizontal() {  
            return new NE(getOwner());  
        }  
        @Override public HandleState swapVertical() {  
            return new SW(getOwner());  
        }  
    }  
}
```

Swapping can also be delegated to the state objects