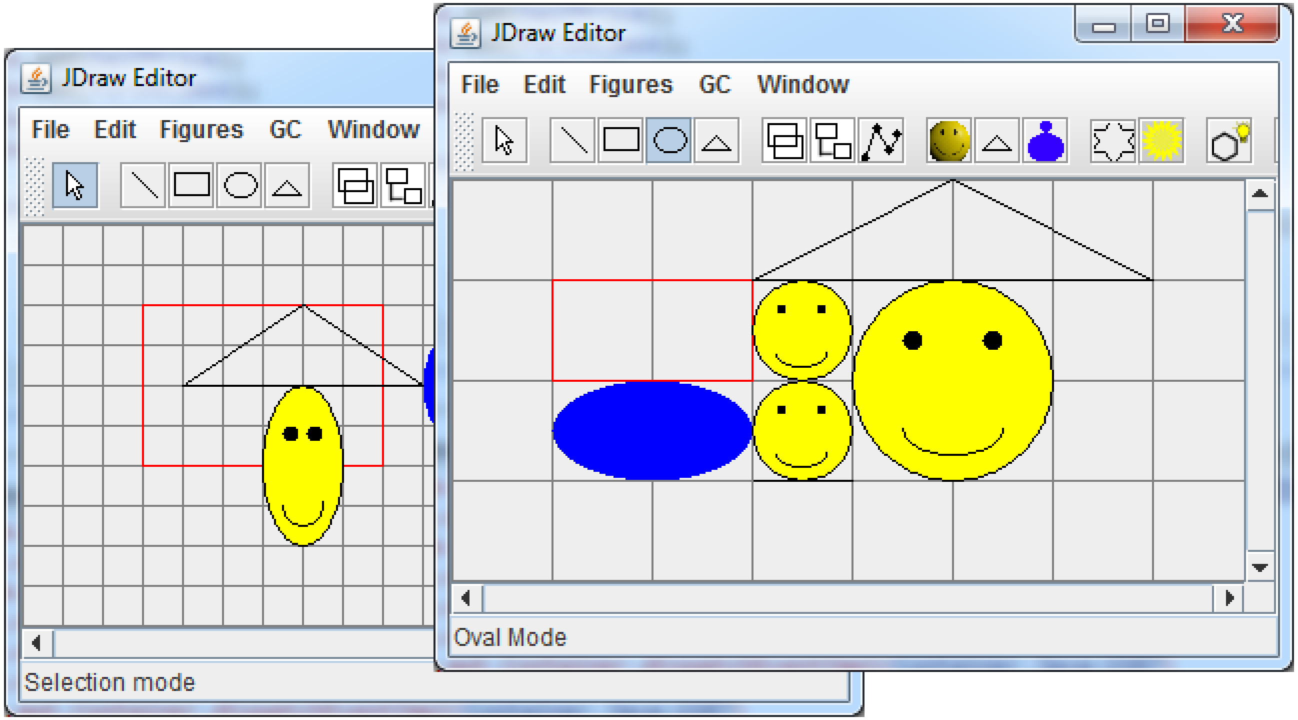
State Pattern

- **Motivation**
  - We would like to constrain the coordinates where figures can be placed or moved to



State Pattern

- **Motivation (code in StdDrawView)**

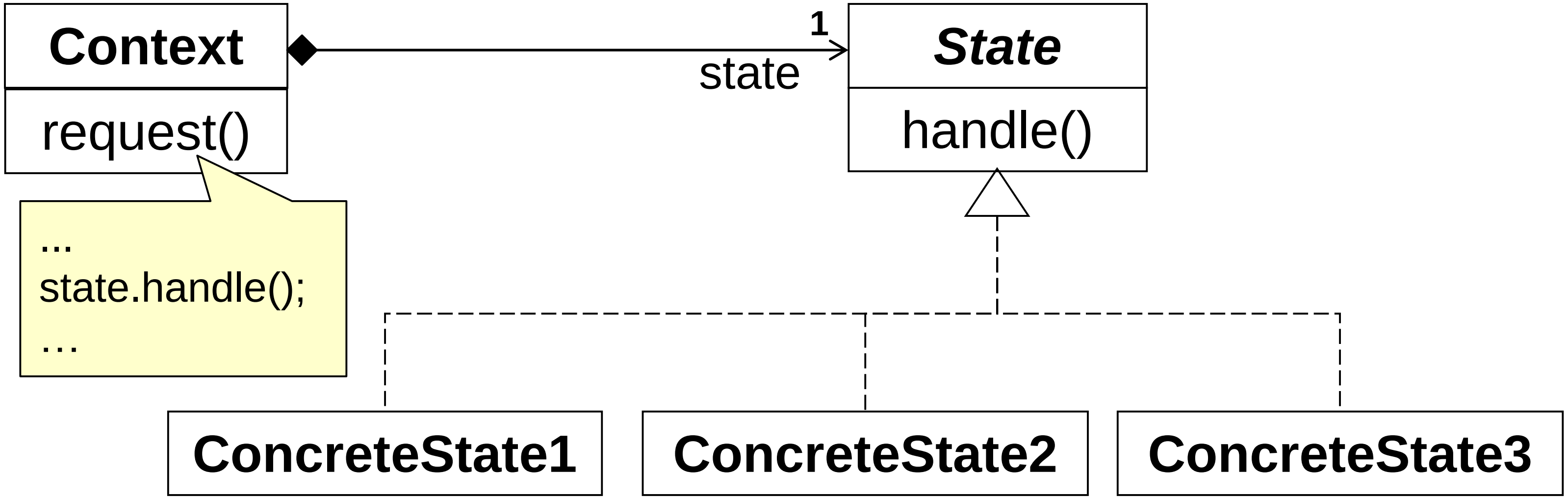
```
@Override
public void mousePressed(MouseEvent e) {
   Point p = new Point(e.getX(), e.getY());
   …
}
@Override
public void mouseDragged(MouseEvent e) {
   Point p = new Point(e.getX(), e.getY());
   …
}
@Override
public void mouseReleased(MouseEvent e) {
   Point p = new Point(e.getX(), e.getY()));
   …
}
```

State Pattern

- **Intent**
  - Allow an object to alter its behavior when its internal state changes
  - Outsourcing of state-dependent behavior

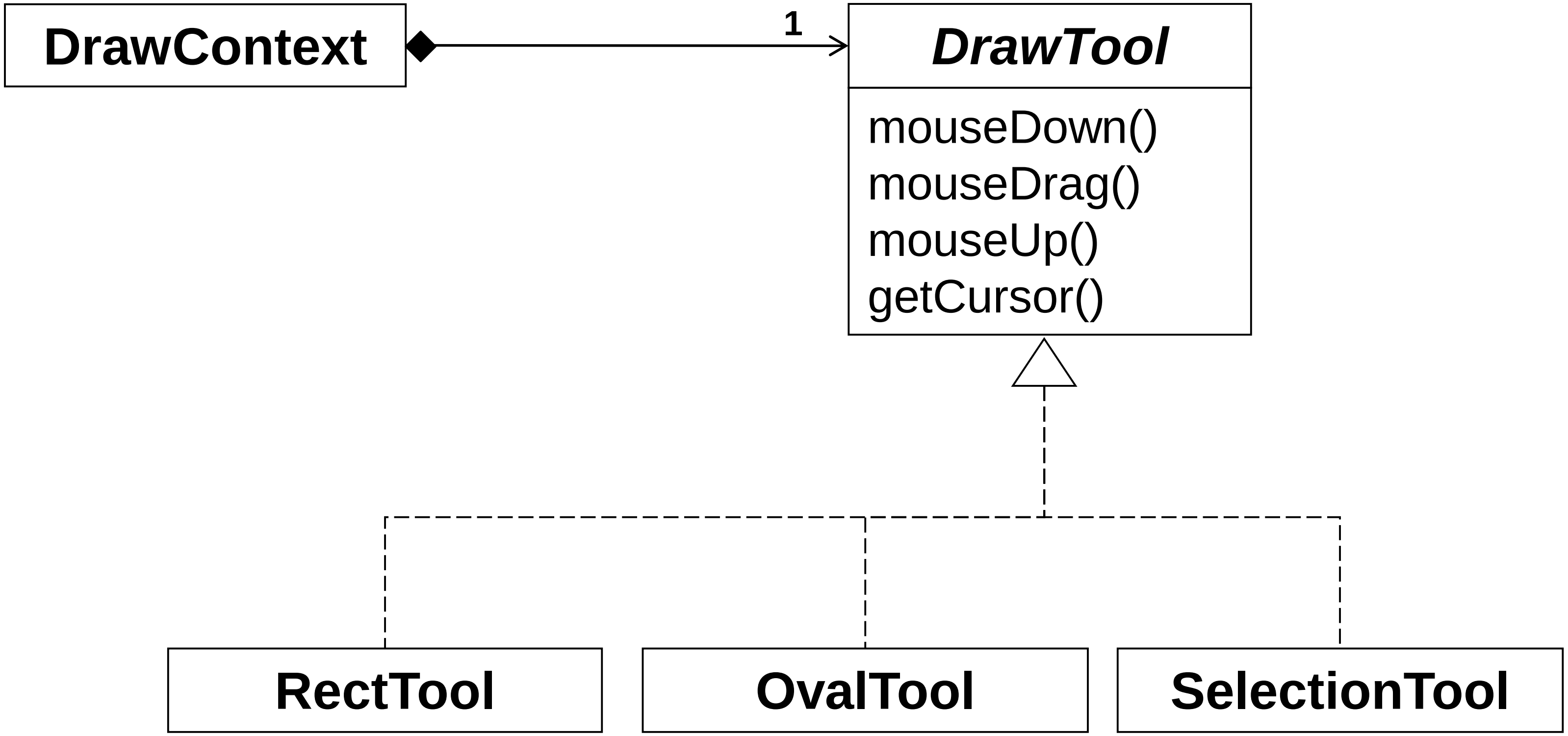
- **Examples**
  - DrawTool (drawing state)
  - DrawGrid (constraining the mouse coordinates)
  - Handles (CTRL/SHIFT pressed)

State Pattern: Structure

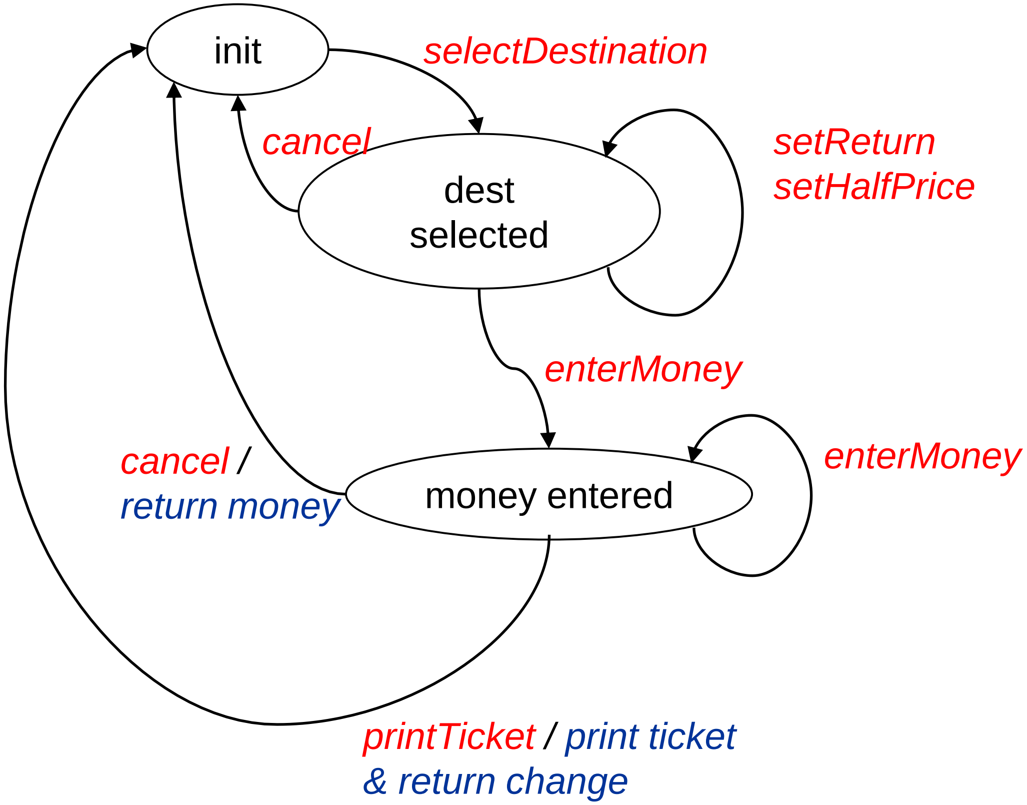


- *Context* contains reference to concrete state which stands for the current state
- *State* defines the interface for the state specific behavior
- *Concrete State* implements state specific behavior

State Pattern: Structure (Example: DrawTool)



State Pattern: Example



State Pattern: Example (without State Pattern)

```
public class TicketMachine {
   private int destination;
   private boolean firstClass;

   private State state = State.INIT;
   private enum State { INIT, DEST_SELECTED, MONEY_ENTERED }

   public void setFirstClass(boolean firstClass) {
      if (state == State.INIT || state == State.MONEY_ENTERED)
         throw new IllegalStateException();
      this.firstClass = firstClass; price = calculatePrice();
   }

   public void enterMoney(double amount) {
      if (state==State.INIT) throw new IllegalStateException();
      if (state==State.DEST_SELECTED) state = State.MONEY_ENTERED;
      this.enteredMoney += amount;
      if (enteredMoney >= price) {
         printTicketWithChange(destination, price, firstClass);
         state = State.INIT;
      }
   }
```

State Pattern: Example (with State Pattern)

```
public class TicketMachine {
   private int destination;
   private boolean firstClass;

   private interface State {
      void setDestination(int destination);
      void setFirstClass(boolean firstClass);
      void setReturnTicket(boolean retour) ;
      void setHalfPrice(boolean halfPrice);
      void enterMoney(double amount);
      void cancel();
   }

   private final State INIT = new StateInit();
   private final State DEST_SELECTED = new StateDestSelected();
   private final State MONEY_ENTERED = new StateMoneyEntered();

   private State state = INIT;
```

State-Interface defines the events which can appear

State Pattern: Example (with State Pattern)

```
public void setFirstClass(boolean firstClass) {
   state.setFirstClass(firstClass);
}
public void enterMoney(double amount) {
   state.enterMoney(amount);
}

abstract class AbstractState implements State {
   public void setDestination(int destination) {
      throw new IllegalStateException(); }
   public void setFirstClass(boolean firstClass) {
      throw new IllegalStateException(); }
   public void setReturnTicket(boolean retour) {
      throw new IllegalStateException(); }
   public void setHalfPrice(boolean halfPrice) {
      throw new IllegalStateException(); }
   public void enterMoney(double amount) {
      throw new IllegalStateException(); }
   public void cancel() { state = INIT; }
}
```

State Pattern: Example (with State Pattern)

```
class StateDestSelected extends AbstractState {
   public void setFirstClass(boolean fc) {
      firstClass = fc;
      price = calculatePrice(destination, firstClass);
   }
   public void enterMoney(double amount) {
      state = MONEY_ENTERED; state.enterMoney(amount);
   }
}

class StateMoneyEntered extends AbstractState {
   public void enterMoney(double amount) {
      enteredMoney += amount;
      if (enteredMoney >= price) {
         printTicketWithChange(destination, price, firstClass);
         state = INIT;
      }
   }
}
```

State Pattern: Issues

- **State transition**
  - Decentralized: may be initiated by state objects
    - States must know its successors
    - States need access to a state transition method in the context
      - `context.setState(s);` (or inner classes)
    - State methods may also return the new state (which is then set by the context)
  - Parameterized: may be signaled by state, executed by context
    - State returns a key (e.g. a String) which describes the new state
    - Association between keys and states is hold in the context
      - State transitions are configurable
      - Separation state-behavior vs transitions
      - State machine may be changed without changing state implementations
  - Centralized: may be initiated by the context
    - State should be informed that it is activated or passivated

State Pattern: Issues

- **Example: DrawTools**
  - DrawTool

```
public interface DrawTool {
   void activate();
   void deactivate();
   ...
}
```

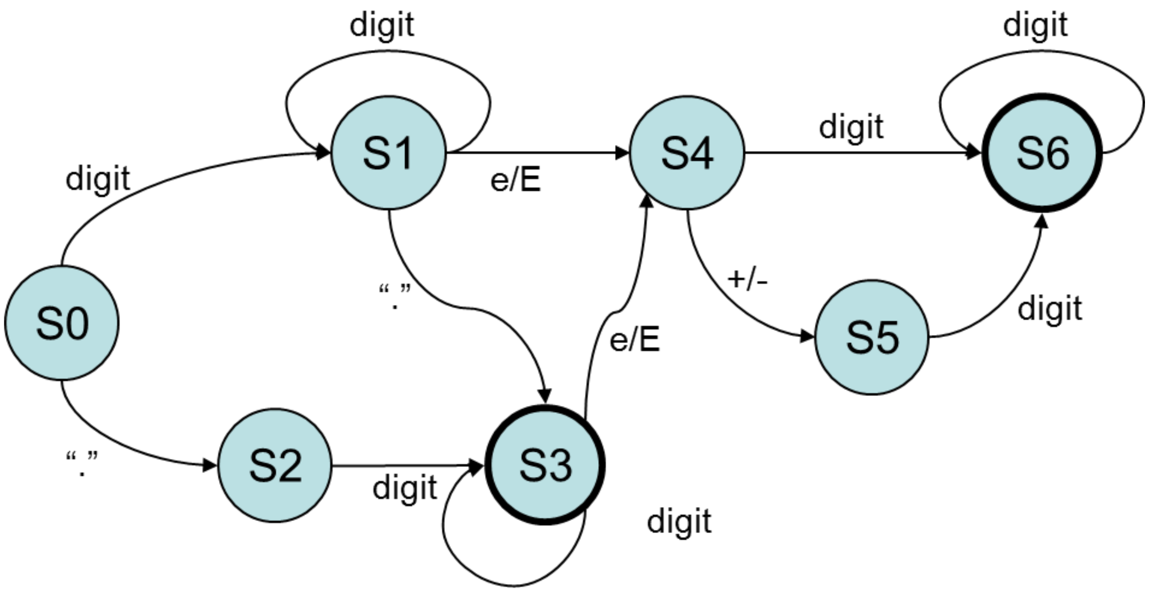
  - DrawView

```
public void setTool(DrawTool tool) {
   if(tool == null) throw new IllegalArgumentException();
   if(this.tool != null) this.tool.deactivate();
   this.tool = tool;
   this.tool.activate();
}
```

If the tool reference were initialized in the constructor of the DrawView, then this test could be omitted.

State Pattern: Example

- **Float Parser**
  - Valid: 1.33 0.4e10 .3 .4E+5 4e-3



State Pattern: Issues

- **Creation of state objects**
  - Created when needed
    - States need to know the concrete state classes (and depend on them)

```
public void stateMethod1(Context c) {
   ...
   c.setState(new StateB());
}
```

    - Useful if state changes happen infrequently
  - Created ahead of time
    - States have to be stored in variables accessible by all states

```
public void stateMethod1(Context c) {
   ...
   c.setState(c.STATE_B);
}
```

    - Useful if state changes occur rapidly

State Pattern: Applicability

- **State dependency** *Design / Architecture*
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
- **Separation** *Refactoring*
  - A class contains many behaviors which appear in multiple conditional statements (switch / if)
  - State is usually represented by one or more enumerated constants
  - Often several operations will contain this same conditional structure

  => move related conditional branches into their own state class