the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”

might return different responses dependent upon the state of the relationship, or the state of mind of the partner.

Same partner but different behaviors depending on the state of mind

each partner ‘HAS-A’ state of mind, and it changes
as in the *Strategy Pattern*, delegate (or hand off) the task to some other class better able to perform the task:

```python
class A(object):
    self.d = Delegate()

    def m1():
        self.d.m1()

    def m2():
        self.d.m2()

So think of *Partners* as having multiple states

```python
class Partner(object):
    self.grumpyState = GrumpyState();
    self.happyState = HappyState();

    def whatz4DinDin():
        if (...):  # there's a better way to do this
            self.grumpyState.m1()
        else
            self.happyState.m1();
```
we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

```python
class Partner(object):
    def __init__(self):
        ...

    def whatz4DinDin(self):
        self.currentState.whatz4DinDin()

    def wannaGo2PDX(self):
        self.currentState.wannaGo2PDX()

    def negate(v):
        return !v

    def addOne(i):
        return i+1
```
HOW TO GET THESE METHODS TO DEPEND ON STATE?

class Partner(object):
    g = GrumpyState()
    h = HappyState()

def whatz4DinDin(): ... depends on g versus h
def wannaGo2PDX(): ... depends on g versus h

Answer: create an ‘interface’ or an ‘abstract class’ to define which methods are state-dependent:

```
abstract public class PartnerState {
  abstract public void whatz4DinDin();
  abstract public void wannaGo2PDX();
}

class GrumpyState extends PartnerState {
  ...
}
class HappyState extends PartnerState {
  ...
}
```

class PartnerState(object):
    def whatz4DinDin():
        pass
    def wannaGo2PDX():
        pass

class GrumpyState(PartnerState): ...
class HappyState(PartnerState): ...
class PartnerState(object):
    def __init__(self, owner):
        self.owner = owner

    def whatz4DinDin(self):
        pass

    def wannaGo2PDX(self):
        pass

class GrumpyState(PartnerState):
    def whatz4DinDin(self):
        print "road kill"

    def wannaGo2PDX(self):
        print "no"

class HappyState(PartnerState):
    def whatz4DinDin(self):
        print "chicken"

    def wannaGo2PDX(self):
        print "yeah!!!"
THE STATE PATTERN (JAVA)

class Partner {
  private GrumpyState  grumpyState;
  private HappyState   happyState;
  private PartnerState currentState;

  public Partner() {
    grumpyState  = new GrumpyState();
    happyState   = new HappyState();
    currentState = happyState;
  }

  public void whatz4DinDin() { currentState.whatz4DinDin(); }
  public void wannaGo2PDX()    { currentState.wannaGo2PDX(); }

  public boolean negate(boolean v) { return !v; }
  public int     addOne(int i)     { return i+1; }
}
abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}

public class SleepState extends PartnerState {
    public void whatz4DinDin() { System.err.println("mumble"); }
    public void wannaGo2PDX() { System.err.println("mummph"); }
}
THE STATE PATTERN (PYTHON)

class Partner(object):
    def __init__(self):
        self.grumpyState = GrumpyState(self)
        self.happyState = HappyState(self)
        self.currentState = None
        self.setHappy()

    def setHappy(self):
        self.currentState = self.happyState

    def setGrumpy(self):
        self.currentState = self.grumpyState

    def whatz4DinDin(self):
        self.currentState.whatz4DinDin()

    def wannaGo2PDX(self):
        self.currentState.wannaGo2PDX()

    public boolean negate(boolean v) { return !v; }
    public int     addOne(int i)     { return i+1; }
CHANGING STATE (THREE OPTIONS)

1) only the class Partner can change its own state

```python
self._currentState = self._grumpyState
```

with only ‘private’ access to its state variables

2) anyone, might change the private state variables provided there are setter methods:

```python
makeGrumpy()
makeHappy()
```

3) the Partner’s private states themselves might decide to change the parent Partner’s state (behind its back)
   [that requires States to know their owners]
class Partner {
  private GrumpyState  grumpyState;
  private HappyState  happyState;
  private PartnerState  currentState;

  public Partner() {
    grumpyState  = new GrumpyState();
    happyState  = new HappyState();
    currentState = happyState;
  }

  private void setGrumpy() { currentState = grumpyState; }
  private void setHappy() { currentState = happyState; }

  public void whatz4DinDin() {
    currentState.whatz4DinDin();
    setGrumpy();
  }

  public void wannaGo2PDX() {
    currentState.wannaGo2PDX();
    setHappy();
  }
}
class Partner(object):
    def __init__(self):
        self._grumpyState = GrumpyState(self)
        self._happyState = HappyState(self)
        self._currentState = None
        self._setHappy()

    def _setHappy(self):
        self._currentState = self._happyState

    def _setGrumpy(self):
        self._currentState = self._grumpyState

    def whatz4DinDin(self):
        self._currentState.whatz4DinDin()

    def wannaGo2PDX(self):
        self._currentState.wannaGo2PDX()
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
    }
}

and maybe HappyState can change parent to GrumpyState as well ...
CLASS WITH PUBLICLY CHANGEABLE STATE (JAVA)

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState(this);
        happyState = new HappyState(this);
        currentState = happyState;
    }

    public void setGrumpy() { currentState = grumpyState; }
    public void setHappy()  { currentState = happyState;  }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX();  }
}

where the states change the Partner’s state without it knowing!
class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void setGrumpy() { currentState = grumpyState; }
    public void setHappy()  { currentState = happyState;  }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX();  }

    where anybody can change the state at any time ...
extra credit (non-quantitative)

implement the midterm
package cm;

public class CMDriver {
    public static void main (String args[])
    {
        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}

A SOFT DRINK DISPENSER (JAVA)
A SOFT DRINK DISPENSER (JAVA)

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
coin inserted...
Please make a selection...
Selection pushed...
Dispensing item...
package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

public class Quarter extends Coin {
    public Quarter() { super(25); }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
   while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}

public int getValue() { return tempValue; }
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State currentState;
    private Ready ready;
    private Accepting accepting;
    private Selecting selecting;
    private CoinBox coinBox;

    public Dispenser() {
        coinBox = new CoinBox();
        ready = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }

    public void insertCoin(Coin c) {
        System.out.println("coin inserted...");
        currentState.accept(c);
    }
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
    coinBox.transferToSafe();
    return true;
}
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }

void setReady() { currentState = ready; }
void setAccepting() { currentState = accepting; }
void setSelecting() { currentState = selecting; }
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {} 
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection... ");
            owner.setSelecting();
        }
    }

    public void select(int n) { System.out.println("Please insert more coins"); } 
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}

THE SELECTING STATE (JAVA)
THE RESPONSIBILITY FOR SETTING STATE

from the text (p. 412):

“The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes.”

AKA:
  tightly coupled!
  co-dependency!