producing software out of thin air

• determining the primary objects of the application
  – think of objects as active agents
    anthropomorphize (give objects intentionality)
    objects have responsibilities (= methods)
    objects interacting with other objects
      collaborations
      sharing information
        need-to-know access
        pushing vs. pulling information
    events that trigger interactions
      cascading events
      events as objects too
    changes of state within some objects, etc.

• how to start?
producing software out of thin air

• Use Cases
  – deciding on what you want to achieve
    software requirements (specifications)
    case-by-case (scenario-by-scenario)
  – start with most important requirements
    (not error cases)
  – for each case or scenario:
    what objects perform what roles?
    with what other objects?
  – noun phrases = objects
  – adjectives = subclasses/superclasses of objects
  – verbs = methods

• Design and analyze before programming
  – diagram to make your ideas concrete
    UML class diagrams (this isa that and hasa that)
    UML sequence diagrams (this calls that in that then that calls ...)
the perils of premature coding
(or: writing without thinking enough)

class Ctr {
    private int count;
    public Ctr() {
        count = 0;
    }
    public void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}

class Ctr(object):
    def __init__(self):
        self._count = 0
    def increment(self):
        self._count += 1
    def getCount(self):
        return self.count

@property
def count(self):
    return self._count

@count.setter
def count(self, value):
    self._count = value

def main():
    c = Ctr()
    for i in range(4):
        c.increment()
        print c.getCount()

results in:
1
2
3
Then maybe we decide we want to count by 2 or any other step size

class Ctr {
    private int cont, inc;

    public Ctr(int stepSize) {
        count = 0;
        inc = stepSize;
    }

    public Ctr() { this(1); }

    public void increment() {
        count += inc;
    }

    public int getCount() {
        return count;
    }
}

class Ctr(object):
    def __init__(self, stepSize=1):
        self._count = 0
        self._inc = stepSize

    def increment(self):
        self._count += self.inc

    def getCount(self):
        return self._count

    @property
def count(self):
        return self._count

    @count.setter
def count(self, value):
        self._count = value

    @property
def inc(self):
        return self._inc

    @inc.setter
def inc(self, value):
        self._inc = value
public static void main (String[] args) {
    Ctr c = new Ctr(2);
    for (int i = 0; i < 3; i++) {
        c.increment();
        System.err.println(c.getCount());
    }
}

>> 2, 4, 6, ...

def main():
    c = Ctr(2)
    for i in range(4):
        c.increment()
        print c.getCount()

>> 2, 4, 6, ...
inheritance rather than revising class

extend Ctr with a subclass Ctr2 which counts in increments of 2:

class Ctr {
    protected int count, inc;
    public Ctr() { this(1); } 
    public Ctr(int stepSize) {
        count = 0;
        inc   = stepSize;
    }
    public void increment()  { count += inc; } 
    public int   getCount()  { return count; } 
}

class Ctr2 extends Ctr {
    public Ctr2() { super(2); }
    public void decrement() { count -= inc; }
}

note we slipped in a new method called decrement
inheritance rather than revising class

and adding even more functionality, e.g. a way to report the number of increments and decrements:

```java
class Ctr2 extends Ctr {
    protected int numIncs, numDecs;
    public Ctr2() { super(2); }

    public void decrement() {
        count -= inc;
        numDecs++;
    }

    public int numIncrements() {
        return numIncs;
    }

    public int numDecrements() {
        return numDecs;
    }
}
```

```python
class Ctr2(Ctr):
    def __init__(self):
        Ctr.__init__(self, 2)
        self._numIncs = 0
        self._numDecs = 0

    def decrement(self):
        self._count -= self._inc
        self._numDecs += 1

    def numIncrements():
        return numIncs

    def numDecrements():
        return numDecs
```

has any bug been introduced?
class Ctr {
    private int count = 0;
    private int numIncs = 0;
    private int numDecs = 0;
    private int inc;

    public Ctr(int stepSize) {
        inc = stepSize;
    }
    public Ctr() {
        this(1);
    }

    public void increment() {
        count += inc;
        numIncs++;
    }
    public void decrement() {
        count -= inc;
        numDecs++;
    }
    public int numIncrements() { return numIncs; }
    public int numDecrements() { return numDecs; }
    public int getCount() { return count; }
    public void reset() {
        count = 0;
        numIncs = 0;
        numDecs = 0;
    }
}

class Ctr2 extends Ctr {
    public Ctr2() { super(2); }
}
fix by refactoring (moving methods around)

class Ctr(object):
    def __init__(self, stepSize=1):
        self._count = 0
        self._inc = stepSize
        self._numIncs = 0
        self._numDecs = 0

    def increment(self):
        self._count += self._inc
        self._numIncs += 1

    def decrement(self):
        self._count -= self._inc
        self._numDecs += 1

    def getCount(self):
        return self._count

    def numIncements(self):
        return self._numIncs

    def numDecrements(self):
        return self._numDecs

    def reset(self):
        self._count = 0  # etc.

Class Ctr2(Ctr):
    def __init__(self):
        Ctr.__init__(self, 2)
the lessons learned

- incremental evolution of designs practically inevitable

- look for **symmetries** in your design; be cautious of breaking symmetries

- be ready to **refactor** to enhance symmetry, regularity, elegance

- **test** thoroughly each change (don't presume)

- analyze rather than hack

- don't be in a hurry to jump into programming

- design for extensibility, testability, ... other ity's

- design/program with tight adherence to OO principles

- attend to your programming style, too!