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?
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)
PERILS OF WRITING CODE EVOLUTIONARILY
(or: writing without thinking enough)

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

class Test {
    public static void main (String[] args) {
        Ctr c = new Ctr();
        for (int i = 0; i < 3; i++) {
            c.increment();
            System.err.println("counter = " + c.getCount());
        }
    }
}

>> java Test
counter = 1
counter = 2
counter = 3 ... and so forth
```
Then maybe we decide we want a new feature: incrementing by a specified step size (not just by 1)

class Ctr {
    private int n, inc;

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

    public Ctr() { this(1); }

    public void increment() { n += inc; }
    public int  getCount()  { return n; }
}

... and next, to test it ...
class Test {
    public static void main (String[] args) {
        Ctr c = new Ctr(2);
        for (int i = 0; i < 3; i++) {
            c.increment();
            System.err.println("counter = " + c.getCount());
        }
    }
}

>> counter = 2, 4, 6, ...
class Ctr {
    protected int n, inc;
    public Ctr() { this(1); }
    public Ctr(int stepSize) {
        n = 0;
        inc = stepSize;
    }
    public void increment() { n += inc; }
    public int getCount() { return n; }
}

defining the following count-by-two class

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

and note we slipped in a new method called decrement
INHERITANCE VERSUS REHACKING (continued...)

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

class Ctr2 extends Ctr {
    protected int numIncs, numDecs;
    public Ctr2() { super(2); }
    public void decrement() {
        n -= inc;
        numDecs++;
    }
    public int numIncrements() { return numIncs; }
    public int numDecrements() { return numDecs; }
}

has any bug been introduced?
Refactoring (moving methods around)

class Ctr {
    private int n = 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() { n += inc; numIncs++; }
    public void decrement() { n -= inc; numDecs++; }
    public int numIncrements() { return numIncs; }
    public int numDecrements() { return numDecs; }
    public int getCount() { return n; }
    public void reset() { n = 0; numIncs = 0; numDecs = 0; }
}

class Ctr2 extends Ctr {
    public Ctr2() { super(2); }
}
THE LESSONS

incremental evolution of designs practically inevitable

look for **symmetries** in your design, and 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
CREATING A HIERARCHY OUT OF THIN AIR

Idea:

two mutual antagonists (Warrior). Their Swords and Armor can have different hardnesses. The Sword must be made of sufficiently hard metal to puncture the Armor or Shield of the enemy. If it is, the opponent’s life energy is reduced, otherwise the opponent simply scoffs at the individual, and blah blah...

Use cases:

individual Enemy has a sword made of sterner stuff than its enemy's armor. Reduce opponent’s life energy. Follow on with consequences of reduced life energy (e.g., die)

classes:
Sword, Shield, Armor, Warrior, ....

methods:
stab(), ...
CREATING A HIERARCHY OUT OF THIN AIR

This is definitely shoot-from-the-hip programming (also called shoot-yourself-in-the-foot programming)

class MetalThing {
    public int hardness;
    MetalThing(int hardness) { this.hardness = hardness; }
    public String toString() { return ("Hardness = " + hardness + " "); }
}

class Shield extends MetalThing {
    Shield(int hardness) { super(hardness); }
}

class Armor extends MetalThing {
    Armor(int hardness) { super(hardness); }
}

Shield and Armor thought of first... then superclass MetalThing to encapsulate the idea of hardness that was starting to be seen as a recurring theme.

the toString method came to mind only when trying to debug it
then swords came to mind, which I knew would involve instances of the Enemy class

class Sword extends MetalThing {
    protected int length;

    Sword(int length, int hardness) {
        super(hardness);
        this.length = length;
    }

    public void stab(Warrior w) { w.beStabbed(this); }
    public String toString() {
        return super.toString() + "Length = " + length + " ";
    }
}

class BattleSword extends Sword {
    protected int width;

    BattleSword(int l, int bladeWidth, int hard) {
        super(l, hard);
        width = bladeWidth;
    }
    public String toString() {
        return super.toString() + "Width = " + width + " ";
    }
}
class Warrior {
    final int FULL_OF_LIFE = 100;
    final int MINIMUM_LIFE = 10;
    final int SWORD_LENGTH = 2;

    String name;
    Shield shield;
    Armor armor;
    Sword sword;
    int lifeEnergy = FULL_OF_LIFE;

    Warrior(int hardness, String name) {
        this.name = name;
        shield = new Shield(hardness);
        armor = new Armor(hardness);
        sword = new Sword(SWORD_LENGTH, hardness);
        System.err.println(name + " created with: ");
        System.err.println("\tshield:" + shield.toString());
        System.err.println("\tsword:" + sword.toString());
        System.err.println("\tarmor:" + armor.toString());
    }
}
public void stab(Warrior w) { w.beStabbed(sword); }

public void beStabbed(Sword s) {
    if (armor.hardness < s.hardness) {
        System.out.println(name + ": stabbed by a " + s.length + "-foot long sword!");
        lifeEnergy /= 10;
        if (lifeEnergy < MINIMUM_LIFE)
            System.out.println(name + ": farewell! <sniff> ");
    } else
        System.out.println(name + ": Ha! Can't puncture me!");
}
and a simple driver

class Test10 {
    public static void main (String[] args) {
        final int SOFT = 20;
        final int HARD = 100;

        Warrior Gardoz = new Warrior(SOFT, "Gardoz");
        Warrior Zormox = new Warrior(HARD, "Zormox");
        Gardoz.stab(Zormox);
        Zormox.stab(Gardoz);
    }
}
>>java Test
Gardoiz created with:
shield: Hardness = 20
sword: Hardness = 20 Length = 2
armor: Hardness = 20
Zormox created with:
shield: Hardness = 100
sword: Hardness = 100 Length = 2
armor: Hardness = 100
Zormox: Ha! Can't puncture me!
Gardoiz: stabbed by a 2-foot long sword!
Gardoiz: stabbed by a 2-foot long sword!
Gardoiz: farewell! <sniff>