designing from scratch

• “Code-and-fix" aka "Hacking"
  1. Program the thing
  2. Design it (in that order)
• versus:
  1. Design it
  2. Program the thing (in that order)

• even good designs involve some evolutionary change.
• refactoring:
  reorganizing the class hierarchies
  adding new classes
  shifting methods around
  changing arguments (signatures) of methods
  adding new constructors, interfaces, etc.
object-oriented programming entails

1) deciding on the objects of your domain
   - active agents
   - passive agents
   - actions/events
   - locations

2) isolate "use cases", e.g.,
   - Dog d barks at Cat c;
   - c is alarmed and runs;
   - d gets excited and chases c

This use case is deceptively difficult
   - when Dog d barks at Cat c, how does c hear the bark?
   - what causes the cat to decide to run?
   - how does the dog see the cat’s movement?
   - what causes the dog to decide to run?
object-oriented programming entails

3) refine the use case while creating a sequence diagram
   - tentatively assign:
     nouns to classes
     verbs to methods or perhaps other classes.

4) look for symmetries
   - Dog creates a Bark -> Cat observes Bark -> Cat runs
     maybe:
     Cat changes state to alarmed -> Alarmed state -> Run
   - Cat creates a Movement -> Dog observes Movement -> Dog runs
     maybe:
     Dog changes state to aggressive -> Aggressive state -> Run
woof!

- a Dog has a $k\%$ likelihood of barking if it has heard a bark.

```java
import java.util.Random;
Random r = new Random();
if (r.nextFloat() < k)
    bark();
```

- And usually $k$ increases if it has already heard a bark

```java
import java.lang.Math;
if (hearBark()) {
    k = Math.max(1.0f, Dog.EXCITABILITY_FACTOR * k);
    ...
}
```

- but how to have `bark()` in one Dog instance cause `hearBark()` to be true in another?
• how to make the bark() method in a given Dog instance be heard by another Dog? Not a good idea to try:
  
  ```java
  // For all other Dog instances d>
  
  d.listenToMeBark(); // like the ol’ beStabbed() method
  }
  ```

• what mechanism is responsible for a given Dog hearing another Dog?
  
  if (r.nextFloat() < k)
      Neighborhood.propagateSound(new Bark());
  }

• where the Neighborhood takes on some responsibility
  
  if (r.nextFloat() < k) {
      setChanged();
      notifyObservers("woof!");
  }

• where the Dog is responsible, but how does it know who (if anybody) is listening?
public class Dog extends Observable implements Observer {
  – then each Dog instance must add itself as an observer of all other Dogs in the vicinity...
  – that responsibility could be given to the Neighborhood, then
    if (r.nextFloat() < k) {
      setChanged();
      notifyObservers("woof!"); // or "WOOF!" etc.
    }
  – results in calling the update(Observable obs, Object o) method in all other dogs in the Neighborhood
  – update might then do
    if (obs instance of Dog) {
      – here it is essentially "hearing" the bark. Object o contains a String (which could be "woof!" or "WOOF!")
        if ((o instance of String) && (String(o).equals("WOOF!")))
with Dogs responsible for observer/observable

Neighborhood neighborhood = new Neighborhood();
Dog fido = new Dog();

• Neighborhood has a collection (set, arraylist, something such) of dogs. When a new dog enters, it is made a mutual observer and observable of all others already there.

public void enter(Dog newbie) {
    dogs.add(newbie);
    dogs.start();
    while dogs.more() {
        Dog d = (Dog)dogs.get();
        d.addObserver(newbie);
        newbie.addObserver(d);
    }
}
how can Dog “be” an Observable and a Mammal?

• we’ve expected Dog to be a subclass of Mammal (or Pet, or whatever)
  – how can Dog also extend Observable (in Java)?
• perhaps use the Adaptor Pattern? (Chapter 7 of the HFDP text)
  – use composition
  – i.e., the Dog owns a (private?) “inner observable”
  – when the Dog barks, it tells its observable which then notifies observers (such as cats and other dogs)
  – the cat, being an Observer, would get an update, and could then pull from the Dog by querying the Dog’s observable
• if Dog “has a” inner Observable, you need to subclass Observable to provide public access to the setChanged method by the Dog.
• but then, any observer of Dog will be notified by that Observable, not the Dog itself, so perhaps send an instance of the Dog as the data (second argument) to notifyObservers
• or … ???
• I still like having dogs create Barks in a neighborhood, and have the Dogs all observe the Neighborhood...
how can Dog “be” an Observable and a Mammal?

so maybe make a Dog look like an Observable (by implementing a new interface)

```java
public class Dog extends Mammal
    implements ObservableI, ObserverI {
private Observable observable;

    public Dog() { observable = new Observable(); }

    public void addObserver(Observer o) {
        observable.addObserver(o);
    }
    public void setChanged() { observable.setChanged(); }
}
```

but still, how do to this setChanged which is protected?
the Java implementation poses a problem

```java
public Observable() {
    public void addObserver(Observer o) ...
    protected void clearChanged() ...
    public int countObservers() ...
    public void deleteObserver(Observer o) ...
    public boolean hasChanged() ...
    public void notifyObservers() ...
    public void notifyObservers(Object arg) ...
    protected void setChanged() ...
}

public interface Observer() {
    public void update(Observable o, Obj arg)
}
```
public Observable() {
    public void addObserver(Observer o) ...
    protected void clearChanged() ...
    public int countObservers() ...
    public void deleteObserver(Observer o) ...
    public boolean hasChanged() ...
    public void notify Observers() ...
    public void notifyObservers(Object arg) ...
    protected void setChanged() ...
}

public Observable2() extends Observable {
    public void clearChanged2() { clearChanged(); }
    public void setChanged2() { setChanged(); }
}

public interface Observer2() extends Observer {
    public void update(Observable2 o, Obj arg)
}