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?

how to design this so that it extends to other related use cases:
• d barks in an empty room
• d barks at two cats, c1 and c2
• d barks at another Dog (with different outcome), ...
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.
• look for new classes that provide for communication, e.g.,
  – between a dog and a cat, s.t. one sees the other
  – between a dog and a bark, and between a bark and a cat
  – between a dog and a dog
• 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
  – maybe:
  – Run + Run = Chase
a Dog has a k\% (0.0 \leq k \leq 1.0) likelihood of barking if it has heard a bark.

```python
import random

if random.uniform(0, 1) < k:
    bark()
```

And usually k increases if it has already heard a bark.

```python
if hearBark():
    k = min(1.0, EXCITABILITY_FACTOR*k)
```

but how to have `bark()` in one Dog instance cause it in another?

```java
import java.util.Random;
Random r = new Random();
if (r.nextFloat() < k)
    bark();
```
• how to make the bark() method in a given Dog instance be heard by another Dog? Not a good idea to try:

```python
for dog in neighborhoodDogs:
    dog.listenToMeBark()  # like the ol' beStabbed() method
```

• what mechanism is responsible for a given Dog hearing another Dog?

```python
neighborhood.propagateSound(Bark())
```

• where the neighborhood takes on some responsibility

```python
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 a 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

- one model is to have all dogs in a neighborhood mutual observers
  - a neighborhood has a collection of dogs. When a new dog enters, it is made a mutual observer and observable of all others already there.
- give the neighborhood that responsibility of setting up observer
  
  ```python
  neighborhood = Neighborhood()
  fido = Dog()
  
  def enter(newbie):
    for dog in neighborhood:
      dog.addObserver(newbie)
      newbie.addObserver(dog)
  
  should fido observe itself? I think so
  fido.bark(); // first bark to start things going
  
  neighborhood could also have new dog observe and be observed by all other suitable objects in the neighborhood (cats, squirrels, etc.)
how can Dog “be” an Observable and a Mammal?

– Java has its problems since it doesn’t allow multiple inheritance
– one approach: use the Adaptor Pattern
  a wrapper that makes a Dog ‘look like’ an Observable (see HFDP)
  the Adaptor Pattern uses composition (each dog has its inner observable)

```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 the Java Observable class has a protected method setChanged()
```
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 notifyObservers() ...

    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)

  }

class Observable(object):
    def __init__(self):
        super(Observable, self).__init__()
        print "initializing Observable"
        self._observers = []

    def addObserver(self, observer):
        if not observer in self._observers:
            self._observers.append(observer)

    def removeObserver(self, observer):
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notifyObservers(self):
        for observer in self._observers:
            observer.update(self)

class Observer(object):
    def __init__(self):
        super(Observer, self).__init__()
        print "initializing Observer"
    def update(self, observable):
        pass

class Dog(Observable, Observer):
    def __init__(self):
        super(Dog, self).__init__().__init__()
the Strategy Pattern in general
a Strategy Pattern example
an Observer Pattern example

```java
for all o in subscribers {
    o.update(this);
}

void update(NewsPublisher newsPublisher) {
    System.out.println(newsPublisher.getLatestNews());
    ...
}
```
the Observer Pattern UML
the Decorator Pattern in general
a Decorator Pattern example
an ice cream example (Java first, then Python)

- http://javapapers.com/design-patterns/decorator-pattern/
the interface, basic version, and decorator in Java

```java
public interface Icecream {
    public String makeIcecream();
}

public class SimpleIcecream implements Icecream {
    public String makeIcecream() { return "Base Icecream"; }
}

abstract public class IcecreamDecorator implements Icecream {
    protected Icecream specialIcecream;

    public IcecreamDecorator(Icecream specialIcecream) {
        this.specialIcecream = specialIcecream;
    }

    public String makeIcecream() { return specialIcecream.makeIcecream(); }
}
```
public class NuttyDecorator extends IcecreamDecorator {

    public NuttyDecorator(Icecream specialIcecream) {
        super(specialIcecream);
    }

    public String makeIcecream() { return specialIcecream.makeIcecream() + addNuts(); }

    private String addNuts() { return " + cruncy nuts"; }
}

public class HoneyDecorator extends IcecreamDecorator {

    public HoneyDecorator(Icecream specialIcecream) {
        super(specialIcecream);
    }

    public String makeIcecream() { return specialIcecream.makeIcecream() + addHoney(); }

    private String addHoney() { return " + sweet honey"; }
}
and the driver (Java)

```java
public class TestDecorator {

    public static void main(String[] args) {
        Icecream icecream = new HoneyDecorator(new NuttyDecorator(new SimpleIcecream()));
        System.out.println(icecream.makeIcecream());
    }
}
```

- decorating with nuts and honey results in
  - Base Icecream + crunchy nuts + sweet honey
and in Python

```python
import sys

class IceCream(object):
    BASIC_CALORIES = 200

    def __init__(self):
        self.calories = IceCream.BASIC_CALORIES

    def describe(self):
        return "vanilla ice cream"

    def getCalories(self):
        return self.calories

class IceCreamDecorator(IceCream):
    def __init__(self, decorated):
        super(IceCreamDecorator, self).__init__()
        self.decorated = decorated
        self.decorationCalories = 0

    def describe(self):
        return self.decorated.describe() + " with ">

    def getCalories(self):
        return self.decorated.getCalories() + self.decorationCalories

    def removeDecorator(self, decoratorType):
        if isinstance(self, decoratorType):
            return self.decorated
        elif isinstance(self.decorated, IceCreamDecorator):
            self.decorated = self.decorated.removeDecorator(decorationType)
        return self
```
and in Python

class NuttyDecorator(IceCreamDecorator):
    NUTS_CALORIES = 100
    
    def __init__(self, decorated):
        super(NuttyDecorator, self).__init__(decorated)
        self.decorationCalories = NuttyDecorator.NUTS_CALORIES

    def describe(self):
        return super(NuttyDecorator, self).describe() + "Nuts"

class HoneyDecorator(IceCreamDecorator):
    HONEY_CALORIES = 150

    def __init__(self, decorated):
        super(HoneyDecorator, self).__init__(decorated)
        self.decorationCalories = HoneyDecorator.HONEY_CALORIES

    def describe(self):
        return super(HoneyDecorator, self).describe() + "Honey"

class SpamDecorator(IceCreamDecorator):
    SPAM_CALORIES = 20

    def __init__(self, decorated):
        super(SpamDecorator, self).__init__(decorated)
        self.decorationCalories = SpamDecorator.SPAM_CALORIES

    def describe(self):
        return super(SpamDecorator, self).describe() + "Spam"
and in Python

class NuttyDecorator(IceCreamDecorator):
    NUTS_CALORIES = 100

    def __init__(self, decorated):
        super(NuttyDecorator, self).__init__(decorated)
        self.decorationCalories = NuttyDecorator.NUTS_CALORIES

    def describe(self):
        return super(NuttyDecorator, self).describe() + "Nuts"

class HoneyDecorator(IceCreamDecorator):
    HONEY_CALORIES = 150

    def __init__(self, decorated):
        super(HoneyDecorator, self).__init__(decorated)
        self.decorationCalories = HoneyDecorator.HONEY_CALORIES

    def describe(self):
        return super(HoneyDecorator, self).describe() + "Honey"

class SpamDecorator(IceCreamDecorator):
    SPAM_CALORIES = 20

    def __init__(self, decorated):
        super(SpamDecorator, self).__init__(decorated)
        self.decorationCalories = SpamDecorator.SPAM_CALORIES

    def describe(self):
        return super(SpamDecorator, self).describe() + "Spam"
how to remove the nuts and honey, and other decorators?
the Stuffing Pattern