interfaces would require a lot of redundant coding
and putting it all together
The Open-Closed Principle (OCP)

- **Open For Extension**
  - This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

- **Closed for Modification**
  - The source code of such a module is inviolate. No one is allowed to make source code changes to it.

- The abstractions are base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.

- Define a family of algorithms, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from clients that use it.
CREATING A HIERARCHY OUT OF THIN AIR

• Idea:
  two mutual antagonists (Warrior). Their Swords and Armor can have different hardnnesses. The Sword must be made of sufficiently hard metal in order 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:
  case 1: 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)

```java
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 Warrior

```java
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 + " "; }
}
```

CREATING A HIERARCHY OUT OF THIN AIR
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);
        Zormox.stab(Gardoz);
    }
}
CREATING A HIERARCHY OUT OF THIN AIR

>>java Test
Gardoz 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!
Gardoz: stabbed by a 2-foot long sword!
Gardoz: stabbed by a 2-foot long sword!
Gardoz: farewell! <sniff>
SOME REVIEW OF SCOPE OF VARIABLES

• scope of variables ... visibility of a given variable

• variables as pronouns, and what they may refer to
  at compile time
  at run time

• setters and getters, and strict encapsulation of object attributes

• how can a class be deriving abilities (methods) without inheritance?
void Blah {
    int x;

    if ... {
        int x; // this x shadows outer x
        ...
    }
}

void Blah {
    int x;

    if ... {
        x = 0; // this x same as outer x
        ...
    }
}

• resolved by looking outward to increasingly larger contexts
class A {
    public int x = 1;
    public void y() { ...println("A") }
}

class B extends A {
    public int x = 2;
    public void y() { ...println("B") }
}

in constructing a B, think of first constructing an instance of A to lay foundation then extensions of B overlaying any counterparts in A

x and y in B overlay (shadow) original versions in A ... but caution (stay tuned)

incidentally, from a design point of view:
why would you use same name y() in class B?
why would you use same name x in class B?
class A {
    public int x = 1;
    public void y() { ...println("A") }
}

class B extends A {
    public int x = 2;
    public void y() { ...println("B") }
}

A a = new A();
B b = new B();
A c = new B();

a.x == ? what does a.y() print?
b.x == ? what does b.y() print?
c.x == ? what does c.y() print?

a.x == 1  a.y() prints A
b.x == 2  b.y() prints B
c.x == 1  c.y() prints B
class MetalThing {
    public int hardness = 1;

    MetalThing() {}  
    public int getHardness() { return hardness; }
}

class Armor extends MetalThing {
    public int hardness = 2;

    Armor() {}  
    public int getHardness() { return hardness; }
}
class Test {
    public static void main (String[] args) {
        MetalThing a = new MetalThing();
        Armor           b = new Armor();
        MetalThing      c = b;

        System.out.print("a.hardness = " + a.hardness);
        System.out.println("   a.getHardness = " + a.getHardness());
        System.out.print("b.hardness = " + b.hardness);
        System.out.println("   b.getHardness = " + b.getHardness());
        System.out.print("c.hardness = " + c.hardness);
        System.out.println("   c.getHardness = " + c.getHardness());
    }
}

>>java Test
a.hardness = 1 a.getHardness() = 1
b.hardness = 2 b.getHardness() = 2
c.hardness = 1 c.getHardness() = 2
but what about being able to have Sword stab other things
  – a piece of fruit, the ground, empty air ??
    Sword sword = new Sword(...);
    Fruit apple = new Fruit("apple");
    sword.stab(apple);
  – but they are not subclass of Warrior

how to share functionality (such as stab()) across hierarchies?
  – one option: make a new superclass Stabbable that unites them
  – provide versions of stab method specific to subclass
    public void stab(Stabbable victim) {
      victim.beStabbed(this;
    }
  – note that the Strategy Pattern would be applicable here

Java does not allow "multiple inheritance", so can't say:
  public class Warrior extends Animate, Stabbable {
  but Java does allow:
  public class Warrior extends Animate
    implements Stabbable {
INHERITANCE VERSUS (JAVA) INTERFACES

• define an interface that describes what methods need to be implemented to account for the notion

```java
public interface Stabbable {
    abstract public void beStabbed(Sword s);
}
```

and require that any class you want to be able to stab implement that interface.

```java
class Apple implements Stabbable {
    public void beStabbed(Sword s) {
        System.out.println("squirt!");
    }
}
```
public interface Stabbable {
    abstract public void beStabbed(Sword s);
}

- So now recall Warrior already has the method beStabbed
  
  public void beStabbed(Sword s) {
      if ( ... and etc.
  }

- which already happens to satisfy that interface ... so just say:

  class Warrior implements Stabbable {
  ...

- and now, with no further modifications to Warrior, this allows:

  zormox.stab(gardoz);  // “stabbed by a 2-foot long sword!”
zormox.stab(apple);    // “squirt!”
create multiple interfaces, e.g.

```java
interface Movable {
    Point3D getPosition();
    Vector3D getVelocity();
    Vector3D getHeading();
    void setPosition(Point3D point);
    void setVelocity(Vector3D velocity);
    void setHeading(Point3D heading);
}

interface Animate extends Movable
interface Inanimate extends Movable
```

then every sort of Animal can have the properties and attributes of being an Animate

```java
class Animal implements Animate{}
class Vertebrate extends Animal{}
class Mammal extends Vertebrate{}
class Primate extends Mammal{}
class Human extends Primate{}
```
INTERFACES PROVIDE A (COSTLY) SOLUTION

- if we also write
  ```java
  interface ThingWithMass {
      final int PLANKS_CONSTANT ...
      void setMass(int mass);
      void getMass(int mass);
      ... and etc.
  }
  ```

- an Animal can implement various interfaces simultaneously:
  ```java
  class Animal extends Animate implements ThingWithMass {}
  class Vertebrate extends Animal {}
  class Human extends Mammal
      implements largerThanBreadboxThing {}
  ```

- that's a lot of expressive power
- but it requires a lot of work to circumvent the restrictions of single inheritance
  - redundant coding
  - Strategy Pattern and a lot of design
9.5 Inheritance (from http://docs.python.org/release/1.5/tut/node65.html)

The syntax for a derived class definition looks as follows:

```python
class DerivedClassName(BaseClassName):
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. Instead of a base class name, an expression is also allowed.

- Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. If an attribute is not found in the class, recursively search the base class.

Derived classes may override methods of their base classes.

- Since methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it.

- An overriding method in a derived class may extend rather than simply replace the base class method of the same name:

  ```python
  BaseClassName.methodname(self, arguments).
  ```
want a duck that can display the weather? (you know you do)

```python
from cis.things.weatherStation import CurrentConditionsDisplay

class WeatherDuck(Duck, CurrentConditionsDisplay):
    def __init__(self, name):
        Duck.__init__(self, name)
        self.quackBehavior = muteQuack

    def id(self):
        return "I'm a Weather Duck"

    def display(self):
        print self.name + 
        " the WeatherDuck says the temperature is now " + 
        str(self.temperature)
```
• in the driver code:

```python
from cis.things.weatherStation import *
from cis.animates.SimUDuck_v2 import *

weatherData = WeatherData()

display = CurrentConditionsDisplay()
weatherData.addObserver(display)

weatherDuck = WeatherDuck('Walter')
weatherData.addObserver(weatherDuck)

weatherData.setTemperature(78)
weatherData.setTemperature(100)
```

• and the result is just ducky (note both react to each update)

```
CurrentConditionsDisplay update: temperature now 78
Walter the WeatherDuck reporting: the temperature is now 78
CurrentConditionsDisplay update: temperature now 100
Walter the WeatherDuck reporting: the temperature is now 100
```
PYTHON PERMITS MULTIPLE INHERITANCE

• the following is from: http://docs.python.org/release/1.5/tut/node66.html
• in general:

        class DerivedClass(BaseA, BaseB, BaseC):

• the only rule necessary to explain the semantics is the resolution rule used for class attribute references:
  – depth-first, left-to-right.
  – if an attribute is not found in DerivedClass, it is searched for in BaseA, then (recursively) in the base classes of BaseA, and only if it is not found there, it is searched in BaseB, and so on.

• [to some people breadth first – searching BaseB and BaseC before the base classes of BaseA – looks more natural however, this would require you to know whether a particular attribute of BaseA is actually defined in BaseA or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of BaseB.]

• the depth-first rule makes no differences between direct and inherited attributes of BaseA
PYTHON PERMITS MULTIPLE INHERITANCE

• a well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class.
• while it is easy enough to figure out what happens in this case
  – (the instance will have a single copy of `instance variables’ or data attributes used by the common base class)
  – it is not clear that these semantics are in any way useful.
• multiple inheritance initialization using `super` is tricky:
  – https://fuhm.net/super-harmful/