

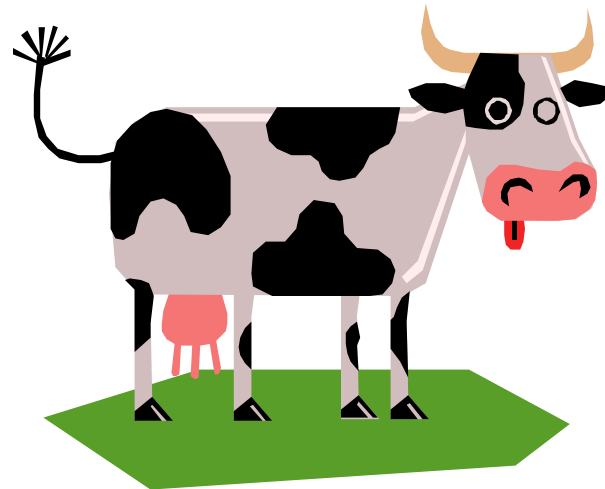
Object Think...

It Really Is Different !

Object Think: It Really Is Different!

Consider the following example:

- Every morning at 4:30 am, Farmer Jones gets up, goes to the barn, and milks his cow.



We all understand this. It makes sense.

**Now, let's look at how we would model this . . .
in a procedural system, and an OO system.**

Milking a Procedural Cow

In the procedural world, we would have a function `Milk()`:

– `float Milk (struct cow, float amount);`

To Milk a cow (i.e., remove milk from it):

```
struct cow
{
    char    name[30];
    float   currentMilkVolume;
    const   float maxMilkVolume = 5.0; //assume gallons
};
```

```
struct Cow  Bessie = {"Bessie", 4.0};
// . . . other code here . . .
// now. . .milk the cow
fReturned = Milk (Bessie, 1.3);
```

Milking a Procedural Cow

Analysis

- The Milk() operation is stand-alone.
 - » **There is no requirement that a Farmer does the milking.**
- Milk() must be told through a *parameter* what Cow to operate on.
 - » **Error-prone**
- Milk() must directly access the struct contents.
 - » **Milk() is, therefore, coupled to the struct.**

Bottom-line:

- In the procedural world, the milk is *taken* directly from the Cow (i.e., a language data structure is directly manipulated).
- There is no concept of a Cow as an *intelligent* entity.

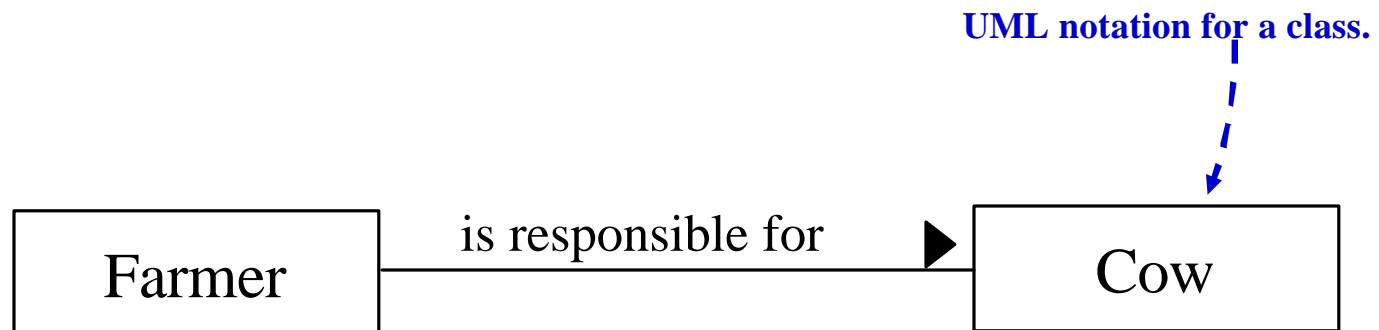
Milking an OO Cow

What entities (called *classes*) are involved?

- A Farmer, and a Cow

What relationships are involved?

- Farmer Jones, a Farmer, *is responsible for* a Cow.



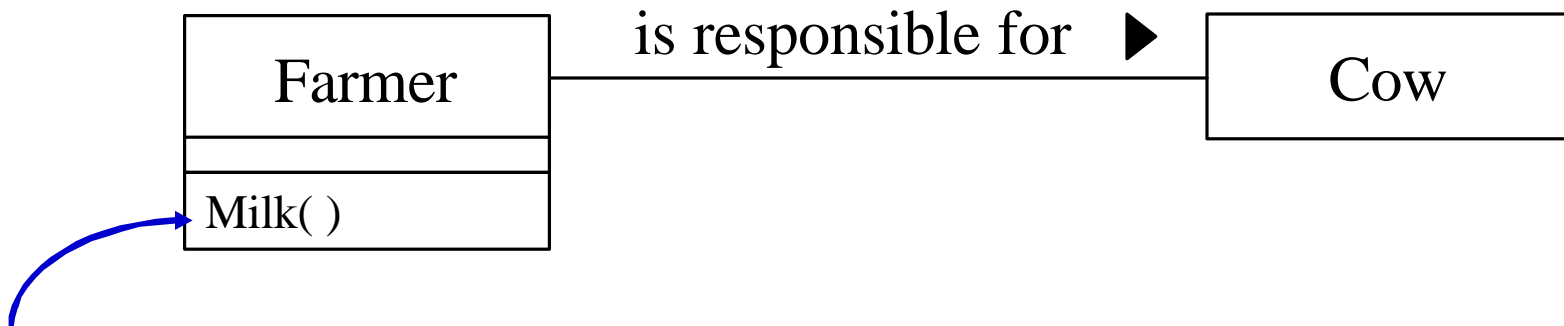
Milking an OO Cow

Now,

- Who owns the `result = Milk(amount)` operation?

In the real world we say “the Farmer milks the cow.”

- So, since the farmer is doing the milking, we put the `Milk()` operation in the Farmer class.



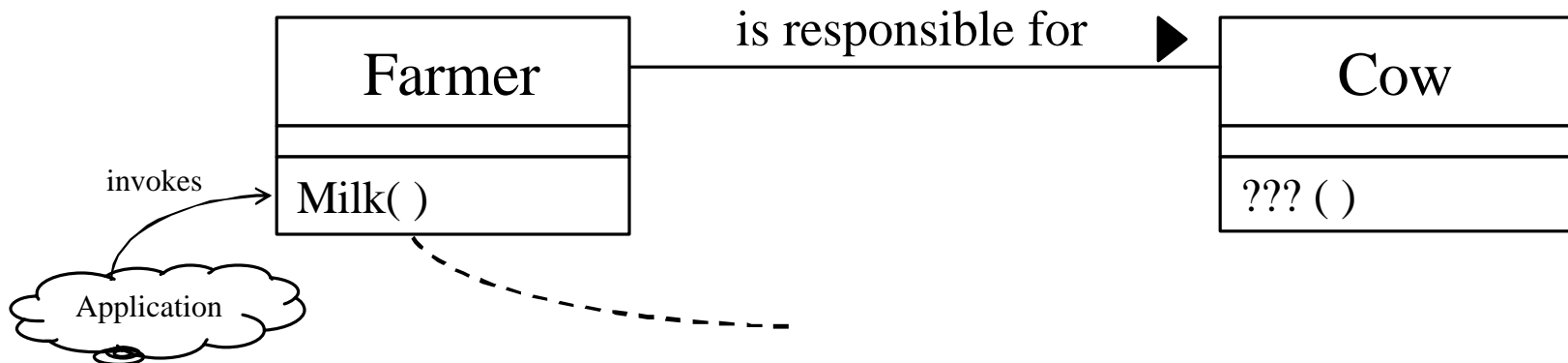
Think “operation” equals “function”

Milking an OO Cow

But does this make sense?

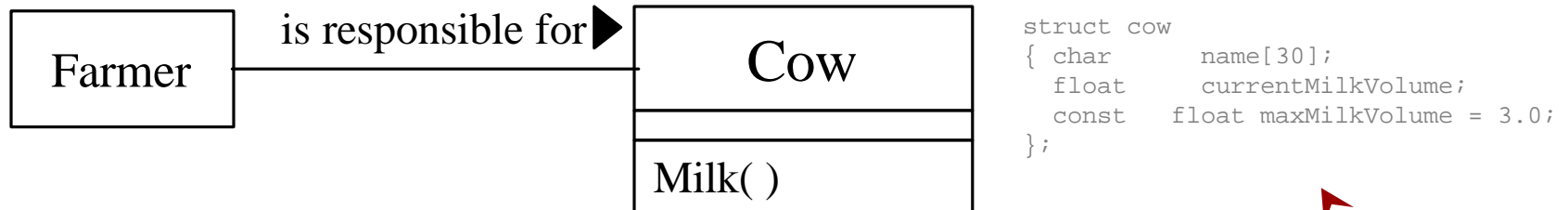
If an application invokes `Farmer.Milk()`, this operation cannot milk the cow, because it is a Farmer operation, and *the milk is in the cow, not the farmer!*

`Farmer.Milk()` still has to access a Cow-scope operation.



Milking an OO Cow

A different approach—let’s put Milk() in Cow.



Now the Farmer invokes the Cow.Milk() operation in order to obtain the milk.

All the “struct” information (the data) is hidden inside Cow.

So, in the OO model the Farmer doesn’t take milk from the cow.

The Farmer asks the Cow to “milk itself”.

- That is, the Cow manipulates its own data values.

Milking an OO Cow

Analysis

- The Milk operation is part of Cow.
 - » **No Cow \Rightarrow no Milk() operation.**
- Milk() does not have to be told what Cow to operate on.
 - » **No possibility of “cross-pollination”.**
- Milk() is an opaque interface, and hides the implementation of Cow from Farmer.

Bottom-line:

- In the OO world, the milk is *requested* from the Cow, and the Cow determines how much to yield—maybe none!

Think About This....

Who should know how much milk the Cow has?

- The cow.

Who should know how to determine whether the Cow is even milkable at the time of the request?

- The cow.

If the Farmer asks for 3 gallons, and the Cow only has 2 gallons to offer, who should determine if 2 gallons, 1 gallon, or 0 gallons, is returned?

- The cow.

And the clincher....

Who's state might change with the invocation of the Milk() operation?

The Cow's state changes, not the Farmer's.

- The Cow may give up the 2 gallons she has, and then transition to the *unmilkable* state until a new store of milk is generated.
 - » **Hmmm, so we should add a Replish(amount) operation to Cow!**
- The Farmer has a bucket of milk that he didn't have before, but his state has not changed.

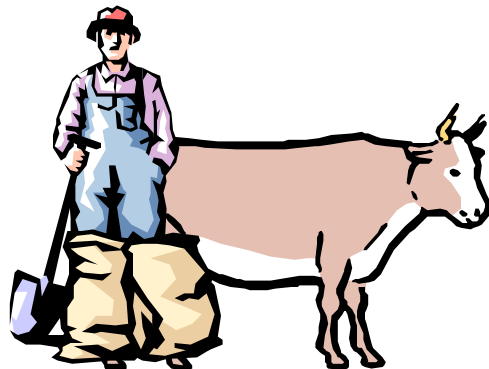
Benefits of the OO Model

What if there is a change to the way the cow determines how much milk to yield?

- Changes to implementation are hidden behind the public interface.
- If the interface is not disturbed, the Farmer is unaware of any change.

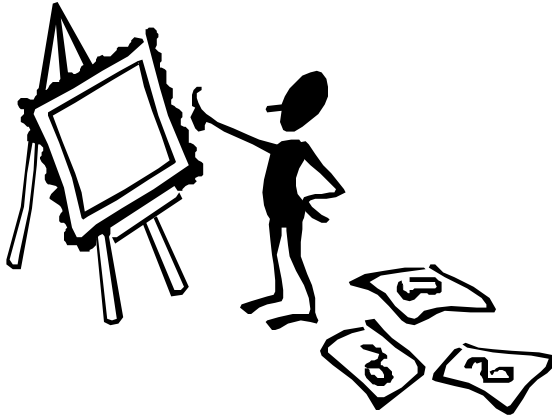
What if another farmer has to milk the cow?

- As long as this farmer observes the cow's interface, he, too can milk the cow.
- In the procedural world, the 2nd farmer...and the 3rd...would have to contain code to do the milking.
- In the OO world, the milking code is in the cow...in one place for reuse.



So, OO Thinking Really is Different!

The point in all of this is:



- **Don't get too distracted by reality.**
- A literal translation of the way the real world works, and the way we talk about it working, are often an impediment to software modeling and implementation.