Object Oriented Analysis and Design

CS-291

Multiple inheritance

- A class can inherit from more than one base class

  => bringing in the members and methods of two or more classes
- Use multiple inheritance judiciously
- Many problems initially solved with multiple inheritance are today solved using aggregation.
- Two problems present themselves when we have multiple inheritance:
  - Name collisions
  - Repeated inheritance

Name collisions

- When two or more different super-classes use the same name for some of their instance variables or methods.

Approaches to resolve

- Language semantics might regard such a clash as illegal and reject the compilation of the class.
- Language semantics might regard the same name introduced by different classes as referring to the same attribute.
- Language semantics might permit the clash but require that all references to the name fully qualify the source of its declaration.

Repeated inheritance

- A class is an ancestor of another in more than one way.
- Meyer – "If you allow multiple inheritance into a language, then sooner or later someone is going to write a class D with two parents B and C, each of which has a class A as a parent— D inherits twice (or more) from A."

Solutions

- Treat occurrences of repeated inheritance as illegal.
- Permit duplication of super-classes but require the use of fully qualified names to refer to members of a specific copy.
- Treat multiple references to the same class as denoting the same class.

```
class base {
    public:
        int i; };

class derived1 : public base {
    public:
        int j; };

class derived2 : public base {
    public:
        int k; };

class derived3 : public derived1, public
derived2 {
    public:
        int sum;
};
```

```
int main()
{
    derived3 ob;
    ob.derived1::i = 10; // ambiguous
    ob.j = 20;
    ob.k = 30;
    // i ambiguous here, too
    ob.sum = ob.i + ob.j + ob.k;
    // also ambiguous, which i?
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

Polymorphism

- *Polymorphism* is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation.
- **Reduces complexity**
- A dog's sense of smell
  - If the dog smells a cat, it will bark and run after it
  - If the dog smells its food, it will salivate and run to its bowl.
- The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose!

- ❑ One way that C++ achieves polymorphism is through the use of function overloading
- ❑ Two or more functions can share the same name as long as their parameter declarations are different.

```
// abs is overloaded two ways
int abs(int i);
double abs(double d);
  int main(){
     cout << abs(-10) << "\n";
     cout << abs(-11.0) << "\n";
  }
int abs(int i){
  cout << "Using integer abs()\n";
  return i<0 ? -i : i;
}
double abs(double d){
  cout << "Using double abs()\n";
  return d<0.0 ? -d : d;
}
```

❑ In C Language?

- Another way to implement the "one interface, multiple methods" approach in C++ is to use inheritance, virtual functions, abstract classes, and run-time polymorphism.

- A virtual function is a member function that is declared within a base class and redefined by a derived class

- When accessed "normally," virtual functions behave just like any other type of class member function

- Base pointer points to a derived object that contains a virtual function

  => C++ determines the version of the function to be called based upon *the type of object pointed to by the pointer* **at run time**

```
class base {
public:
   virtual void vfunc() {
   cout << "This is base's vfunc().\n";}
};
class derived1 : public base {
public:
   void vfunc() {
   cout << "derived1's func().\n";}
};
class derived2 : public base {
public:
   void vfunc() {
   cout << "derived2's func().\n";}
};
```

```
int main()
{
base *p, b;
derived1 d1;
derived2 d2;

p = &b;
p->vfunc(); // access base's vfunc()

p = &d1;
p->vfunc(); // access derived1's vfunc()

p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

```
void f(base &r) {
  r.vfunc();
}
int main()
{
  base b;
  derived1 d1;
  derived2 d2;
  f(b); // pass a base object to f()
  f(d1); // pass a derived1 object to f()
  f(d2); // pass a derived2 object to f()
  return 0;
}
```

Virtual function & Inheritance

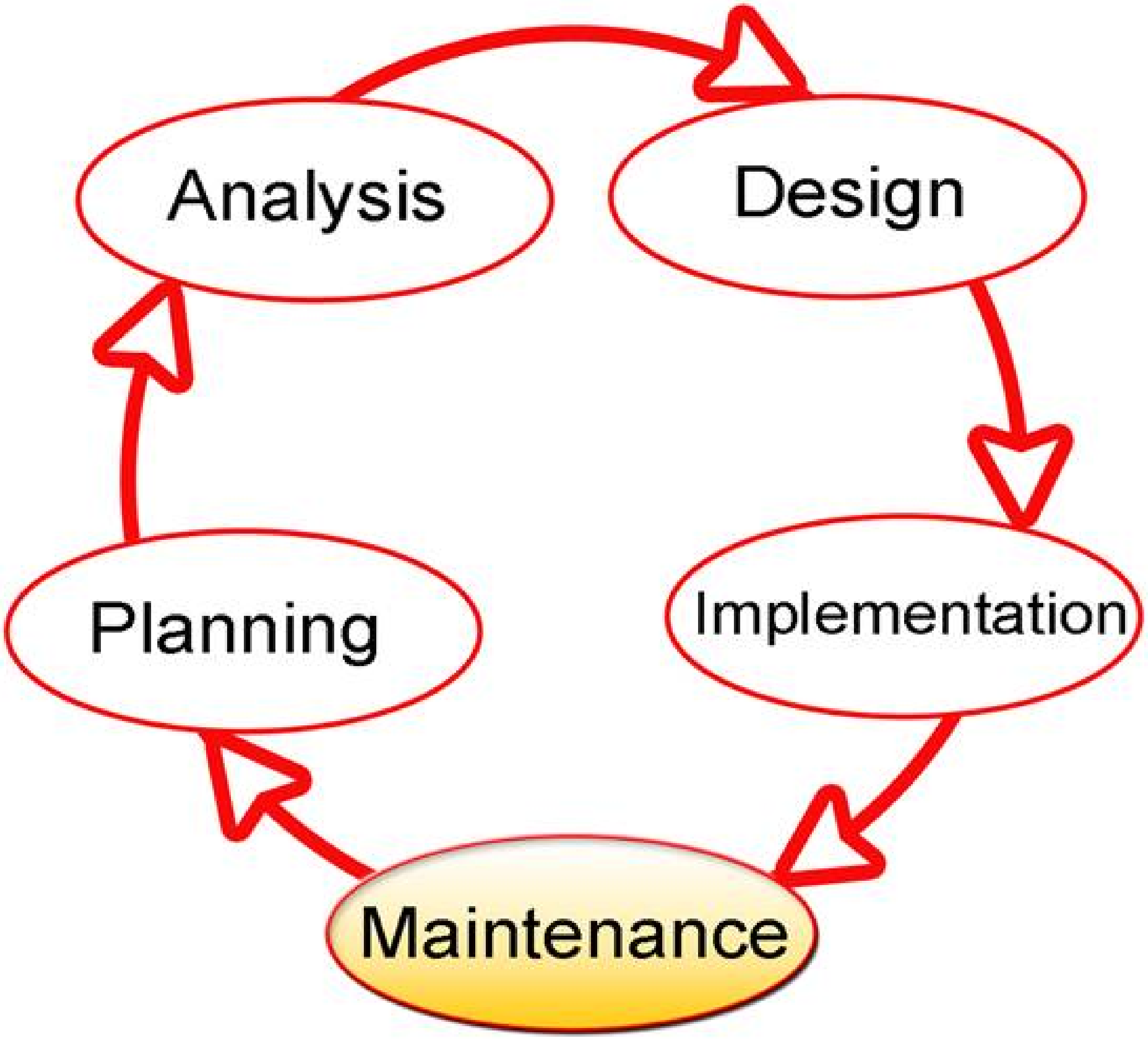
- When a virtual function is inherited, its virtual nature is also inherited
- Now assume that class derived2 in previous example has been derived from class derived1
- What if a derived2 fails to override a vfunc()?

```
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;

    p = &b;
    p->vfunc();

    p = &d1;
    p->vfunc();

    p = &d2;
    p->vfunc();
    return 0;
}
```

```
This is base's vfunc()
This is derived1's
vfunc()
This is derived1's
vfunc()
```

- The first redefinition found in reverse order of derivation is used

System Development lifecycle (SDLC)



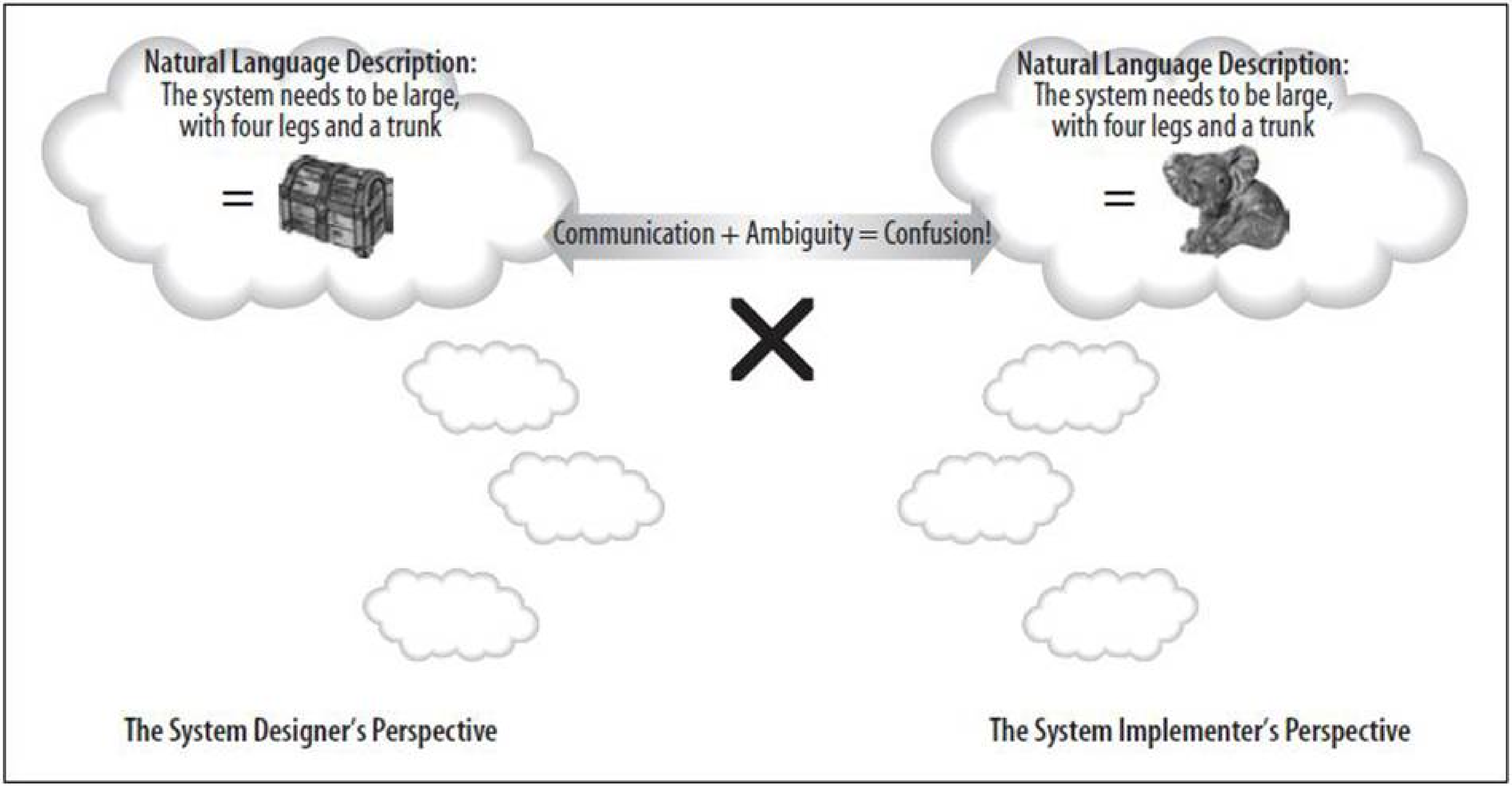
- ❑ **A Model** is a meaningful abstraction of something
- ❑ It is created for the purpose of understanding something before building it
- ❑ **A modeling language** is nothing more than a convention for how we'll *draw our model on paper*

Why A Modeling Language?

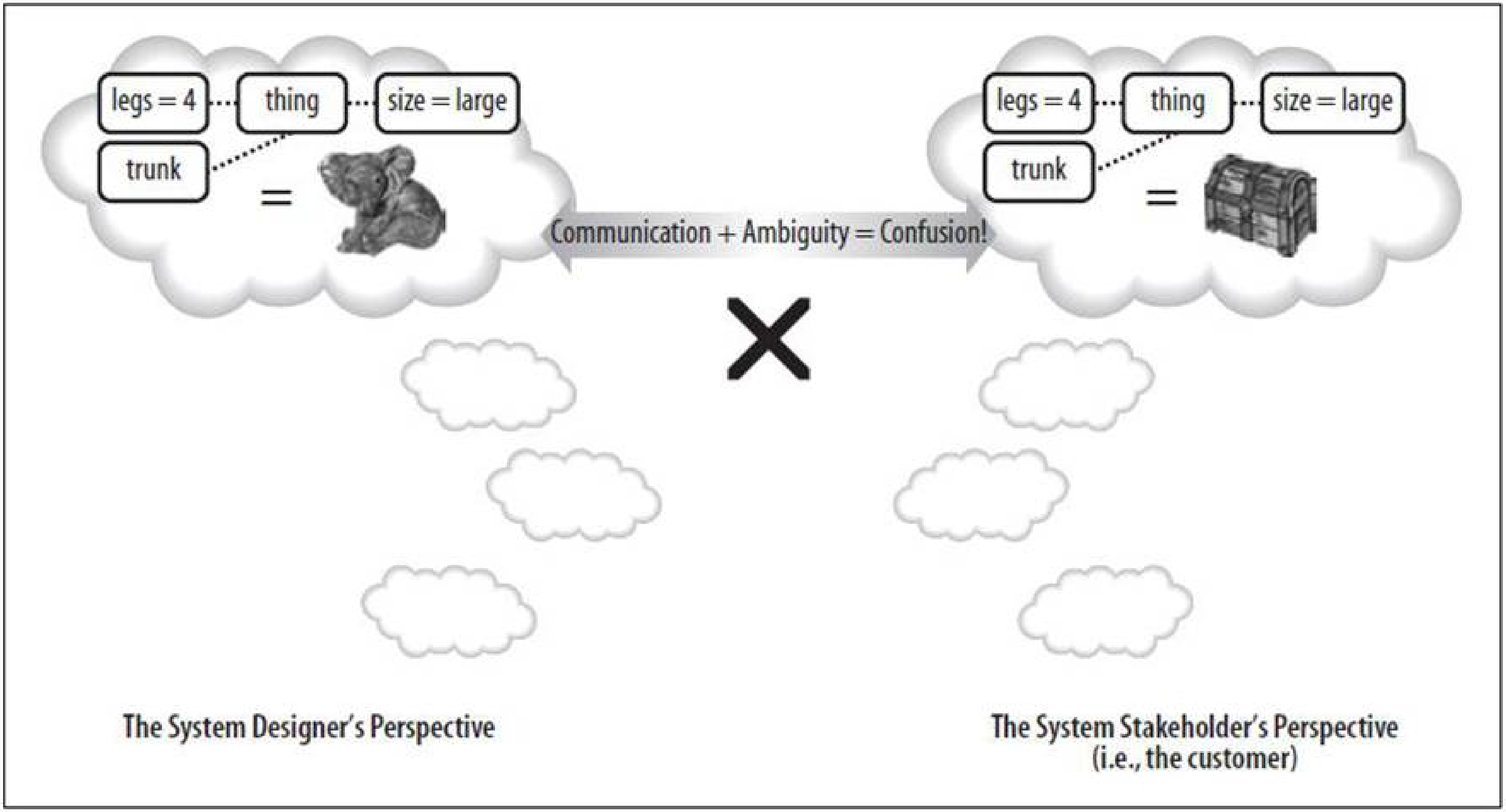
- Modeling with Code
  - Unambiguous definition of what the software will do
  - All details about the objects and their relationships
  - Has meaning to the compiler
  - With proper comments accurately represents the software system
  - Cannot tell you how the software is to be used and by whom, nor how it is to be deployed

The bigger picture is missing entirely if all you have is the source code

Modeling with Natural Language


Natural Language Description:
The system needs to be large, with four legs and a trunk
=
Natural Language Description:
The system needs to be large, with four legs and a trunk
=
Communication + Ambiguity = Confusion!
The System Designer's Perspective
The System Implementer's Perspective

Ambiguity in informal notation


legs = 4
thing
size = large
trunk
=
legs = 4
thing
size = large
trunk
=
Communication + Ambiguity = Confusion!
The System Designer's Perspective
The System Stakeholder's Perspective
(i.e., the customer)

The basic problem with informal languages is that they don't have exact rules for their notation

- ❑ We can decide that we'll draw our classes as triangles, and that we'll draw the inheritance relationship as a dotted line.
- ❑ We'll need to explain our conventions to everyone else with whom we work, and
- ❑ Each new employee or collaborator will have to learn our convention
- ❑ It would be more convenient if everyone in the industry agreed on a common modeling language.