Modularity and Object-Oriented Programming — OO

Steven Zeil

Nov. 4, 2003

Contents

---

1. OOP Philosphy
2. OOP Languages
3. Working With Inheritance
4. Some OOP Languages
5. Implementing OO

---

1 OOP Philosphy

OOP stems from work in Simula, a simulation language.

> Every program is a model of some part of the world. The quality of a program design is directly proportional to how faithfully it reflects the structure of the world that it models.

1.1 Object-Oriented Programs

An OOP is viewed as

- a set of **objects**
- interacting by passing **messages** to one another
- and responding to messages by customized **methods**

1.2 Are objects new?

In some ways, these are familiar programming concepts already:

OOPL	traditional PL
object	variable , constant
identity	label, name, address
state	value
class	type
message	funct decl
method	funct body
passing a message	funct call

But the renaming encourages us to think in terms of simulation:

- Choosing names appropriate to the application domain
- Constant communications with domain experts

---

2 OOP Languages

Essential characteristics of OOPL's:

1. Inheritance
2. Subtyping

3. Dynamic Binding

2.1 Inheritance

A class `C` **inherits** from class `D` if `C` has all the data members and messages of `D`.

- `C` may have additional data members and messages not present in `D`.
- `D` is called a base class of `C`.

Ada has a very pure form of inheritance. Given

```
package Ovens is
  type Oven is private;
  function temp(o: Oven) returns real;
  procedure setTemp (o: in out Oven,
                     desired: real);
private
  type Oven is record
    setting: real;
    temp: real;
  end record;
end Ovens;
```

We can then write

```
with Ovens; use Ovens;
package MoreOvens is
  type MicrowaveOven is new Oven;
  procedure setTimer (o: in out Oven,
                      desired: real);
end MoreOvens;
```

`MicrowaveOven` inherits the data and functions of `Oven`. It adds a new procedure, `setTimer`.

```
procedure bakeBread (o: in out Oven);
o: Oven;
m: MicrowaveOven;

setTemp(o,v); -- OK
setTemp(m,v); -- OK (inh)
setTimer(o,t); -- illegal
setTimer(m,t); -- OK
bakeBread(o); -- OK
bakeBread(m); -- illegal
```

2.2 Subtyping

A type `C` is a **subtype** of `D` if a value of type `C` may be used in any operation that expects a value of type `D`.

- `C` is called a subclass or subtype of `D`.
- `D` is called a superclass or supertype of `C`.

As a general rule, OOPL's combine inheritance and subtyping.

```
class Oven {
public:
  double temp() const;
  void setTemp (desired: real);
private:
  double setting;
  double temperature;
};
```

C++/Java combine inheritance and subtyping.

```
class MicrowaveOven: public Oven
{
public:
  void setTimer (double);
private:
  double time;
};
```

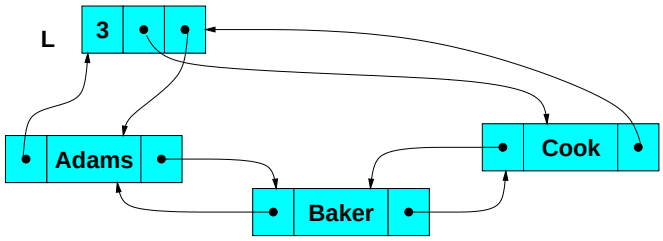
```
void bakeBread (const Oven &);

Oven o;
MicrowaveOven m;

o.setTemp(v); -- OK
m.setTemp(v); -- OK (inh)
o.setTimer(t); -- illegal
m.setTimer(t); -- OK
bakeBread(o); -- OK
bakeBread(m); -- OK (sub)
```

2.2.1 Subtyping: Linked List

As an example of subtyping in action, consider a doubly linked list. It is convenient for many reasons to organize the list as a pair of rings, linked at the beginning and end via a header node.



We might first try:

```
struct Node {
  string data;
  Node *prev;
  Node* next;
};
```

```
struct Header {
  int listSize;
  Node* last;
  Node* first;
};
```

But sometimes a `Node` must point to a `Header` rather than another `Node`.

An elegant solution is offered by subtyping.

- Suppose that `Node` and `Header` were each subtypes of another type, `DLLBase`.
- Then by the subtyping rule, a `Node*` or `Header*` could be substituted anywhere a `DLLBase*` was expected

---

```
struct DLLBase {};

struct Node: public DLLBase {
  string data;
  DLLBase *prev;
  DLLBase* next;
};

struct Header: public DLLBase {
  int listSize;
  DLLBase* last;
  DLLBase* first;
};
```

Still not ideal, because code like

```
DLLBase* p = header->first;
while (p != header)
  {
    process (p->data);
    p = p->next;
  }
```

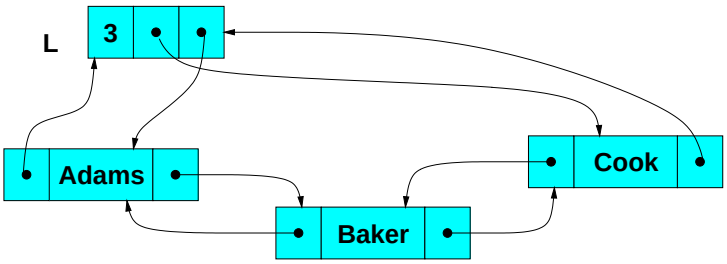
fails because a `DLLBase` does not have `next` or `data` fields.

We can rewrite using casts:

```
DLLBase* p = header->first;
while (p != header)
  {
    process (((Node*)p)->data);
    p = ((Node*)p)->next;
  }
```

---

or we can recognize that everything in this list has two pointers.



```
struct DLLBase {
  DLLBase *backward;
  DLLBase *forward;
};


struct Node : public DLLBase {
  string data;
};


struct Header : public DLLBase {
  int listSize;
};
```

```
DLLBase* p = header->forward;
while (p != header)
  {
    process (((Node*)p)->data);
    p = p->forward;
  }
```

Although we still have a type cast, this is relatively insignificant.

If I package up my `DLList` as an ADT, I will have something like:

```
class StringList {
private:
  Header* header;
public:
  typedef DLLBase* Position;

  StringList();
  ~StringList();

  Position begin() const {return header->next();}
  Position end() const   {return header;}

  Position next(Position p) const  {return p->next;}
  Position prev(Position p) const  {return p->prev;}
   /* ... insert, erase, find ... */
  string& at(Position p)  {return ((Node*)p)->data;}
};
```

2.3 Dynamic Binding

Consider the binding of function body (method) to a function call.

Given the following:

```
a = foo(b);
```

when is the decision made as to what code will be executed for this call to foo?

2.3.1 Static Binding

```
a = foo(b);
```

In a traditional, compiled imperative language (FORTRAN, PASCAL, C, etc.), the decision is made at compile-time.

- Decision is immutable
- If this statement is inside a loop, same code will be invoked for foo each time.
- Compile-time binding — low execution-time overhead.

2.3.2 Unconstrained Dynamic Binding

```
a = foo(b);
```

In a traditionally interpreted language (LISP, BASIC, etc), the decision is made at run-time.

- Decision is often mutable
- If this statement is inside a loop, different code may be invoked for foo each time.
- Run-time binding — high execution-time overhead.

2.3.3 OOP Dynamic Binding

OOPLs typically feature an "intermediate" form of dynamic binding in which the choice of method is made from a relatively small list of options.

OO Dynamic Binding

- list is determined at compile time
- final choice made at run-time
- options are organized according to the inheritance hierarchy
- determined by the actual type of the object whose member function has been selected

---

- In SmallTalk & Java, all function calls are resolved by dynamic binding.
- In C++, we can choose between compile-time and dynamic binding.

2.3.4 Dynamic Binding in C++

- A non-inherited function member is subject to dynamic binding if its declaration is preceded by the word "`virtual`".

- An inherited function member is subject to dynamic binding if that member in the base class is subject to dynamic binding.
  - Using the word "`virtual`" in subclasses is optional (but recommended).
- Even if a member function is virtual, calls to that member might not be subject to dynamic binding.

Dynamically Bound Calls

Let `foo` be a virtual function member.

- `x.foo()`, where `x` is an object, is bound at compile time (static)
- `x.foo()`, where `x` is a reference to an object, is bound at run-time (dynamic).
- `x->foo()`, where `x` is a pointer, is bound at run-time (dynamic).

```
class Animal {
public:
  virtual String eats() {return "???";}
  String name() {return "Animal";}
};

class Herbivore: public Animal {
public:
   virtual String eats() {return "plants";}
   String name() {return "Herbivore";}
};
```

```
class Ruminants: public Herbivore {
public:
   virtual String eats() {return "grass";}
   String name() {return "Ruminant";}
};

class Carnivore: public Animal {
public:
   virtual String eats() {return "meat";}
   String name() {return "Carnivore";}
};
```

```
void show (String s1, String s2) {
   cout << s1 << " " << s2 << endl;
}
```

```
Animal a, *pa, *pah, *par;
Herbivore h, *ph;
Ruminant r;
pa = &a; ph = &h; pah = &h; par = &r;

show(a.name(), a.eats());
show(pa->name(), pa->eats());
show(h.name(), h.eats());
show(ph->name(), ph->eats());
show(pah->name(), pah->eats());
show(par->name(), par->eats());
```

3 Working With Inheritance

Example: Shapes

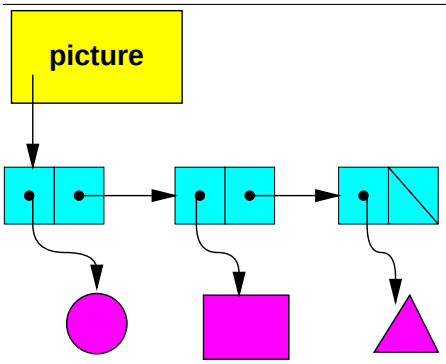
- A `Picture` contains a list of `Shape`s and information on the size of the picture.
- Every `Shape` has a center point and a bounding rectangle.

First, some utility classes:

```
struct Point {
  double x, y;
};

struct Rect {
  Point ul, lr;
  void merge (const Rect& r)
    {
      ul.x = min(ul.x, r.ul.x);
      ul.y = min(ul.y, r.ul.y);
      lr.x = max(lr.x, r.lr.x);
      lr.y = max(lr.y, r.lr.y);
    }
};
```

```
class Shape {
public:
  Shape();
  virtual ~Shape ();
  virtual void draw() const;
  virtual void zoom (double factor);
  virtual const
     Point center() const;
  virtual const Rect bound() const;
};
```




```
struct ShapeList {
   Shape* shape;
   ShapeList* next;
};
```

```
class Picture {
public:
```

```
    Picture();
    virtual ~Picture();

    void clear();
    void add(const Shape&);

    Rect bound() const;
    void draw() const;

    void zoom(double factor);
private:
     ShapeList *shapes;
};
```

```
class Circle: public Shape {
    Point center;
    double radius;
public:
    Circle(Point cent, double r);
    Circle(const Circle&);
    ~Circle();

    void draw() const;
    void zoom(double factor);
    Rect bound() const;
};
```

```
class Line: public Shape {
    Point ul, lr;
public:
    Line(Point p1, double p2);
    Line(const Line&);
    ~Line();

    void draw() const;
    void zoom(double factor);
    Rect bound() const;
};
```

Drawing a picture:

```
void Picture::draw() const
{
  ShapeList* s = shapes;
  while (s != 0) {
    s->shape->draw();
    s = s->next;
  }
}
```

Relies on dynamic binding of `draw()`.

Without dynamic binding, this would have been written as

```
void Picture::draw() const
{
  ShapeList* s = shapes;
  while (s != 0) {
    switch (s->tag) {
      case lineTag:
        drawLine(*s); break;
      case circleTag:
        drawCircle(*s); break;
        ⋮
    }
    s = s->next;
  }
}
```

Moving to OOP

A key marker in OOP style is the replacement of

- unions (variant records)
- multi-way branches

by dynamic binding over an inheritance hierarchy.

Let's look at what's involved in computing the size of a picture:

```
Rect Circle::bound() {
  Rect b;
  b.ul.x = center.x − radius;
  b.ul.y = center.y − radius;
  b.lr.x = center.x + radius;
  b.lr.y = center.y + radius;
  return b;
}
```

```
Rect Line::bound() {
  Rect b;
  b.ul = ul;
  b.lr = lr;
  return b;
}
```

```
Rect Picture::bound() const
{
  Shape* s = shapes;
  if (s != 0) {
```

```
    Rect r = s->shape->bound();
    s = s->next;
    while (s != 0) {
      r.merge(s->shape->bound());
      s = s->next;
    }
}
```

Relies on dynamic binding of `bound()`.

What's involved in adding a new shape (e.g., ellipse)?

- In traditional languages, design new `Ellipse` type.
  - Must supply ellipse-specific code for `drawEllipse()`, `boundEllipse()`, etc.
- ...

- ...
- Find all code everywhere (`Picture` plus applications) that switches based on tags.
  - add a new case to that code

What's involved in adding a new shape (e.g., ellipse)?

- In OOPL, design new `Ellipse` class.
  - Subclass of `Shape`
  - Must supply ellipse-specific code for `draw()`, `bound()`, etc.
- No changes to `Picture` required

---

4 Some OOP Languages

- OOP in C++
- OOP in Java
- OOP in Smalltalk
- OOP in Modula 3

4.1 OOP in C++

We've seen the basics already.

- C++ is a hybrid language.
  - Not all types are classes.
  - Not all class member functions dynamically bound.
  - Inheritance hierarchies tend to be small and special-purpose.

4.1.1 Abstract Classes

Sometimes we present class interfaces that are intended only as interfaces — no implementation is expected.

```
class Shape {
public:
  Shape();
  virtual ~Shape();
  virtual void draw() const;
  virtual void zoom(double factor);
  virtual const
     Point center() const;
  virtual const Rect bound() const;
};
```

- Can't really provide meaningful code for `draw()`, `bound()`, etc.
- Rather than provide bogus code for the operations, we mark them as **abstract**.
  - No bodies provided for abstract functions
  - Abstract classes cannot be instantiated as an actual object.
  - The implementation is done in a subclass.

```
class Shape {
public:
  Shape();
  virtual ~Shape();
  virtual void draw() const = 0;
  virtual void zoom(double factor) = 0;
  virtual const
     Point center() const = 0;
  virtual const Rect bound() const = 0;
};
```

4.2 OOP in Java

Another hybrid, but purer than C++.

- Not all types are classes.
- All class member functions are dynamically bound.
- All classes organized into a single inheritance tree
- Garbage collection
- C++-like syntax

Java implementation of Shapes is very similar to C++:

```
class Point {
  double x, y;
}

class Rect {
  Point ul, lr;
  void merge (Rect r) {
    ul.x = min(ul.x, r.ul.x);
    ul.y = min(ul.y, r.ul.y);
    lr.x = max(lr.x, r.lr.x);
    lr.y = max(lr.y, r.lr.y);
  }
}
```

```
abstract
class Shape {
  abstract void draw();
  abstract void zoom (double factor);
  abstract Point center();
  abstract Rect bound() const;
}
```

Establishes the common interface for all shapes.

```
class ShapeList {
   Shape shape;
   ShapeList next;
};
```

Since all class objects are assigned by reference, no need for explicit pointers.

```
class Picture {
   private ShapeList shapes;

   Picture()  {...}
   void clear() {...}
   void add (const Shape s) {
     ShapeList newNode = new ShapeList;
     newNode.shape = s;
     newNode.next = shapes;
     shapes = newNode;
     }

   Rect bound() {...}
   void draw() {...}

   void zoom (double factor) {...}
};
```

```
class Circle extends Shape {
   private Point center;
   private double radius;

   Circle (Point cent, double r) {...}

   void draw() {...}
   void zoom (double factor) {...}
   Rect bound() {...}
};
```

Drawing a picture:

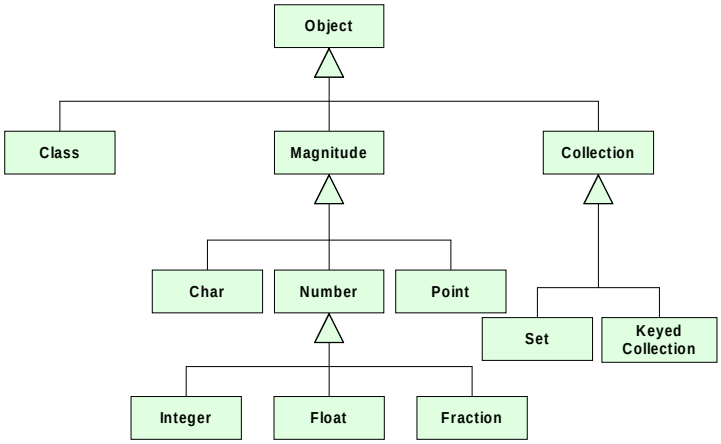
```
class Picture {
  ⋮
  void draw() {
  {
    Shape s = shapes;
    while (s != null) {
      s.shape.draw();
      s = s.next;
    }
  }
}
```

4.3 OOP in Smalltalk

Smalltalk is both a language and a GUI development environment.
In many ways, this is the language that kicked off the OO boom.

- interpreted (usually)
- Everything is an object. (Even classes are objects!)
- All function calls are resolved dynamically.

- All classes are arranged into a single inheritance tree:




```
AShape subclass: #Rectangle
  instanceVariableNames:
                    'theHeight theWidth'
  classVariableNames:

setHeight: anInteger
  "set the height of a rectangle"
  theHeight := anInteger

setWidth: anInteger
  "set the width of a rectangle"
  theWidth _ anInteger
```

```
height
  "return the height of a rectangle"
  ^theHeight

width
  "return the width of a rectangle"
  ^theWidth
```

```
draw
  "draw the rectangle"
  | aLine upperLeftCorner |
  aLine := Line new.
  upperLeftCorner :=
        theCenter x - (theWidth / 2)
    @ (theCenter y - (theHeight / 2)).
```

`ARectangle` inherits a data member `theCenter` from `AShape`.

`theCenter x` means "send the "x" message to the `theCenter` object".

- Note the lack of "syntactic sugar"
  - one seldom really looks at "listings" of an entire class
- Variables are weakly typed.
- Automatic garbage collection

A SmallTalk Browser

Graphics	Array	*i*nsertion	*a*dd:
*C*ollections	Bag	removal	addAll:
Numerics	*F*IFOQueue	testing	
System	Set	printing	

```
add: x
  "Adds x to the end of the queue"
  size := size + 1.
  dataArray at: size put: x.
```

4.4 OOP in Modula 3

In the Pascal/Modula 2 tradition.

- kept Pascal/Modula 2 style syntax and emphasis on simplicity
- adopted C-like (structural equivalance) type system
- Modularity separate from type declaration
- garbage collection by default on all pointers
  - programmer can exempt selected classes from garbage collection (for efficiency)
  - avail of garbage collection simplifies language and code
    - much less emphasis on constructors & destructors

```
INTERFACE Stack;

TYPE T <: REFANY;

PROCEDURE Pop(VAR s: T): REAL;
PROCEDURE Push(VAR s: T; x: REAL);
PROCEDURE Create(): T;

END Stack.
```

- This is a *module* `Stack` that declares a *class* `Stack.T`
- `<:` is the ”inherits from” operator
- The code of the procedure bodies is very Pascal-like.

---

5 Implementing OO

How are inheritance, dynamic dispatching, etc., implemented?

5.1 Implementing Inheritance and Subtyping

Inheritance of data members is achieved by treating all new members as extensions of the base class:

```
struct DLLBase {
  DLLBase *backward;
  DLLBase *forward;
};


struct Node: public DLLBase {
  string data;
};
```

Extension

Inherited members occur at the same byte offset as in the base class:

`p->forward` can translate the same whether `p` points to a `DLLBase` or a `Node`.

Assignment & Subtyping

This implementation explains why, in languages with copy semantics assignments, we can do

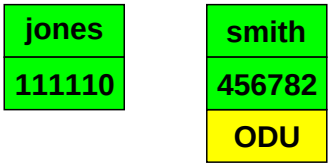
```
superObj = subObj;
```

but not

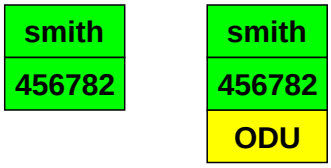
```
subObj = superObj;
```

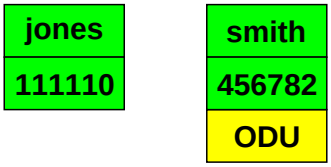
```
class Person {
public:
  string name;
  long id;
};

class Student: public Person {
public:
  string school;
}
```

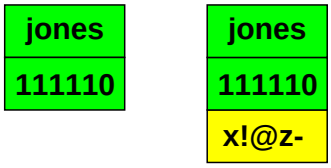



```
jones = smith;
```






```
smith = jones;
```



5.2 Implementing Dynamic Binding

- Single Dispatching
- VTables

Single Dispatching

Almost all OO languages, including C++, offer a **single dispatch** model of message passing:

the dynamic binding is resolved according to the type of the single object to which the message was sent ("dispatched").

Single Dispatching (cont.)

- In C++, this is the object on the left in a call: `obj.foo(...)`
- There are times when this is inappropriate.
  - But it leads to a fast, simple implementation

- Each object with 1 or more virtual functions has a hidden data member.
  - a pointer to a **VTable** for it's class
  - this member is always at a predictable location (e.g., start or end of the object
- Each virtual function in a class is assigned a unique, consecutive index number.

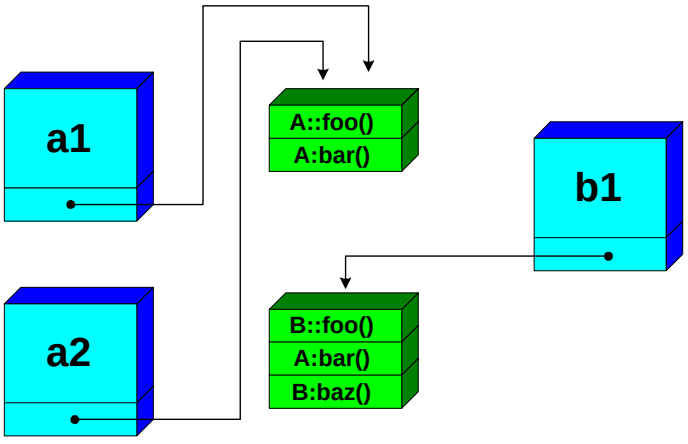
- `(*VTable)[i]` is the address of the class's method for the i'th virtual function.

```
class A {
public:
  A();
  virtual void foo();
  virtual void bar();
};
```

```
class B: public A {
public:
  B();
  virtual void foo();
  virtual void baz();
};
```

```
A* b = (rand() % 2) ? new A: new B;
b->foo();
```

`foo()`, `bar()`, and `baz()` are assigned indices 0, 1, and 2, respectively.



The call `b->foo()` is translated as

```
(*b->VTABLE[0])();
```