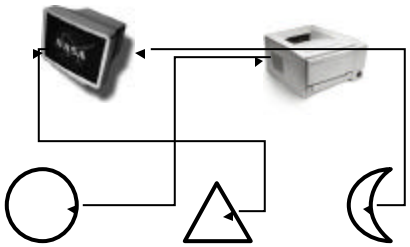
"There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

- C. A. R. Hoare

A Topic in Object-Oriented Programming:

Multiple Dispatch

(or "Calling All Arguments")



Schedule

1. Issues in OOP.
2. Dynamic method binding and virtual method tables.
3. Multiple polymorphism.
4. Discussion of techniques for implementing multiple polymorphism.
5. Wrap up: an implementation in Java.

What do we want in a language?

1. Efficiency.
2. Simplicity.
3. Uniformity.
4. Elegance.
5. Generality.
6. Expressiveness.

Properties of OOP

1. Encapsulation.
2. Inheritance.
3. Polymorphism.

Benefits of OOP

1. Abstraction. Reduces conceptual load.
2. Modularity. Degree of independence among components.
3. Information hiding. Fault containment.
4. Code reuse.

A Bit of Notation

```
displayPort.draw(shape, penMode);
```

- displayPort is the *receiver*.
- draw is the *message*.
- shape and penMode are the *message parameters*.

Static and Dynamic Type

- Principle consequence of inheritance: derived class B has all the members of base class A.
- B can be used whenever A is expected.

Static and Dynamic Type

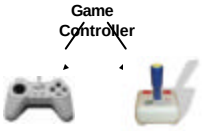
```
class Joystick extends GameController { … }
class GravisPad extends GameController { … }

Joystick j    = new Joystick();
GravisPad g   = new GravisPad();

GameController a   = j;
GameController b   = g;
```

Static and Dynamic Type

- Distinction between static type of variable and dynamic type of object.
- For statically typed languages, variable of type T can have a dynamic type of T or any subtype of T. This is the essence of *polymorphism*.
- Static type is still used to determine legality of operation at compile time.

Dynamic Method Binding

```
Joystick  j = new Joystick();
GravisPad g = new GravisPad();

GameController a = j;
GameController b = g;

j.Move("left");          "Move joystick left"
g.Move("right");         "Move Gravis pad right"
a.Move("up");            "Move joystick up"
b.Move("down");          "Move Gravis pad down"
```



Dynamic Method Binding

- Type of reference: static method binding.
- Type of object: dynamic method binding.
- The latter is more natural.
- Static binding denies control over the consistency of the object's state.

Dynamic Method Binding

```
class GravisPad extends GameController {
   private int autoFire;

   public void Fire() {
      autoFire = 0;
      System.out.println("Fire Gravis pad");
   }
}
```

Dynamic Method Binding

Advantages:

- No need to test for class type with switch/case statements.
- Greater flexibility.
- More reuse of code.
- Add new objects with minimal recoding.

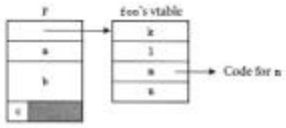
Disadvantages:

- Dynamic method binding is slower.
- OO Languages spend more than 20% of their time calling methods.

Virtual Method Tables

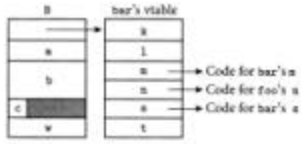
- Every class has a virtual method table

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```



Virtual Method Tables

```
class bar : public foo {
    int w;
public:
    void m ();  //override
    virtual double z ( ...
    virtual char *t ( ...
    ...
} B;
```



b.m();

```
r1 = f
r2 = *r1
r2 = *(r2 + (3 - 1) x 4)
call *r2
```

Virtual Method Tables

- Slow but compact.
- Problem: we have to load the VMT into the cache.

Myth

Dynamic method binding is inefficient.

Importance of Static Type

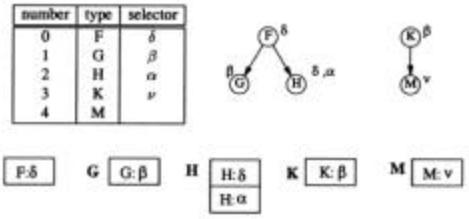
If our language has no static type information…

- No compile-time checks to see if the method is allowed.
- The compiler doesn't know if the method can be called by the object.
- We have to do a linear search through the hierarchy.

Message Dispatch for Dynamically-typed languages

Technique #1: Method Lookup

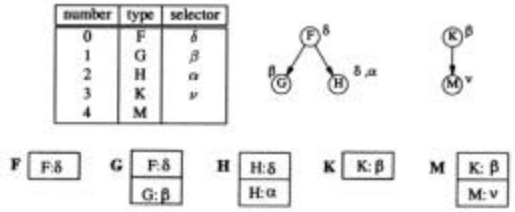
• Each class has a "method dictionary."



Message Dispatch for Dynamically-typed languages

Technique #1: Method Lookup (variation)

• We store the methods of the parents as well



Message Dispatch for Dynamically-typed languages

Technique #2: Global Lookup Cache

• Cache is a set of entries (method, class, function ptr).\

• Whenever a method is dispatched, it is added to the cache.

0	1	2	3
C= nil σ = nil A= nil	C= G σ = δ A= ml	C= nil σ = nil A= nil	C= nil σ = nil A= nil

after dispatching δ0

Message Dispatch for Dynamically-typed languages

Technique #3: Selector Table Indexing

• Type and selector indices are unique.

• If entry is 0, no method exists.

• There are many schemes to compress selector tables. For example, we can remove trailing empty entries.

selectors	index	F	G	H	K	M
δ	0	F:δ	F:δ	H:δ	-	-
β	1	-	G:β	-	K:β	K:β
α	2	-	-	H:α	-	-
ν	3	-	-	-	-	M:ν

Message Dispatch for Dynamically-typed languages with Multiple Inheritance

In order to maintain sanity for the rest of the presentation, I will not discuss this.

R. Dixon. *A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance*. OOPSLA '89 Proceedings.

Method Overloading

• *Multiple polymorphism:* implies all parameters can be used in the selection of a method.

• How method overloading works in Java:

1. Find all the methods that could possibly apply to the function invocation.
2. If only one method matches, invoke it.
3. Otherwise, choose method that is closest.
4. If that fails, invocation is ambiguous!

Method Overloading

```
class BorderCollie extends Dog {
  public void Mate (Dog d) {
    System.out.println("Border collie mating with dog.");  }
  public void Mate (BlackLab d) {
    System.out.println("Border collie mating with black labrador."); }
}

BorderCollie lassie = new BorderCollie();
BlackLab fido = new BlackLab();
Dog dido = fido;

lassie.Mate(fido);
lassie.Mate(dido);
```

Templates

• Barbara Liskov. *Programming with Abstract Data Types.*

```
Stack: cluster(element_type: type)
          is push, pop, top, erasetop, empty;

  rep(type_param: type) = (tp: integer;
                           e_type: type;
                           stk: array[1..]
                                 of type_param;
```

Enter Daniel Ingalls

Enter Daniel Ingalls

His solution:

```
class DisplayPort {
   display (Rectangle o);
   display (Oval o);
   display (Bitmap o);
   …
}
```

```
class PrinterPort {
   display (Rectangle o);
   display (Oval o);
   display (Bitmap o);
   …
}
```

Another Example

+(int, int)
+(int, float)
+(int, complex)
+(int, real)
+(float, complex)
+(float, real)
+(float, float)

Multiple Polymorphism

- What Ingalls proposed was "Double Dispatch."
- We want multiple dispatch… more expressive power!
- Definition of *multi-method dispatch*: one or *more* arguments are used in determining which method to invoke.

$$(o_1, \ldots, o_k).m(o_{k+1}, \ldots, o_n)$$

Multiple Polymorphism

- Conceptually, there are three different kinds of inheritance hierarchies:
  1. State.
  2. Code.
  3. Interface.
- Multi-methods are defined on groups of classes and do not fit the conceptual model of methods being encapsulated within a class.

Motivational Example: The Binary Method

```
class Point {
   int xval, yval;

   public int x( ) { return xval; }
   public int y( ) { return yval; }
   public boolean equal(Point p) {
      return xval == p.x() && yval == p.y();
   }
}

class ColorPoint extends Point {
   int colorval;

   public int color( ) { return colorval; }
   public boolean equal(ColorPoint p) {
      return xval == p.x() && yval == p.y() && colorval == p.color();
   }
}
```

Motivational Example: The Binary Method

```
class Problem {
   Point myPoint;

   public boolean equalToMe (Point p) {
         return p.equal(myPoint);
   }
}
```

Are there other solutions besides multiple polymorphism?

instanceOf and downcasting

```
class ColorPoint extends Point {
   int colorval;

   public int color( ) { return colorval; }
   public boolean equal(ColorPoint p) {
      return xval == p.x() && yval == p.y()
         && (!(p instanceOf ColorPoint)
               || colorval == p.color()));
   }
}
```

Double Dispatch

(what Ingalls proposed)

```
class Point {
   int xval, yval;

   public int x( ) { return xval; }
   public int y( ) { return yval; }
   public boolean equal(Point p) {
      return xval == p.x() && yval == p.y();
   }
    public boolean equal(ColorPoint p) {
         return self.equal((Point)p) && colorval == p.color();
   }
}

etc...
```

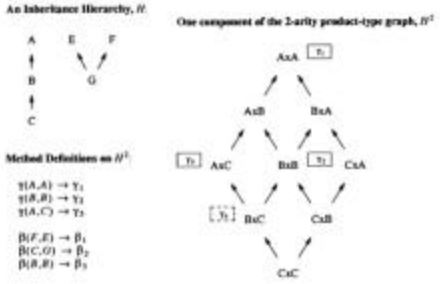
The Multi-Method Solution

```
// Interface.
Boolean equal(Point, Point).equal();

// Implementation.
Boolean equal (Point p1, Point p2) {
    return p1. x( ) == p2. x( ) && p1. y( ) ==a p2. y( );
}

Boolean equal (ColorPoint p1, ColorPoint p2) .equal() {
    return p1. x( ) == p2. x( ) && p1. y( ) == p2. y( )
        && p1.color( ) == p2. color( );
}
```

Formal Multiple Polymorphism

- Method dispatch is the run time determination of a method to invoke.
- The static type of a argument *i* is T^i.
- Dynamic type of an argument is $\{T \mid T < T^i\}$.
- $dom(m_j) = T^1 \times T^2 \times \ldots \times T^k$
- In general, an inheritance conflict occurs at a type *T* if two different methods of a behaviour are visible in supertypes by following different paths up the type hierarchy.

Formal Multiple Polymorphism



Multi-method Dispatch Techniques

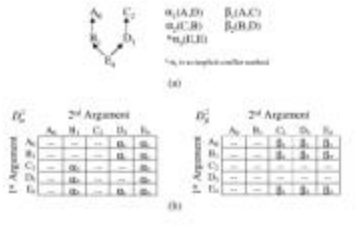
Technique #1: Method lookup

- Generalization of single-method lookup. Every tuple of types has an associated VMT.
- For *k* types, that's n^k different VMTs, where *n* is the number of classes in the hierarchy.
- Not very practical!

Multi-method Dispatch Techniques

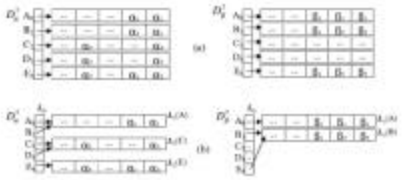
Technique #2: N-Dimensional Dispatch Table

- A *k*-dimensional dispatch table has H^k entries.
- Must resolve conflicts.



Multi-method Dispatch Techniques

Technique #3: Multiple Row Displacement



Discussion

op. \ rep.	Cartesian	Polar	Origin
x	return xval;	return r*cos(a);	return 0;
y	return yval;	return r*sin(a);	return 0;
distanceFrom	...	...	...

Discussion

2nd / 1st	Point	ClrPnt	Pnt3D
Point	(⟨Point, Point⟩)	(⟨Point,ClrPnt⟩)	(⟨Point,Pnt3D⟩)
ClrPnt	(⟨ClrPnt,Point⟩)	(⟨ClrPnt,ClrPnt⟩)	
Pnt3D	(⟨Pnt3D,Point⟩)	?	(⟨Pnt3D,Pnt3D⟩)

Further Reading

This presentation will be available at:
www.cs.ubc.ca/~pcarbo/

Recommended articles

•Wade Holst. *The Tension between Expressive Power and Method-Dispatch-Efficiency in Object-Oriented Languages.*

• Jan Jitek and R. Nigel Horspool. *Taming Message Passing: Efficient Method Look-Up for Dynamically Typed Languages.*