CSE583: Programming Languages

David Notkin
1 February 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Administrivia

- Assignments; reducing from 5 to 4
  - #2 – due 2/11 (on OO)
  - #3 – due 2/25 (on logic/constraint)
  - #4 – due 3/10 (on domain-specific, visual languages,etc.)
- More paper topics listed
- Returning assignment #1
- First term paper back next Tuesday

Lecture schedule

- Tonight and next week (2/1 & 2/8): OO
- Following two weeks (2/15 & 2/22): logic and constraint
- Next to last week (2/29): visual programming, literate programming
- Last week (3/7): domain-specific languages (me or Tom Ball)

Object oriented programming languages

- Basic background and introduction
  - A number of you have surely done more OO programming than me
  - Definitely comment early and often!
- A deeper look at
  - types, multiple inheritance, etc.
- Quick looks at a few classic and interesting languages

A few OO languages

- Simula-67: where it all started
- Smalltalk-80: popularized OO
- C++: OO for the hacking masses
- Java: OO for the web and ???
- CLOS: Powerful OO with Lisp
- Others? Yeah, lots

Object Oriented

- OO programming
- OO design
- OO modeling
- OO analysis
- OO databases
- ...

gOOd's middle name

Dimensions of OO

Primary focus in this course

- Programming language design
  - What features are there, and why?
- Programming language implementation
  - Are these features implemented with sufficient efficiency?
- Software engineering
  - Do these language features help improve software quality or reduce costs?

OO has three key thrusts

- Abstract data types (ADTs)
  - A way to structure programs
- Inheritance
  - A way to exploit the relationships between ADTs
- Dynamic binding
  - Run-time selection of appropriate implementation

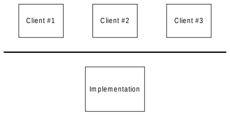
Anything else central to OO?

- Or any of these three that aren't central to OO?

Abstract data types

- An instance of Parnas' information hiding principle
  - How to choose among alternative modularizations
  - Identify aspects of a program that are likely to change, and those that are likely to be stable
  - Capture the stable parts in interfaces, and the likely to change parts in implementations
  - (There's a bit more to it)

Information hiding



- Clients cannot rely on knowledge of the implementation, just it's specification
- The implementation can change without affecting the clients

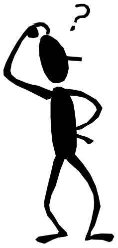
ADTs

- The changeable part of the program is identified to be
  - the representation of the data and
  - the implementation of the operations
- The interface is the stable part
  - The "signature" is the syntactic definition of the interface
  - Semantics are usually given informally

ADTs

- The representation and the operations are packaged together
  - The representation and implementation details are encapsulated and hidden from clients
- An ADT is a kind of module, but one that (usually) allows clients to instantiate multiple instances of the ADT
- Ada packages, Modula modules, etc.

Aside: any weaknesses of information hiding?



- If you took 584 from me, you must remain silent :-)
- Weakness of information hiding fall onto ADTs, too

Classes

- To the first order, an ADT is called a class in an OO language
  - Data structures are called objects and instances
  - Operations are called methods
  - Data inside the class are called instance variables
- (Later, some discussion of class vs. type)

The classic example: a stack

```
class Stack[T] {
  push(item:T):void
  pop():T
  top():T
  size():int
}
```

```
s:Stack[int] :=
    new Stack[int];
s.push(3);
s.push(5);
print(s.pop());
```

Polymorphic

Method

Message send:
"Ask the object to do something"

Instantiation

Two implementations

Instance variable

```
class Stack[T] {
private:
 items:array[10] of T;
public:
 push(item:T):void {
   items[top] := item;
   top := top + 1;
 pop():T {
   top := top - 1;
   return items[top];
 size():int {return top;}
```

```
class Stack[T] {
private:
 items:list[T] := nil;
public:
 push(item:T):void {
   items.add_first(item);
 pop():T {
  return
   items.remove_first();
 size():int {return
  items.length();
}
```

Method implementation

Inheritance

- Define new class as an incremental modification of an existing class
- Perhaps the most recognizable aspect of OO languages and programs
  - ADTs but no inheritance does not usually earn the OO moniker

Inheritance

- New class is *subclass* of the original *superclass*
- By default, subclass inherits the superclass' methods and instance variables
- Can add more methods and instance variables in the subclass
- Can override (replace) methods in the subclass
  - But usually cannot override instance variables

Example

```
class Rectangle {
private:
  center:Point;
  h,w:int;
public:
  area():int
     {return h*w;}
  draw(screen:ODev):void
    {…}
  move(newc:Point):void
    {…}
…
}
```

```
class ColorRectangle
  inherits Rectange {
private:
  color:Color;
public:
  draw(screen:ODev):void
    {…}
}
```

```
r:Rectangle := new
  Rectangle;
cr:ColorRectangle := new
  ColorRectangle;
print.r.area();
print.cr.area();
r.draw();
cr.draw();
```

Benefits of inheritance

- Achieve more code sharing by factoring code into common superclass
  - Encourages development of rich libraries of related data structures
  - Increases reuse
- May model real world scenarios well
  - Use class to model different things
  - Use inheritance for classification of things
    - Subclass is a special case of superclass

Classic hierarchies

- A *square* is-a *rectangle* is-a *polygon* is-a *2D-shape*
- A *domestic cat* (species) is-a *lesser cat* (genus) is-a *cat* (family) is-a *meat-eater* (order) is-a (class) *mammal*
  - mammalia.carnivora.felidae.felis.cattus
  - Herding cats is not for wusses

Rich OO hierarchies

- Smalltalk-80, Java JDK, …

- Magnitude
  - Association
  - Character
  - Date
  - Number
    - Float
    - Fraction
    - Integer
    - LargeNegativeInteger
    - LargePositiveInteger
    - SmallInteger
  - Time

- class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
  - class java.awt.Button
  - class java.awt.Canvas
  - class java.awt.Checkbox (implements java.awt.ItemSelectable)
  - class java.awt.Choice (implements java.awt.ItemSelectable)
  - class java.awt.Container
    - class java.awt.Panel
    - class java.applet.Applet
    - class java.awt.ScrollPane
    - class java.awt.Window
    - class java.awt.Dialog
    - class java.awt.FileDialog

The world is not perfectly hierarchical

- An elephant is a mammal
- An elephant is a gray thing
  - Unless it is albino
- An elephant is a big thing
- An elephant has four legs
  - Unless it lost one
- …leads to issues in multiple inheritance…

Pitfalls of inheritance

- Often overused, especially by novices
- Code gets fragmented into small, factored pieces
- Tracing control logic of code is harder
- Simple extension and overriding may be too limited
  - Ex: exceptions in classification hierarchies

Dynamic binding

- Allow subclass to be used wherever a superclass is expected
  - Allows reuse of superclass' code
- When message is sent, proper operation is located and invoked

```
r:Rectangle := …
cr:ColorRectangle := …

r := cr;
…
r.draw();
```

which `draw` is invoked?

Dynamic binding (more)

- This is a new kind of polymorphism: subtype (inclusion) polymorphism
  - We'll come back to this later
- Dynamic binding requires run-time class information for each object
  - Needed to figure out proper method to invoke
- Also known as message passing, virtual function calling, generic function application

---

Method lookup

- Given a message `obj.msg(args)`
- Start with run-time class C of `obj` (the "receiver")
- If `msg` is defined in `C`, invoke it
- Otherwise, recursively search in the superclass of `C`
- If a match is never found, report run-time error (`"Do not understand"`)
  - In a statically typed OO language, this error will never be reported

---

Example: displaying shapes in list

```
forall s:Shape in scene do
  if s.is_rectangle() then
    rectangle(s).draw();
  elseif s.is_square() then
    square(s).draw();
  elseif…
  else
     error("unknown
 shape");
  fi
end
```

```
forall s:Shape in scene do
  s.draw();
end
```

Add new shapes?

Benefits of dynamic binding

- Allows subtype polymorphism and class-specific methods
- Allows new subclasses to be added without modifying clients
- More important than inheritance?

Pitfalls of dynamic binding

- Makes logic of program harder to follow
- Adds run-time overhead
  - Space for run-time class information
  - Time to do method lookup
    - But only an indirect jump, not a search

Time for questions and comments

- Specific questions about OO: why, what, how?
- Observations from experience about what aspects of OO are most crucial

Types

- Under what conditions are instances of two types the same?
  - Constrains assignment (and related operations) in most languages
- Arises even in “old” imperative languages like Pascal

Name vs. structural equivalence

```
record cartesian {x,y: float};
record polar (r,theta: float};
a,b: cartesian;
c: polar;
…
a := b;
c := b;
a.x := c.theta;
```

Polymorphism

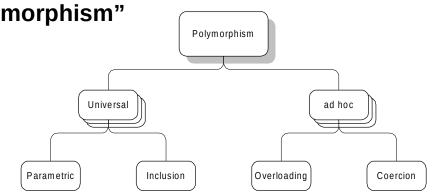
- A walk through some definitions
- Many from the OO FAQ on the web
- It’s more than just definitions
  - At the same time, many of the definitions are definitely tricky (or worse)

Strachey (1967)

- "Parametric [true] polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure
- “Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type”

Cardelli and Wegner (1985)

- Expand on Strachey’s definition by adding “inclusion (or subtype) polymorphism”



Definitions

- Polymorphic Languages:
  - Some values and variables may have more than one type
- Polymorphic Functions:
  - Functions whose operands (actual parameters) can have more than one type
- Polymorphic Types:
  - types whose operations are applicable to operands of more than one type

---

More definitions

- Parametric Polymorphism:
  - A polymorphic function has an implicit or explicit type parameter that determines the type of the argument for each application of that function
    - Ex: A list of ints is not a list of strings, but they are both lists
- Inclusion Polymorphism:
  - An object can be viewed as belonging to many different classes that need not be disjoint; that is, there may be inclusion of classes
    - Ex: a ColorRectangle is also a Rectangle

Universal polymorphism

- Parametric and inclusion are closely related
  - Implementation approaches are distinct, however
- Parametric polymorphism is referred to as generics
  - Each generic instantiation can create a specialized version of the code
    - Ex: STL (standard template library)
- In a "true polymorphic system", only a single implementation is used

---

Inheritance (Cardelli/Wegner)

- Subtyping on record types corresponds to the concept of inheritance (subclass) in languages, especially if records are allowed to have functional components
  - [These functional components in records are methods]

How do we determine if a type `A` is a subtype of a type `B`?

- `A <= B` means `A` is a subtype of `B`
- Consider types as records
  - `A` must have all the fields that `B` has; `A` can have more fields
- For all fields in common, $f_A <= f_B$

Example

```
type object = (age : int)
type vehicle =
   (age : int,
    speed : int)
type machine =
   (age : int,
    fuel : string)
type car =
   (age : int,
    speed : int,
    fuel: string)
```

```
type 2V-garage =
  (v1 : vehicle,
   v2 : vehicle)
type 2C-garage =
  (v1 : car,
   v2 : car,
   j : junk)
type 2M-garage =
  (v1 : machine,
   v2 : machine)
```

OO languages have methods, too

- How does subtyping play here
- Again, the question is, under what conditions is it meaningful to apply a function to an argument?
- The basic rule is:
  - Given
    - f: S → T
    - a : S’ and S’ <= S
  - Then
    - f(a) is meaningful and f(a): T

Example

- Consider any function `g: t → car`
  - Ex: `serial_number: int → car`
- Since `g` returns a `car`, it necessarily also returns a `vehicle`, since `car <= vehicle`
- That means that (`t → car`) **<=** (`t → vehicle`)
  - because `car <= vehicle`

Further example

- Now consider
  - `speed: vehicle → int`
- We can use this to determine the speed of a `car` (because it is-a `vehicle`)
- This means that
  - `(vehicle → int)` <= (`car → int)` because `car <= vehicle`
- Or, more generally
  - `(vehicle → t) <= (car → t)` because `car <= vehicle`

Note carefully

- The reversal of the two examples, depending on whether the subtype relation is on the left or the righthand side of the function arrow
- Cardelli argues this leads to the basic rule for subtyping of functions:
  - if `S’ <= S` and `T <= T’`
    then `S → T <= S’ → T’`
  - Because you can generally constrain the domain of a function and unconstrain the range of a function, without harming the function

Contravariant typing

- This set of rules leads to the notion of contravariant typing
- Again, it ensures that if you have `A <= B` and `a:A` and `b:B` then you can always safely use `a` where you had `b`, and
- you’ll never have a reference to an instance variable that is unknown or to a function that is not meaningful

Example

```
2Dpoint =
  <x : Int,
   y : Int,
   equal : 2Dpoint -> Bool>
3Dpoint =
  <x : Int,
   y : Int,
   z : Int,
   equal : 3Dpoint -> Bool>
```

Contravariant?

- For this example, in small groups for 5 minutes, determine if `2Dpoint <= 3Dpoint`, `3Dpoint <= 2Dpoint`, or neither (can’t be both…why?)

Covariance

- The covariant rule is different, swapping the function relationships
  - if `S’ <= S` and `T <= T’`
    then `S’ → T’ <= S → T`
- This allows different programs to be written, but it cannot guarantee that a “do not understand” error will never arise
  - Eiffel uses covariance checking
  - It uses “system validity checking” to catch some type errors

Some issues in OOP

- Basic object model
  - Hybrid vs. pure OO languages
  - Class-based vs. classless (prototype-based) languages
  - Single vs. multiple dispatching
  - Single vs. multiple inheritance
- Type checking
  - Types vs. classes
  - Subtype polymorphism

Hybrid vs. pure object model

- In a pure object model, everything is an object
  - Not only user-defined objects, but integers, bits, floats, lists, etc.
    - `3.+(4)`
- Everything is instantiated, everything is dynamically dispatched, etc.
- This gives a terrifically consistent programming model
- Ex: Smalltalk-80, Cecil, …

Why hybrid?

- Primarily because of performance
  - Who wants to ask an integer to dispatch a method to add an integer to it?
  - Even “just” an added indirection can be costly, if done frequently enough
- So, hybrid languages (C++, …) allow the programmer to choose what is an what isn’t an object
  - Usually with some constraints; for example, constraining non-objects to a predefined set of types

Class-based vs. classless languages

- Most OO languages have classes
- Some are instead classless
  - Also, prototype-based
- Why?
- The distinction between classes and objects is important but tricky
  - Classless languages eliminate the distinction
  - In principle, this gives a clearer programming model, just like a pure object model does

---

How does it work? Delegation

- Given a message `obj.msg(args)`
- If `msg` is defined in `obj`, invoke it
- If not, the `msg` is passed on to another object that `obj` “delegates” to
  - In some languages, there may be more than one delegate
- If no delegate exists, then it’s an error

How does it work?

- Usually, a programmer simulates a class hierarchy
- A “regular” object delegates to a “class” object
- A “class” object delegates to its “superclass” object

How to create objects?

- In class-based languages, a class holds a constructor (`new` method) that creates new instances of that class
  - This isn’t always exactly right, but it’s close
- In classless languages, there is an object called a *prototype* that knows how to clone itself to create new objects

---

The Smalltalk-80

- Smalltalk-80 is a class-based language
- And it has a pure object model
- That is, everything is an object, and everything has a class
- This means that a class is actually an object, since everything is an object

---

Metaclasses

- If classes are objects, what is their class?
- For each class in the system, there is an associated metaclass
  - Each metaclass is constrained to have a single instance: the given class object
  - The Smalltalk system creates the associated metaclass when you create a class
- If you don’t like how classes work in Smalltalk, you can change the…metaclass class

Single vs. multiple dispatching

- Resolving `obj.msg(args)` is generally done on the class of `obj` alone
  - The class of `args`, for instance, is immaterial
  - (This is true even in classless languages)

But…

- This “single dispatching” can lead to contorted code
- How to handle?
  - 3+1
  - 4.1+5.9
  - 2+6.5
  - 3.5+8
- Two different code bodies
  - + for int, + for float
- Two cases inside each code body
  - One that coerces the argument, one that doesn’t

Multiple dispatch

- Allows the classes of more than just the receiver to determine which method body is invoked

Static type checking

- Types can be separated from classes
  - Types can define signatures
  - Classes can define implementations
- `interface` vs. `class` in Java

Next week

- We’ll look in more detail at some languages that make many of these points more concrete