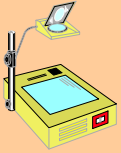


CpSc 372 Software Design & Implementation



What's Object-Orientation

The promises of OO

- (1) easier to modify code
- (2) easier to extend code

2

What people say OO is; a grain of truth in each one

- It's a "new way" of looking at things
- It's a paradigm shift
- You have methods instead of functions
- you pass messages
- If you've used the "procedural approach" than you can't transition to OO
- You use objects, whatever they are!
- You use polymorphism, whatever that is!
- C and C++ are about the same

3

Let's compare OO to the "old ways"

- Previous approach stressed algorithms.
- Referred to as "procedure-oriented" approach
- I like A. Riel's term: "**Action-Oriented**" approach

4

Action-Oriented Approach

- Stresses "getting the algorithm correct"
- uses "functional decomposition"
- Has a "main" calling routine that choreographs the activity, calling functions to implement the application

**It is frequently, but not
always, called "main"!**

5

Object-Oriented Approach

- Stresses the data
- Decompose the data, together with the functions that operate on that data
- Put the data and operations into a "class"
- An instance of the "class" is an object
- Each "object" maintains its own data and operations: must **initialize** and **clean up!**
- Decentralize control

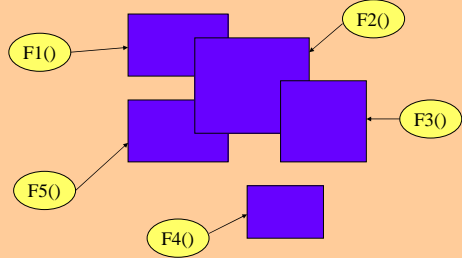
6

Let's consider an example

- Let's compare the worst-case action-oriented with the best-case object-oriented
- Then, we'll consider the case when the action-oriented goes **right!**
- And, the object-oriented goes **wrong!**

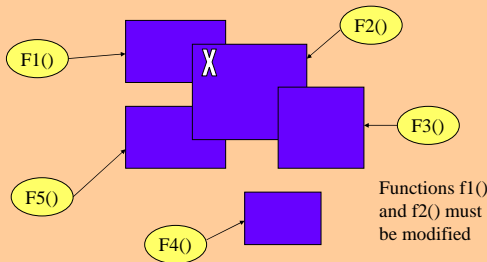
7

A typical action-oriented topology



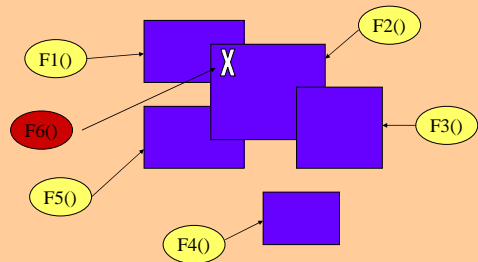
8

Consider a change to the data, marked X



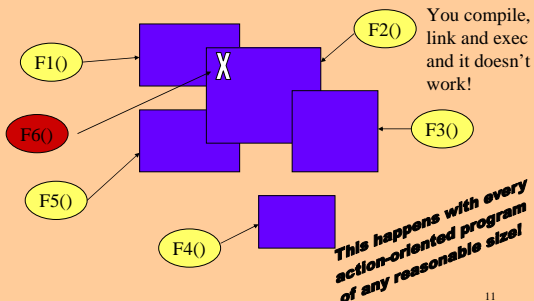
9

However, suppose you or another developer added f6() (and forgot you did it!)



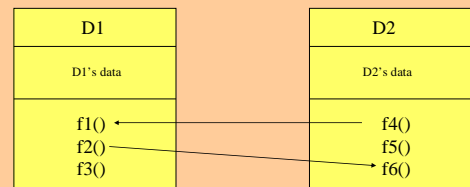
10

However, suppose you or another developer added f6() (and forgot you did it!)



11

How does the OO approach control this complexity



12

Decompose the problem

- Data is encapsulated in a class (or object)
- well-defined public interface permits operations on the data
- This attempts to fulfill the first promise: *“easier to modify code”*

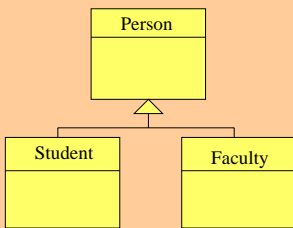
13

Inheritance & polymorphism

- aka: generality-specialization
- base class represents the general item
- the derived class represents a special case of the base class

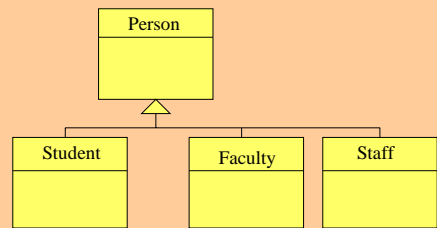
14

example of inheritance



15

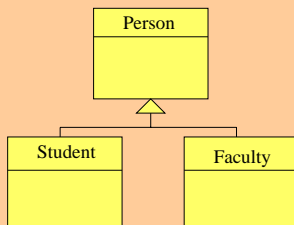
Can extend this class hierarchy without changing existing classes



This attempts to fulfill the second promise:
(2) easier to extend

16

Polymorphism & dynamic binding

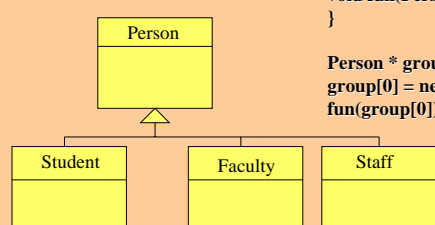


```
void fun(Person * p) {
}

Person * group[100];
group[0] = new Student;
fun(group[0]);
```

17

Fun() still works even if I add *Staff*



```
void fun(Person * p) {
}

Person * group[100];
group[0] = new Student;
fun(group[0]);
```

18

There's a price to pay for the benefits of OO

- Must design the program carefully
- don't overuse inheritance
- design for low coupling, high cohesion
- ridiculous for small programs
- great for large programs that will be reused and extended

19

Does action-oriented approach work? YES!

- Many people build systems using files, with each data structure in a separate file
- the functions that operate on data are placed in the same file
- some languages contain primitives to restrict operations within/across files (e.g. static, extern)
- takes on attributes of OO approach

20

How can OO go wrong

- The god class: performs most of the work, leave minor details to other classes
- proliferation of classes: too many classes for the size of the problem

21

Class design heuristics

- All data is hidden in a class
- implement a minimum public interface that all classes understand
- do not put implementation details in the public interface (e.g. common-code private functions)
- A class should capture one, and only one, key abstraction
- Keep related data and behavior in one place

Messages, inheritance & polymorphism

from Meilir Page-Jones

23

9 properties central to OO

1. Encapsulation
2. information/implementation hiding
3. state retention
4. object Identity
5. messages
6. classes
7. inheritance
8. polymorphism
9. genericity

24

Encapsulation according to Page-Jones

- the grouping of related ideas into one unit, which can thereafter be referred to by a single name
- not a new idea: subroutine -- circa 1940
- OO encapsulation: the packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only via the interface provided by the encapsulation.

25

class Hominoid

```

new: Hominoid
    // create & return an instance
turnLeft
    // turn hominoid counterclockwise by 90
turnRight
    // turn clockwise by 90
advance(noOfSquares : int, outAdvanceOk : bool)
    // moves the hominoid noOfSquares
    // in the direction it's facing, and returns
    // whether successful
    
```

26

class Grid

```

new: Grid
    // create & return an instance of Grid w/
    // a random pattern
start: Square
    // returns the square designated as start of path
finish: Square
    // returns the square designated as finish
insertHominoid(hom: Hominoid, location : Square,
    out insertOk: bool)
    // places hominoid in specified grid
display
    // shows the grid as a pattern on the screen
    
```

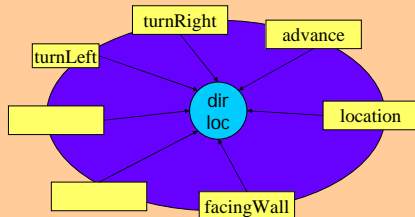
27

1st property: OO Encapsulation

- packaging of operations and attributes representing state into an object so that state is accessible or modifiable only via the interface provided by the encapsulation

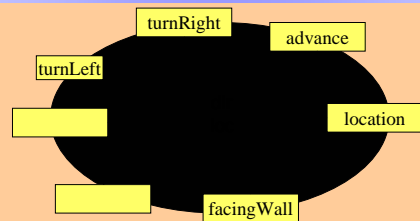
28

OO Encapsulation of Hominoid



29

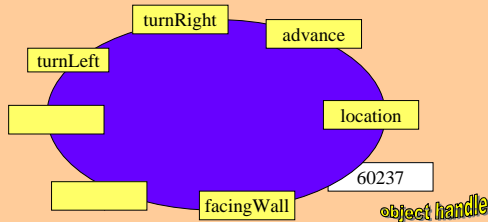
2nd property: information/implementation hiding



info hiding: private info inaccessible from outside
impl hiding: details of implementation hidden
(don't know how direction is implemented)

30

3rd: Object Identity



Each object can be identified and treated as a distinct entity

31

5th property: messages

- a message is a vehicle by which a sender object tells a target object to apply one of its operations

32

3 Parts of a message

- needs the handle of target object (probably stored in one of its variables)
- name of the operation to apply
- any necessary supplementary info, ie arguments

33

format of a message

```
var hom1 : Hominoid := Hominoid.new;  
hom1.turnRight;
```

notice the inversion, as compared to a traditional procedure call & that the message needed no arguments:
call turnRight(hom1);

this preference for object-first format is an important difference between OO structure and traditional structure.

34

four roles objects play

- sender of a message
- target of a message
- pointed to by a variable w/in another object
- pointed to by an argument

35

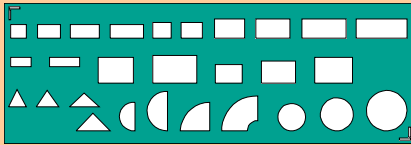
Three types of messages

- **informative message**: provides the target with info to update itself. Past-oriented message
- **interrogative message**: request the target object for info about itself. Present-oriented message
- **imperative message**: requests the target to take some action on itself, another object, or on the system. A future-oriented message

36

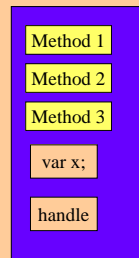
property of OO: Classes

- The stencil from which objects are created!
- a class is what you design & program
- an object is what you create at run-time
- all objects of a class have same structure



37

How about sizes of objects



This object has **instance** operations and **instance** variables.

However, storing the methods in each instance would be wasteful. Thus, they will be stored only once!

38

class operations & attributes

- needed to cope with situations that cannot be the responsibility of any individual object
- the message new is typically not sent to an individual object (since it's not created yet!)
- however, some languages allow overloading new/delete
- static

39

What to do?

Suppose you're designing software that includes a list of items. You notice that most of the items are of type A, except that there are some items, say type B, that are almost exactly like A. What should you do?

Why not duplicate the attributes & operations of A and put them in B?

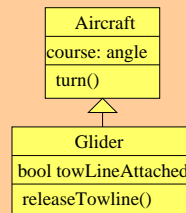
40

property: Inheritance

- The derived class gets the attributes & operations of the base class.
- can build software incrementally:
 - build a class to handle the most general case
 - add special cases that inherit from the general

41

Inheritance



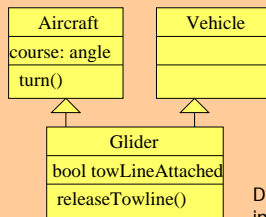
```
var ac : Aircraft := Aircraft.new;  
var gl : Glider := Glider.new;
```

```
ac.turn(NewCourse, out turnOk);  
gl.releaseTowline();  
gl.turn(NewCourse, out turnOk);  
ac.releaseTowline();
```

gl is an instance of more than 1 class!
Glider is an example of **is-a**

42

Multiple inheritance



Design issues: subclass can inherit clashing attributes or operations from the base class.

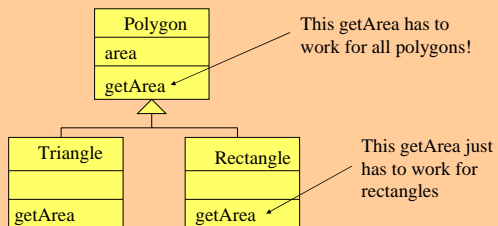
43

property: Polymorphism

- literally: “many” “forms”
- **def #1**: the facility by which a single operation or attribute name may be defined upon more than one class and may take on different implementations in each of those classes.
- **def #2**: the property whereby an attribute or variable may point to (hold the handle of) objects of different classes at different times.

44

The is-a relationship



This getArea has to work for all polygons!

This getArea just has to work for rectangles

twoDShape.getArea -- object may not know the exact class of the target object it's sending the message to!⁴⁵

45

Implementation hiding

```
var p : Polygon;
var t : Triangle := Triangle.new;
var h : Rectangle := Rectangle.new;
```

```
if (itsOkay) then p := t;
else p := h;
p.getArea;
```

We can add a new subclass of Polygon without changing this code.

getArea, defined on several classes, is an example of defn #1 of polymorphism.
Variable p, capable of pointing to objects of several different classes, is example of defn #2.

46

Dynamic Binding

```
var p : Polygon;
var t : Triangle := Triangle.new;
var h : Rectangle := Rectangle.new;
```

```
if (itsOkay) then p := t;
else p := h;
p.getArea;
```

is the technique whereby the exact piece of code to be executed is determined only at run-time (as opposed to compile time).

47

Three easy-to-confuse terms

- **polymorphism** -- depends on the class of the target object
- **overloading** -- depends on the signature of the operation
- **overriding** -- same signature in base and derived class

48

genericity

- the construction of a class so that one or more classes that it uses internally are supplied at the time of instantiation.