



Essential OO Concepts

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
<http://www.iam.unibe.ch/~ducasse/>
 Stéphane Ducasse
stephane.ducasse@univ-savoie.fr

S.Ducasse

1

Outline

- OOP
- Objects, classes
- Inheritance
- Composition
- Comparison



S.Ducasse

2



Object-Orientation

- Is a paradigm not a technology
- Reflects, simulates the real world
- Thinks in terms of organization
- Tries to
 - Handle complexity
 - Enhance reusability
 - Minimize maintenance cost

S.Ducasse

3



Evolution

- Procedures
- Structured Programming
- Fourth Generation Languages
- Object-Oriented Programming
- ???

S.Ducasse

4



Traditional Point of View

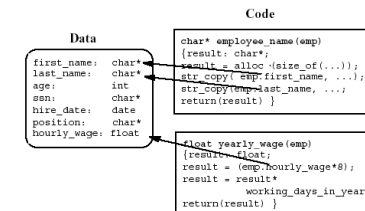
- Focuses upon procedures
- Functionality is vested in procedures
- Data exists solely to be operated upon by procedures
- Procedures know about the structure of data
- Requires large number of procedures and procedure names

S.Ducasse

5



Data and Procedures



S.Ducasse



Roadmap

- OOP
- Objects, classes
- Inheritance
- Composition
- Comparison



S.Ducasse

7



What is OOP?

- An application is a collection of interacting entities (objects)
- Objects are characterized by behavior and state
- Inter-object behavior needs to be coordinated
- Inter-object communication is the key to coordination

S.Ducasse

8



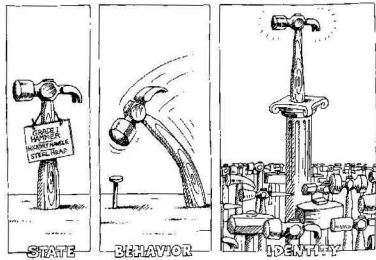
Object-Oriented Viewpoint

- An **application** is a set of **objects** interacting by sending **messages**
- The functionality of an object is described by its **methods**, its data are stored in private **variables**
- An object's functionality can be invoked by sending a **message**
- **Everything** is an object

S.Ducasse



State + Behavior + Identity



S.Ducasse

10



State + Behavior + Identity

- State: Objects it contains or refers to
• Ex: point location
- Behavior: an object understands a given set of messages
- Identity: an object can be the same (of the same class) than another one but it has still a different identity (location in memory)

S.Ducasse

11



Equality and Identity

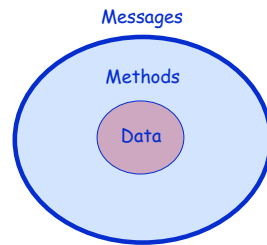
- I want to eat the pizza that you are eating
- Equality: I want to eat the "same" kind of pizza
- Identity: I eat your pizza

S.Ducasse

12



Data/Messages/Methods



S.Ducasse



What vs. How

- What: Messages
 - Specify what behavior objects are to perform
 - Details of how are left up to the receiver
 - State information only accessed via messages
- How: Methods
 - Specify how operation is to be performed
 - Must have access to (contain or be passed) data
 - Need detailed knowledge of data
 - Can manipulate data directly

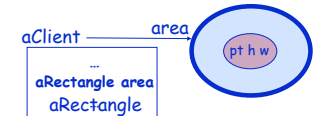


S.Ducasse



Message

- Sent to receiver object: receiver-object message
- A message may include parameters necessary for performing the action
- In Smalltalk, a message-send always returns a result (an object)
- Only way to communicate with an object and have it perform actions

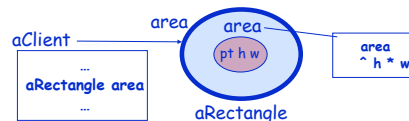


S.Ducasse



Method

- Defines how to respond to a message
- Selected via method lookup technique
- Has name that is the same as message name
- Is a sequence of executable statements
- Returns an object as result of execution



S.Ducasse



Object Encapsulation

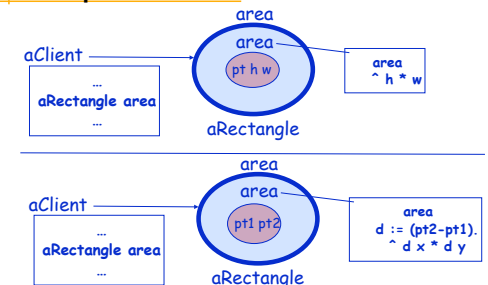
- Technique for
 - Creating objects with encapsulated state/behaviour
 - Hiding implementation details
 - Protecting the state information of objects
 - Communicating/accessing via a uniform interface
- Puts objects in control
- Facilitates modularity, code reuse and maintenance
 - External perspective vs. Internal perspective
 - What vs. How
 - Message vs. Method

S.Ducasse

17



Encapsulation at Work



S.Ducasse



Objects

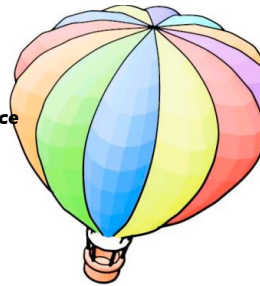
- Unique identity
- Private state
- Shared behavior among other similar objects

S.Ducasse

19

Roadmap

- OOP
- Objects, classes
- **Classes and Inheritance**
- Composition
- Comparison



S.Ducasse

20

Class: Factory of Objects

- Reuse behavior
=> Factor into class
- Class: "Factory" object for creating new objects of the same kind
- Template for objects that share common characteristics

Rectangle
length, width, origin, color, borderSize, borderColor
area
intersect: center
sideNearestTo:



S.Ducasse



Class: Mold of Objects

- **Describe** state but not value of all the instances of the class
 - Position, width and height for rectangles
- **Define** behavior of all instances of the class
 - area
 \wedge width * height

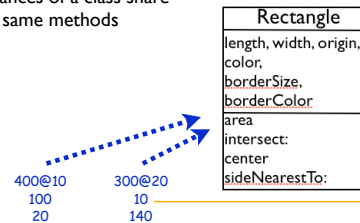
Rectangle
length, width, origin, color, borderSize, borderColor
area
intersect: center
sideNearestTo:

S.Ducasse



Instances

- A particular occurrence of an object defined by a class
- Each instance has its own value for the instance variables
- All instances of a class share the same methods



S.Ducasse

23

How to Share Specification?

- Do not want to rewrite everything!
- Often times want small changes
- Class hierarchies for sharing of definitions
- Each class defines or refines the definition of its ancestors
=> inheritance

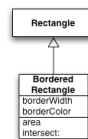
S.Ducasse

24



Inheritance

- New classes
 - Can **add** state and behavior
 - Can **specialize** ancestor behavior
 - Can use ancestor's behavior and state
 - Can **hide** ancestor's behavior
- Direct ancestor = superclass
- Direct descendant = subclass

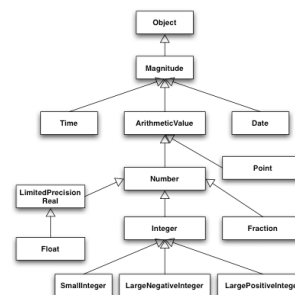


S.Ducasse

25



Comparable Quantity Hierarchy



S.Ducasse

26



Polymorphism - Late binding

- Same message can be sent to different objects
 - Different receivers react differently (different methods)
-
- aCircle area
 - aRectangle area
 - aColoredWindow open
 - aScheduledWindow open
 - aWindow open

S.Ducasse

27



Late binding: "Let's the receiver decide"

Mapping of messages to methods deferred until run-time (dynamic binding)
Allows for rapid incremental development without the need to recompile the complete applications
Most traditional languages do this at compile time (static binding)

S.Ducasse

28



Roadmap

- OOP
- Objects, classes
- Classes and Inheritance
- **Composition**
- Comparison



S.Ducasse

29



Composition

- An object is composed of other objects in a part-of relationship
- The object uses its parts to implement its behavior
- The object can delegate to its parts

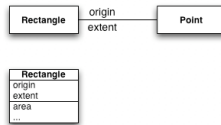
S.Ducasse

30



Example

A rectangle can be composed of two points:
to represent its origin and extent
to represent its topleft and bottomleft corners
or 4 numbers



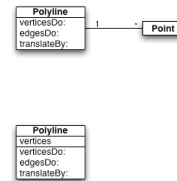
S.Ducasse

31



Example (2)

Polyline has a list of vertices



S.Ducasse

32



Agrégation

- L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination. Elle représente une relation de type "ensemble / élément".
- Peut (mais pas nécessairement) exprimer :
 - qu'une classe (un "élément") fait partie d'une autre ("l'agregat"),
 - qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
 - qu'une action sur une classe, entraîne une action sur une autre.

S.Ducasse

33



Composition vs. Inheritance

Inheritance supports extension: ColoredRectangle

But

static, properties are difficult to change dynamically
we have to change classes at run-time
explosion of classes
class with too much responsibilities

With composition
run-time changes are easier: plug another objects (with the same interface)
but lot of objects

S.Ducasse

35



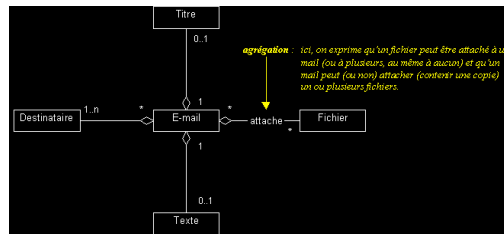
Outline

- OOP
- Objects, classes
- Inheritance
- Composition
- **Comparison**



S.Ducasse

36



S.Ducasse

34



Graphical Editor

- Managing list of objects: square, rectangle, circle...
- Intersect, color, rotate translate....
- We want to know the total area of a list of figures



Procedural Solution

```
tArea
element class = Circle
then tArea := tArea + element.circleArea.
element class= Rectangle
then tArea := tArea + element.rectangleArea
...
```

Same for ...
intersect, color, rotate translate....



In Java for example

```
public static long sumShapes(Shape shapes[]) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        switch (shapes[i].kind()) {
            // a class constant
            case Shape.CIRCLE:
                sum += shapes[i].circleArea();
                break;
            case Shape.RECTANGLE:
                sum += shapes[i].rectangleArea();
                break;
            ... // more cases
        }
    }
    return sum;
}
```



Problems

- Adding a kind of graphical element
- Change all the methods area, intersect, rotate, translate...
- Always have to check what is the data I manipulate



Object-Oriented Solution

```
Circle>>area
^ Float pi * r * r
```

```
Rectangle>>area
^ width * height
```

```
XXX>>area
elements do:
    [each | tArea := tArea + each area]
```



Advantages

- Adding a new graphical object does not require to change the list operations
- I do not have know the kind of objects I'm manipulating as soon as they all share a common interface



Recap

OOP see the world as interacting objects

Objects

Have their **own** state

Share the behavior among similar objects

Classes: Factory of objects

Define behavior of objects

Describe the structure of objects

Share specification via hierarchies



Recap

- OOP is based on
 - Encapsulating data and procedures
 - Inheritance
 - Polymorphism
 - Late Binding
- OOP promotes
 - Modularity
 - Reuse

