Object orientation

Object orientation is imperative programming with some additions

- Abstraction is obtained (also) by *encapsulation*, with the purpose to hide and protect data
- Modularization, which is obtained by a variety of abstraction mechanisms
- Data independence, which is obtained from abstraction and modularization.
  An abstract data type or a class is in essence an existentially quantified data type, e.g.:

```haskell
exception EmptyStack;
abstype 't Stack = S of 't list
  with
    val newStack = S([])
    fun empty (S []) = true
    | empty (S _) = false;
    fun push (S s) x = S(x::s);
    fun pop (S []) = raise EmptyStack
    | pop (S (_::xs)) = S(xs);
    fun top (S []) = raise EmptyStack
    | top (S (x::_)) = x;
end;
```

Its type is

\[ \forall t . \exists S . \text{unit} \to S, S \to \text{bool}, S \times t \to S, S \to S, S \to t \]

where the existential quantification mainly tells us that there is a need for an internal structure to implement the type.

Most of the ideas in object oriented programming stem from Simula, a Norwegian programming language developed in the mid 1960-s.

The purpose was to implement a base for representing processes when simulating reactions in nuclear reactors.

You did not model activities or processes with functions but by finding a representation for its physical counterpart. This means that an object encapsulate its own data (its internal state) and its behavior.

Objects are instances of classes and have an identity (in that every object is unique) and in most object oriented systems every object has a unique object identifier (OID) that

- is automatically generated by the system
- is unique for a specific object
- is invariant (never changes during the life span of an object)
- is independent of its internal state (two distinct objects may have identical inner state but they still have different OID)
- is (ideally) invisible to the user.
Object orientation . . .

OID benefits (and drawbacks)

- OIDs are efficient as they use a minimum of space, typically less than text strings, foreign keys and other semantically based references.

- OIDs are quick as they directly point at the space that the object occupies.

- OIDs cannot be modified by the user as they are system generated and (often) invisible. Thus, they allow for a reference integrity that the user don’t have to manage.

- OIDs don’t depend on object content and thus, the content may be changed without compromising the objects identity.

The last property gives rise to a potential drawback (deficit?). Two objects may have the same content and still be separate objects. How is that managed? And how do you distinguish between the two concepts?

Objects are *identical* if they share OID and *equal* if they have identical content. Sometimes this is referred to as shallow and deep equality.

Objects inherit properties from other objects if the class they belong to is a subclass to another class.

Objects may be very complex by containing references to other objects.

How do we store objects in databases?

One way is of course to use traditional relational databases and then either use a BLOB getting all the usual problems with BLOBs or we can decompose the objects and map their inner state on tables. However, this technique gives rise to problems.

Suppose we have a simple class hierarchy:

Employee
  ssn
  fName
  lName
  position
  sex
  bonus
  startDate

Seller
  ssn
  district

Exec
  ssn
  bonus
  startDate

Admin
  ssn
  field

Let’s ignore the fact that we might have to use special tools for some data types and suppose that we can handle all types without problems.

Still we lose in terms of semantics. How do we distinguish between classes and subclasses? The only way is to code it in the software we use to access the database.
How do we store objects . . .

2. Map each subclass on a base relation.

Seller (ssn, FName, LName, position, sex, salary, district)
Exec (ssn, FName, LName, position, sex, salary, bonus, startDate)
Admin (ssn, FName, LName, position, sex, salary, field)

Again we lose semantics. It is far from clear that these tables represent subclasses to a common superclass and to list all employees we have to collect data from three tables and create the union of the common attributes from the three.

3. Map the complete class hierarchy onto one single table.

Employee (ssn, FName, LName, position, sex, salary, district, bonus, startDate, field, empType)

I added empType to see to which subclass an object belongs.
We get the same semantic loss as with the other methods and we still cannot see that we have three subclasses to a common superclass and also we will store a lot of NULL values.

Object oriented databases

Atkinson, et.al. published “The Object-Oriented Database System Manifesto” in 1989 in which they claimed that object orientation would dominate the future and gave directions as to what had to be included.

They meant that one must create object oriented database management systems (OODBMS) to meet the need for persistence in object oriented lanuages.

In short they wanted OODBMS to have all properties of object oriented lanuages as well as all properties of traditional DBMS.

In 1991 Kim defined what he considered to be
- an object oriented data model (OODM), a data model describing the semantics of an object as they are supported in object oriented programming,
- an object oriented database (OODB), a place to store persistent sharable objects as defined in OODM and finally
- an object oriented database management system (OODBMS), a DBMS made for the management of an OODB.

Object oriented databases

The definitions are quite “fuzzy” and show, in essence, that there is no uniform and commonly accepted OODM as a fundament for OODBMS as in the world of relational databases where all RDBMS use the common relational model as common fundament

There is, however, a uniform functional data model, defined by Kerschberg in 1976 that could have served as a basis for object oriented systems

It builds on objects and functional relationships.
Researchers saw many strategies to develop OODBMS. One was to extend an OOP language like Smalltalk, C++ or Java. A direct extension of Smalltalk is GemStone, attempting to turn Smalltalk into a full fledged OODBMS.

Application programs may be written in Opal (the extension of Smalltalk), C(!), C++, Java and Pascal(!).

The system was built on Smalltalk-workstations (PC, Mac, Tektronix, ...) and a server running in VMS on a Vax-machine or on some Unix-machine.

GemStone builds entirely on the Smalltalk model with classes, metaclasses, objects and messages.

Classes are organized as a hierarchy with simple inheritance. OPAL is used as DDL, DML, programming language and command language. There is an IDE (OPE = OPAL Programming Environment) that contains:

- a class browser that allows for adding, deleting, inspecting and modifying GemStone classes (in the usual Smalltalk manner),
- a file management module (Bulk Loader/Dumper)
- an editor (Workspace editor) that allows you to write and run OPAL programs.
- PIM (Procedural Interface Modules) that makes it possible to connect to C and Pascal programs via remote procedure calls.
The server is divided into two parts:

- Gem-processes, implementing the virtual machine (standard Smalltalk) and the object memory. Gem compiles and runs OPAL methods and manages authentication and session control.
- Stone-process, managing secondary memory, concurrency control, access rights, transactions, recovery and workspace for active sessions.

Another strategy is to provide extensible OO database libraries as in ObjectStore.

ObjectStore is a multi-client/multi-server system where each server manages an object store and also manages concurrency control and access control.

A client connects to any server

There is one server and/or any number of clients on each computer in the network.

On every computer hosting at least one ObjectStore client there is also a cache manager and each application has its own client cache where objects are kept when activated (automatically fetched from secondary memory when accessed) and are waiting to be rewritten to persistent memory.
ObjectStore is written entirely in C++ and manages objects in binary format without mapping and remapping pointers (swizzling).

An object is directly accessed if already in some client cache and if not an error is generated.

Errors are generated if the object is in a cache but never accessed or accessed with different access rights than at previous access.

Errors of this kind will reactivate the object (reread it from DB) possibly after synchronizing it.

Objects are accessed with a query language (an extension to C++) and standard C++ templates are used to define which objects shall be persistent.

CORBA – Common Object Request Broker Architecture

emphasis on “common” and “broker”.

CORBA does not standardise object representation or implementation but only how to handle objects.

CORBA provides an enabling technology as a kind of infrastructure for incorporating component into distributed application programs.

One objective was to make the building of large distributed systems as simple as building centralised systems.

What has been decided?

In spite of the apparent lack of cooperation and lack of standards some things have been agreed upon mainly by consortiums of enterprises.

The Object Management Group (OMG) has defined a standard for object communication named CORBA (Common Object Request Broker Architecture) with certain interesting properties.

Not so easy but the CORBA consortium has delivered a language and platform independent standard for distribution of program components.

CORBA is built as a client-server system

- a server offers a service
- a client makes use of it
- there are objects in the network that can act either client, server or both

There are bindings to “almost all” languages.

Whoever wants to know more can find links on the course link page.