



---

**Poseidon House  
Castle Park  
Cambridge CB3 0RD  
United Kingdom**

TELEPHONE:  
INTERNATIONAL:  
FAX:  
E-MAIL:

**Cambridge (01223) 515010  
+44 1223 515010  
+44 1223 359779  
apm@ansa.co.uk**

---

## **Training**

# **Computational (Intro to ANSA)**

**Yigal Hoffner**

### **Abstract**

The business problem addressed is...

The technical problem created by that business problem is ...

The solution being offered is....

---

APM.1647.00.01

**Draft**  
Briefing Note

2nd November 1995

---

**Distribution:**  
**Supersedes:**  
**Superseded by:**





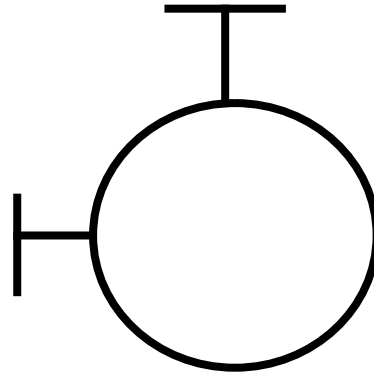
## Object Orientation

- **The Computational Model of objects**
  - describes how logical objects provide services to each other
- **Programming languages and implementation techniques**
- **Discusses OOPs and the concepts they use**



---

# The Computational Model





## In this session

- Outline the rationale for object orientation
- Describe the Computational Model of objects:
  - Defines objects and interfaces
  - Describes usage and interaction relationships
  - Ignores engineering and implementation issues



## Why object technology?

- *Object technology enables systems to be built out of modular components*
- *These systems offer*
  - Greater productivity through reuse
  - Better quality through proven components
  - Lower costs through reduced maintenance overheads



## Influence of object technology

- ***Now affecting all areas***
  - operating systems
  - programming languages
  - databases
  - user interfaces
- ***Products and standards are emerging in all of these***
  - but at different levels of maturity...
  - ...with patchy support from development methods and tools
  - ...and with important technical differences
- ***Not all of these products and standards are 'open'***
  - this poses a challenge to those building systems with object technology



## What is the Computational Model?

- **It is a model for specifying service provision and use**
  - **Objects may provide multiple services to other objects**
  - **Objects may use multiple services from other objects**
- **Objects and their services may be created and destroyed dynamically**
- **The ability to use a service may be passed from one object to another**
- **The Computational Model describes interacting logical objects**
  - **It does not address how objects are implemented or distributed**



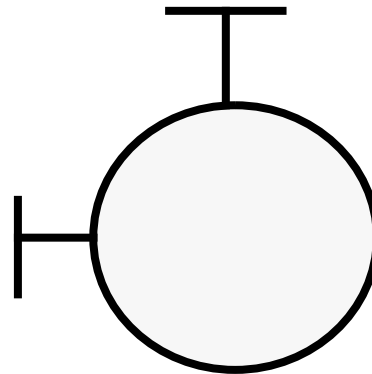


## What is meant by 'object'?

- **Nowadays all software is supposed to be 'object-oriented'**
  - OOA/OOD (analysis and design methods)
  - OOL (programming languages)
  - OODBMS (databases)
  - ... and many more
- **All these have one key concept in common...**

## Encapsulation - the key concept

- **Objects are encapsulated**
  - every object provides a service via interfaces
  - the interface is public; the implementation is private and hidden
  - encapsulation forms a boundary; the only access to an object is via its interfaces



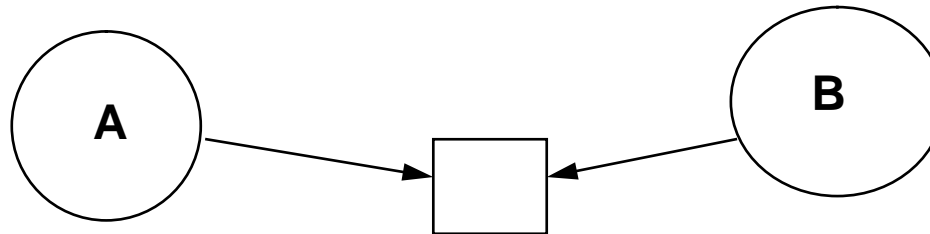


## Encapsulation hides data representation

- **The boundary hides the data representation used by the object**
  - **Diversity: different objects use different representations**
  - **In a distributed system, different representations are used in different places**
  - **One object cannot manipulate another's data (only via the interface)**
  - **Data representations cannot be transferred between objects**

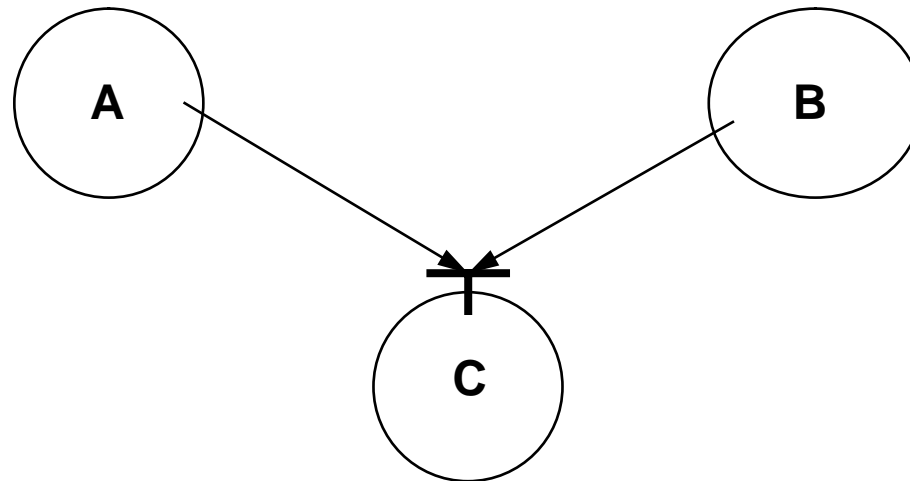
## Encapsulation rules

- We assume nothing about the location of computational objects
  - Objects A and B could be on the same machine, or in different countries
- Objects cannot share state directly (only access it via interfaces)
  - *this is not allowed*



## Sharing state via an object

- **No back door is needed; use an interface to a shared object instead**
  - **Both A and B can access their shared state stored in C**





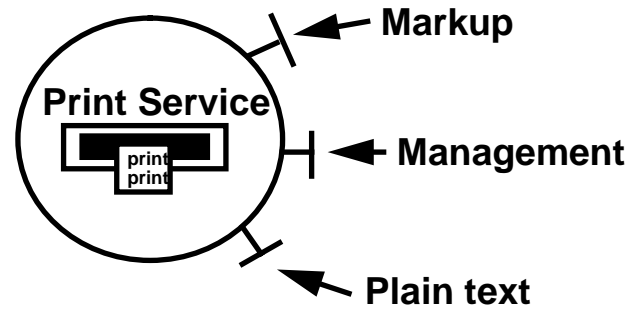
## Multiple interfaces to an object

- **Multiple interfaces allow flexible configurations...**
- **... Consider a service for printing documents**
- **It must print two kinds of document**
  - **plain text documents**
  - **documents in some markup language, for example PostScript (TM)**
- **It must also have a management interface for**
  - **starting and stopping a print queue**
  - **delete print jobs**
  - **other administrative tasks**



## A print service with multiple interfaces

- One object with plain text, markup, and management interfaces





---

## Objects and Interfaces - summary so far

- **Objects encapsulate state**
  - and must take full responsibility for it
- **Interfaces provide the service access points**
- **Objects can have multiple interfaces**
- **[ Object-oriented programming languages:**
  - use one construct for both encapsulation and service provision
  - provide only a single interface ]

with their shared state





---

## Operations - the actions in an interface

- An interface defines one or more *operations*
- Operations are the actions that the user of the interface can invoke
  - Similar to "methods" in object-oriented languages
- Because of encapsulation, the defined operations are the only way to act on the object
  - No "back door" [ not even for management or debugging ]



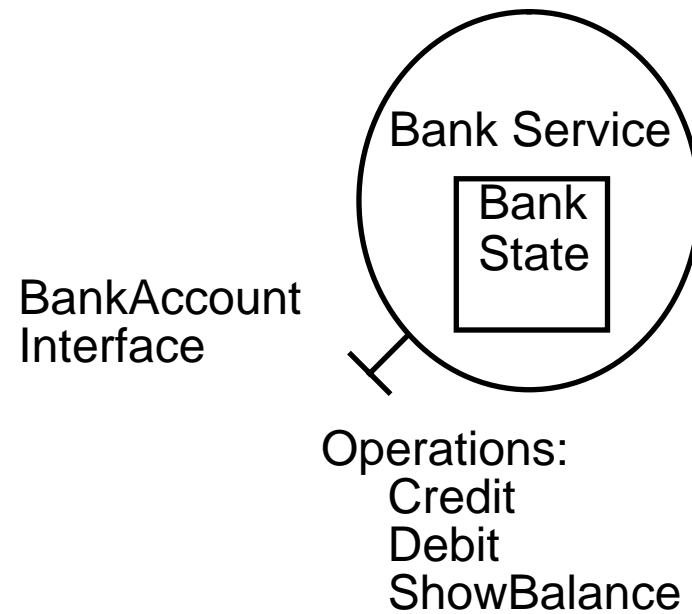
---

## Operations - a Simple Bank example

- **Consider a bank account service...**
- **Each bank account has state**
  - **account balance**
  - **name of customer**
  - **... and others**
- **Each bank account has actions that affect this state**
  - **Credit: depositing money, increases the balance**
  - **Debit: writing a cheque, decreases the balance**
  - **Show: enquires the account balance**
- **These actions are the only way of manipulating the bank account; these actions are the *operations***



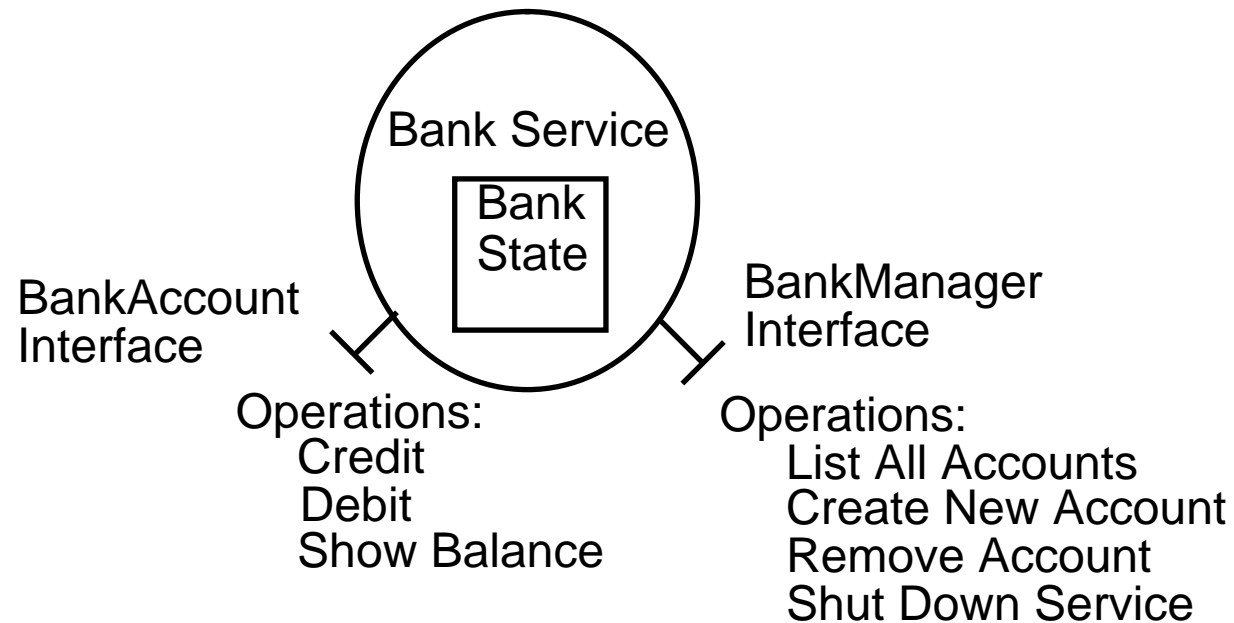
## Simple Bank - Service, interface, and operations





## Simple Bank - Operations in multiple interfaces

- A service can have more than one interface
- Each interface has its own operations





## Operations and Arguments

- **Each operation has a name, input arguments, and output arguments**
  - for example, the **Credit** operation returns the new balance  
  
Credit (amount : Integer) -> (newBalance : Integer)
  - **amount** is a **input argument**; **newBalance** is an **output argument (result)**
- **Each operation can have multiple input and output arguments**
  - **this example only has one of each**



## Operations and Terminations

- An operation can have more than one possible outcome
- Consider operations on the BankAccount:
  - Credit (amount : Integer) -> (newBalance : Integer)
  - Debit (amount : Integer) -> (newBalance : Integer)
  - ... but the balance may be insufficient! - so
  - Debit (amount : Integer) -> (newBalance : Integer) ->InsufficientFunds()
- There is one Debit operation with two possible outcomes; two separate *terminations*



## Named Terminations

- The InsufficientFunds outcome is called a *Named Termination*
  - the normal outcome is an unnamed (anonymous) termination
  - one termination of an operation may be anonymous
- Each termination may return multiple results
- The invoker of an operation distinguishes between the different terminations by their names



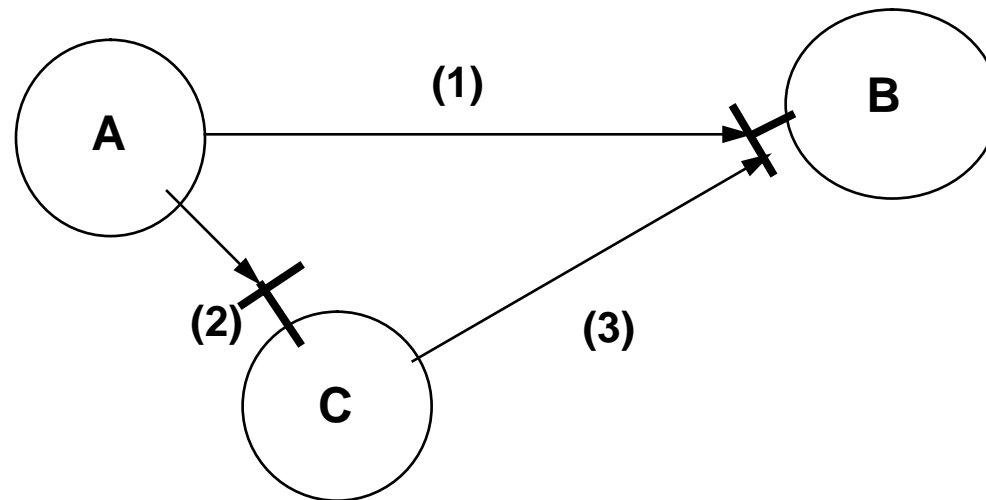
## Operations, Arguments, and Terminations - summary

- **An *operation* is the unit of interaction. It has:**
  - a name
  - a signature (list of input arguments)
  - a set of *terminations* (possible outcomes)
- **A termination has:**
  - a name
  - a signature (list of output arguments)
- **Terminations make operations more powerful than ‘functions’ or ‘methods’**



## Using Interfaces

- Object A is using one of object B's interfaces
- Suppose it needs to tell C to use the same interface



- It must be possible to pass a binding to B's interface between A and C

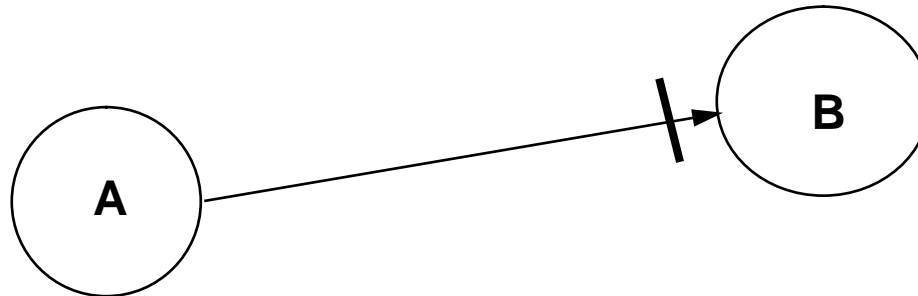


## Interface References

- Interfaces are identified by *interface references*; these can be passed to other objects as arguments (or results)
  - for example, when A passes B's interface reference to C, it calls...  
`C_op (b: TypeOfInterfaceToB) ->()`
  - ... with B's interface reference as the argument
- Only the interface reference is passed, not the interface...
- ... In fact, all operations are invoked via proxies constructed using interface references

## The right interface?

- Suppose object A has a binding for an interface of B...
- ... how does object A know that B has the right kind of interface?
  - it can't inspect the object itself
  - objects A and B could be on the same machine, or in different countries



- It can tell, because interfaces have an *interface type*

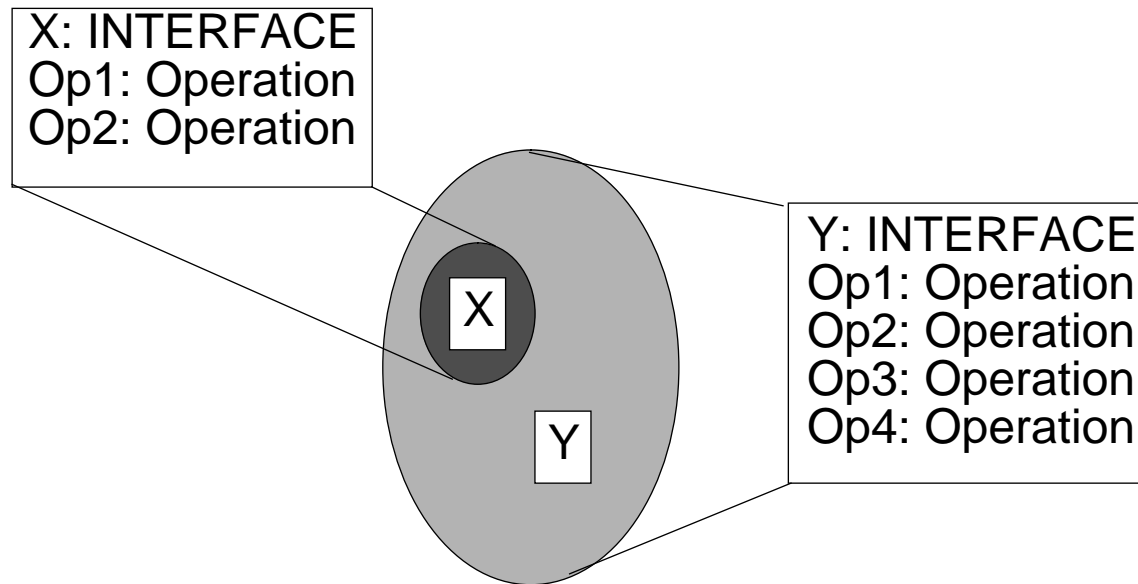


## Interface types

- The interface type consists of the set of its operations
  - Plus all their signatures and terminations
- Object A can use an interface of object B, provided that the interface provided by B *conforms* with that expected by A
- This means that object A must state what interface type it expects

## Type Conformance

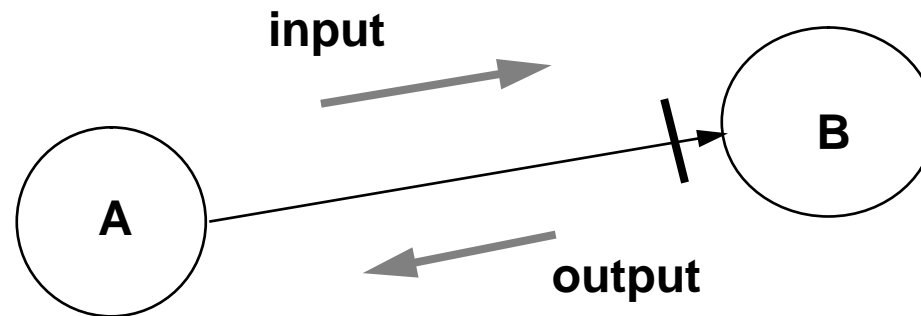
- **Conforming interface types do not have to be identical...**
- **...Interface type *Y conforms to* interface type *X***



- ***Y is a 'subtype' of X... but Y can 'do more' than X, not less!***

## Type Conformance Has Two Sides

- To conform, B must provide at least the operations expected by A...



- ...but also, A must handle at least the terminations given by B
  - same idea, but the 'at least' rule is the other way round for terminations
- To conform, both of these must be checked... [ a two-sided *contract* ]
  - B must not receive unknown operations
  - A must not receive unknown terminations



## Summary

- **Objects enforce strict encapsulation**
  - state cannot be shared between objects
  - only whole objects can be re-configured (*part of an object cannot be*)
- **Interfaces provide the points of service provision**
  - objects may have multiple interfaces
  - interfaces are typed
- **Interfaces are composed of operations**
  - operations can have multiple terminations with multiple results
- **For more information:**
  - for object-oriented concepts, see *Distributing Objects* (TR.018.01)
  - for a formal description, see *The ANSA Computational Model* (AR.001.01)



---

## Computational Model - topics not covered

- **Concurrency**
  - threads and sub-threads, enforcing tree-structured concurrency
- **Streams - continuous (possibly bi-directional) information flows, for example:**
  - speech, in telephony and voice processing: an audio stream
  - video, in multimedia: a video stream
  - sensor data, in telemetry: a data stream
- **Synchronous programming constructs**
  - reactive systems, bounded execution paths
  - testing, receiving, waiting for and transmitting signals
  - watchdogs