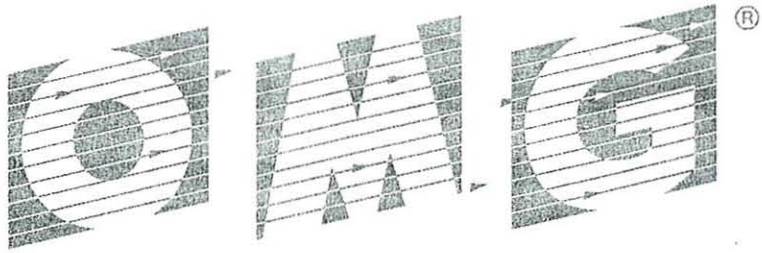
NON-TERMINATING ARGUMENTS IN DISTRIBUTED OBJECT ARCHITECTURE

A Watson

Rapporteur: Dr P D Ezhilchelvan



OBJECT MANAGEMENT GROUP

Non-terminating arguments in distributed object architecture

Andrew Watson
andrew@omg.org
VP & Technical Director, OMG

Overview

- Background
  - OMG objectives
  - A little on how OMG works
  - How and why the discussions happen
- The arguments
  - The Classical vs. Generic Object Model argument
  - The Object Reference Identity argument
  - The Type arguments

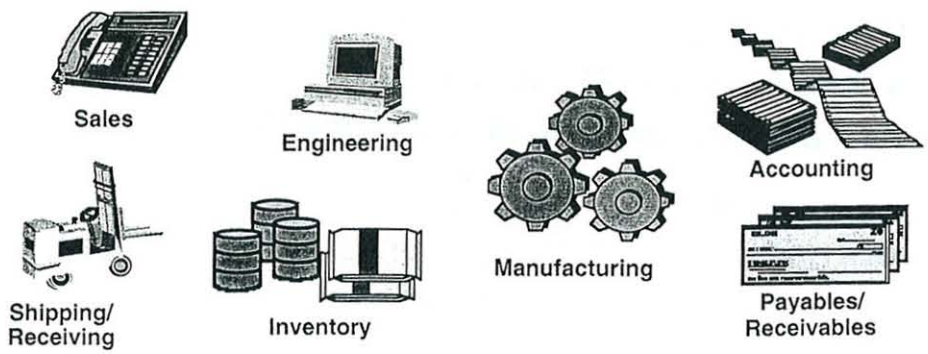
OBJECT MANAGEMENT GROUP

OMG beginnings

- Began in 1989 as informal cross-company workshops
  - Early adopters saw promise of OO, but had almost no tools
  - HP, Sun, DG, etc. lacked resources for both tools and apps
- Objective: to set interoperability standards for object software
- OMG's technical meetings still dominated by technologists
  - ... but have grown from 40 to 600 attendees
- Initially OMG focussed on using the hot new technology
  - Today we realise problem is integration, distributed objects are just a means to that end

OBJECT MANAGEMENT GROUP

Integration

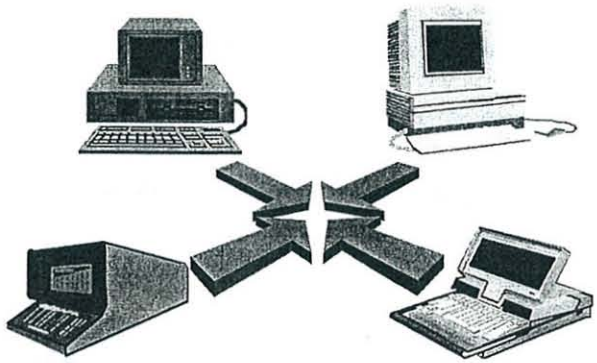


Application integration is a common problem

OBJECT MANAGEMENT GROUP

It gets worse ...

- Your partners have legacies too, and they're incompatible with yours
- No-one is in overall charge
- No-one can impose one technology
  - No single language
  - No single OS
  - No single network



OBJECT MANAGEMENT GROUP

Heterogeneity abounds

- In programming languages
  - 3 million programmers write COBOL for a living
  - c.f. 1.6 million use Visual Basic, 1.1 million C and C++
- In operating systems
  - Unix, MVS, MacOS, NT, Windows, Windows CE, PalmOS ...
  - A significant fraction of Windows installations are still 3.x
  - Then there's your pager, cell phone, set-top box ...
- In networks
  - Ethernet, ATM, IP, SS7, Appletalk, USB, Firewire ...
  - ... and whatever links the 30-odd computers in your car

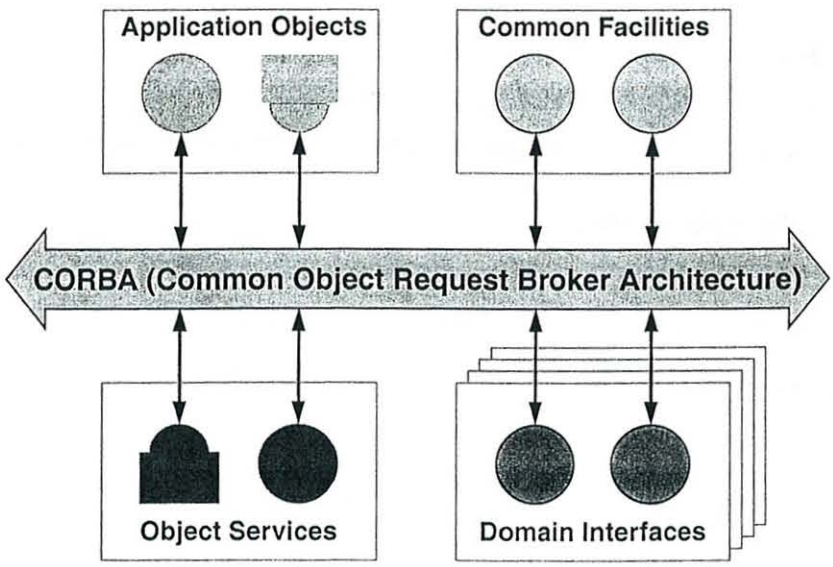
OBJECT MANAGEMENT GROUP

Focus on interfaces

- There will not be consensus on hardware platforms
- There will not be consensus on operating systems
- There will not be consensus on network protocols
- There will not be consensus on programming languages
- There must be consensus on interfaces and interoperability

OBJECT MANAGEMENT GROUP

OMA Reference Model



OBJECT MANAGEMENT GROUP

OMG background (1)

- OMG as a whole maintains overall technical architecture
- OMG groups write requirements ("RFP"), then evaluate submissions in particular domains
  - "Platform" groups handle CORBA, UML and related specs
  - "Domain" groups work in healthcare, telecoms, finance etc.
- Submitters cooperate on interoperability standards
  - ... then compete selling products implementing them
  - Specifications written by small teams outside OMG process (monopoly & anti-trust laws)
  - Organisation now has more users than vendors

OBJECT MANAGEMENT GROUP

OMG background (2)

- Organisation spent 1989-90 determining technical direction
  - OO desktops? Like later OpenDoc, OLE, GNOME
  - OO databases? Group left OMG to start ODMG
  - OO analysis & design? UML effort started 7 years later
  - OO middleware CORBA: OMG's first technology
- Characterised by requirements discussions
  - Desktops need fast method dispatch for performance
  - Databases rely on closed-world assumption
  - Distributed systems must be extensible, flexible

OBJECT MANAGEMENT GROUP

Architecture discussions

- In the course of setting architecture, groups of experts hammer out requirements, discuss shared assumptions
- Today these are often about telecoms, utilities, etc., but in early days they were about fundamental issues in object architecture
- Most architecture definitions & decisions now settled
  - e.g. "What is inheritance?"
  - However, some repeatedly re-surface
  - New participants bring them back in like old email viruses
  - Seem to reflect a general lack of consensus Out There

OBJECT MANAGEMENT GROUP

The Non-terminating arguments

- "Generic" vs. "Classical" object model
- Object Reference equality tests
- Type-related arguments
- Multiple interface arguments (if we have time)

OBJECT MANAGEMENT GROUP

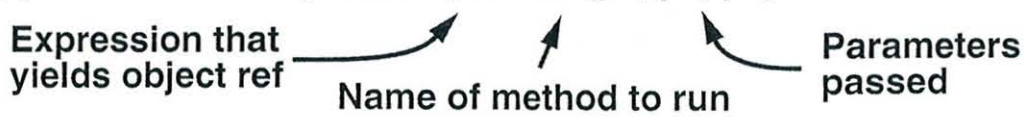
Argument the First

Classical vs. Generic Object models

OBJECT MANAGEMENT GROUP

Generic vs. Classical model

- "Classical" object model is the one everyone knows today
  - Implemented by Smalltalk (pure form), C++ (slightly debased), CORBA
  - Object = encapsulated data + methods operating on them
  - Data only manipulated via methods
  - Every invocation directed to one object
  - Every method associated with just one object class
- Typical non-LISP syntax: pr.name(p1, p2, p3)



OBJECT MANAGEMENT GROUP

Generic function model (1)

- Appears to have originated with Lisp OO extension syntax
- Typical non-OO LISP syntax: (fn p1 p2 p3)

- Typical OO LISP syntax: (name obj p1 p2 p3)

- At some point someone (who?) asked "What's special about obj's parameter position?"

OBJECT MANAGEMENT GROUP

Generic function model (2)

- Generic model LISP syntax: (gf obj1 obj2 obj3)

- Semantics now change radically
  - All expressions evaluated - first yields generic function
  - Types of obj1 .. 3 are determined at run time, used to look up method in generic function
  - Selected method runs, directly manipulating concrete representations of obj1 .. 3

OBJECT MANAGEMENT GROUP

Generic function model (3)

- Implemented by Common Lisp Object System, IRIS database, a couple of experimental languages (e.g. Cecil)
- New terms, and old terms given new meanings
  - Object: A pure record with no associated code
  - Method: Pure code able to operate on multiple objects
  - Generic function: A collection of methods, one of which is selected based on invocation parameter types
- Example CLOS Generic Function:

```
(defmethod distance ((p1 cartesian) (p2 cartesian)) .. )
(defmethod distance ((p1 polar) (p2 polar)) .. )
```

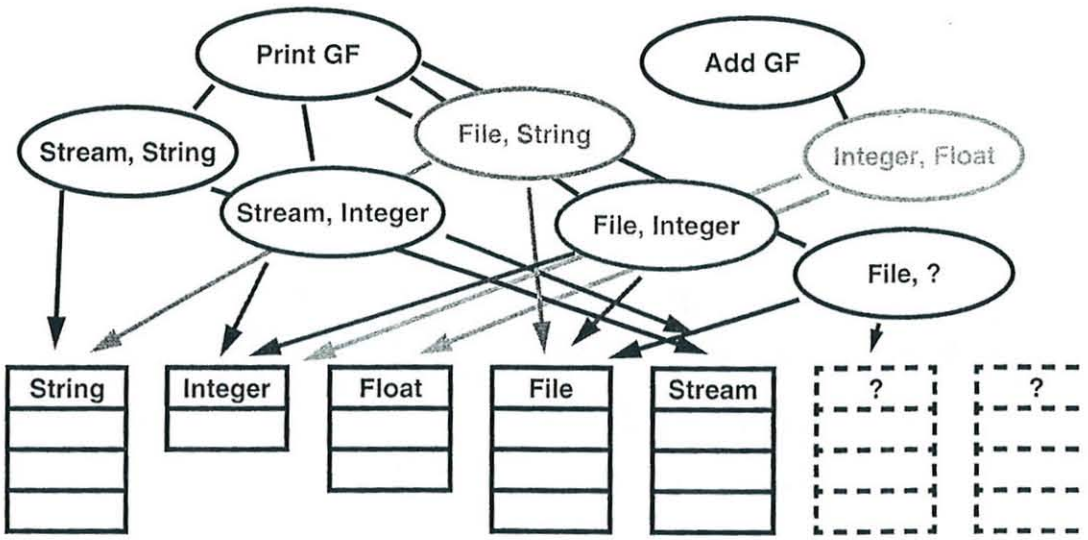

OBJECT MANAGEMENT GROUP

Are these really objects?

- Much early OMG discussion of "What is an object?"
- By my definition, Object systems must have three properties
  - Encapsulation
  - Run-time instantiation
  - First-class object references
- Generic Function model doesn't encapsulate
  - Methods know concrete representation of multiple "objects"
  - "Objects" can be operated on by multiple methods
- Too put it another way - what is the Unit of Distribution?

OBJECT MANAGEMENT GROUP

Unit of distribution?



OBJECT MANAGEMENT GROUP

Multi-methods and subtyping

- If "objects" can be subtyped, method selection a nightmare
  - Horribly complex rules to cope with cases like:

```
(defclass sub1 (super1) <slots> )
(defclass sub2 (super2) <slots> )
(defmethod wombat ((x sub1) (y super2)) <code> )
(defmethod wombat ((x super1) (y sub2)) <code> )
(wombat (make-instance sub1) (make-instance sub2)
```

  - Include multiple class inheritance, and it gets really complex
  - Makes closed world assumption, knowledge of "all" types

OBJECT MANAGEMENT GROUP

Argument One Conclusion

- "Classical" object model provides encapsulation
- "Generic" object model doesn't
  - And so (by my definition) isn't an object model
  - Cannot be distributed
- Ironically, CLOS programmers very rarely use multi-methods
  - So wouldn't notice if they went back to a "classical" model
- After much early discussion, this argument now rarely heard

OBJECT MANAGEMENT GROUP

Argument the Second

Object reference comparison

OBJECT MANAGEMENT GROUP

A Scenario

- My Java function maps file descriptor to file name, as a string
  - Clients use function to find multiple access to the same file
- One day I modify implementation to copy file name before returning it
  - Results look the same, are equal via string.equals()
- Some clients break
  - What happened?

OBJECT MANAGEMENT GROUP

Java language manual 1.0 (excerpt)

15.20.3 Reference Equality Operators == and !=

...

At run time, the result of == is true if the operand values are both null or both refer to the same object or array; otherwise, the result is false.

...

While == may be used to compare references of type String, such an equality test determines whether or not the two operands refer to the same String object. The result is false if the operands are distinct String objects, even if they contain the same sequence of characters. The contents of two strings s and t can be tested for equality by the method invocation s.equals(t) (§20.12.9). See also §3.10.5 and §20.12.47.

OBJECT MANAGEMENT GROUP

To paraphrase ...

- Two Java strings can be completely identical in every string-related respect, and behave identically under all string operations, yet test as different using ==
- You can use == on strings if you like, but it won't tell you anything useful
- == isn't really a "reference equality test", it's an object instance location test
  - Two object instances can behave identically but be in different places

OBJECT MANAGEMENT GROUP

CORBA 2.3.1 spec (excerpt)

4.3.6.2: Equivalence Testing

```
boolean is_equivalent(in Object other_object);
```

The is_equivalent operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns TRUE if the target object reference is known to be equivalent to the other object reference passed as its parameter, and FALSE otherwise.

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object.

OBJECT MANAGEMENT GROUP

To paraphrase (again) ...

- A valid CORBA implementation can return FALSE for all calls to is_equivalent, including:

```
a.is_equivalent(a)
```

- This all seems unusually unhelpful - why can't CORBA have a "real" object equivalence test like Java, C++, etc.?

OBJECT MANAGEMENT GROUP

What is ==?

- Most non-distributed OOPLs have something like it
- Hard to explain semantics
  - So most textbooks and language specifications don't try
  - Common Lisp spec explains lists in terms of [car, cdr, cons] abstraction - but watch the authors wriggle explaining "eq"
- Easy to explain implementation
  - Most authors appeal to machine-level explanation
  - Bitwise comparison of short (32 bit?) pointer
  - Runs like greased lightening

OBJECT MANAGEMENT GROUP

Reference comparison breaks encapsulation

- Encapsulation prevents client knowing anything about an object that the object chooses not to tell it
  - But == isn't under the control of the object
  - Abstraction bypassed, allowing client to discover an implementation detail (address of instance)
  - Like any encapsulation violation, limits ability to substitute equivalent implementations for existing objects
- It's possible to design == into the abstraction
  - LISP symbols are strings that are "interned" to collapse all instances of equal strings into == symbols
  - Implies ability to search whole string space (closed world)

OBJECT MANAGEMENT GROUP

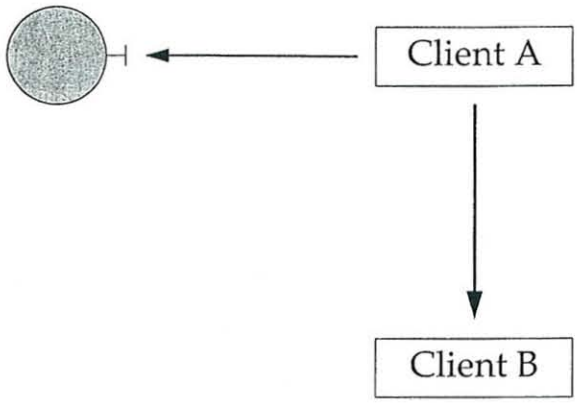
Reference comparison and distribution

- In non-distributed environment, one = = ref is a copy of the other, or they're both copies of a common ancestor
- Even these semantics break down in a distributed environment
  - Distributed object references aren't simple pointers
  - Copying and moving them can change representation
  - ... as can relocating, monitoring or debugging object ...
  - ... and novel implementations (e.g. groups)

OBJECT MANAGEMENT GROUP

CORBA object relocation (1)

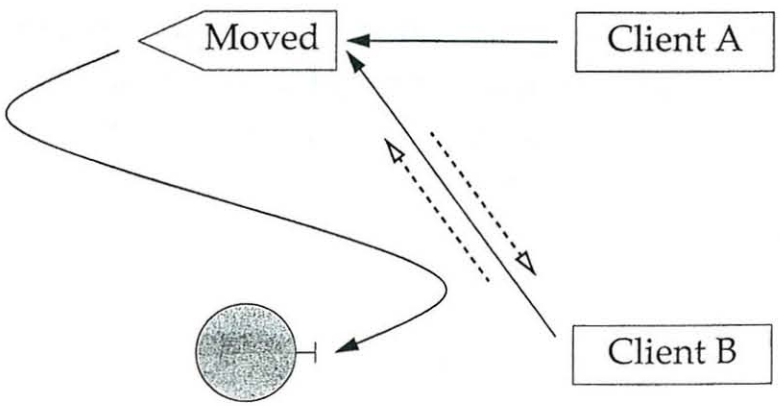
Client A passes object reference to client B



OBJECT MANAGEMENT GROUP

CORBA object relocation (2)

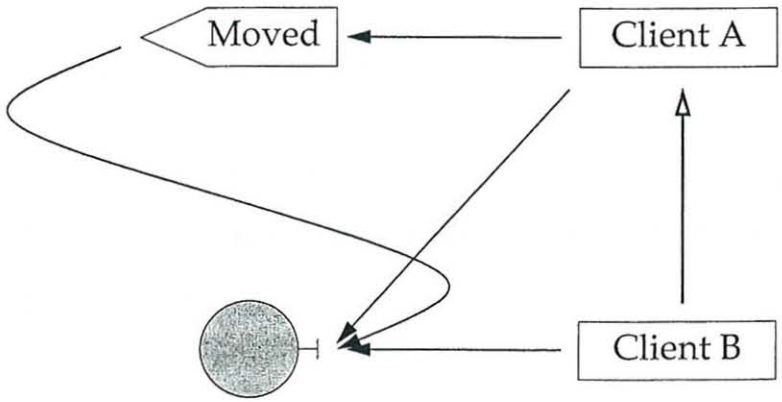
Object migrates, leaving forwarding pointer. Client B invokes object and learns new location.



OBJECT MANAGEMENT GROUP

CORBA object relocation (3)

Client B sends updated reference back to A, which compares them, but does not discover they were originally copies.



OBJECT MANAGEMENT GROUP

Groups implement objects

- **Fault tolerant CORBA implements object abstraction as replicated groups of objects**
- **Behind the object abstraction, client object reference is "really" a list of references to distributed replicas**
  - **Client maintains replicas to avoid single point of failure**
  - **Population can change as replicas die, are replaced**
  - **Lazy update protocols, not all clients have same group list**
  - **Similar problem to relocation case arise**
- **Interposition for debugging, monitoring suffers same problem**

OBJECT MANAGEMENT GROUP

Aside: What does "same" mean?

- My, there's a slippery word
- Leibniz's Identity of Indiscernibles usually stated as follows:

  If, for every property F, object x has F if and only if object y has F, then x is identical to y.

- "Objects are equal iff each of their attributes are equal"
  - Essentially the "Deep Equality" test (equal or equalp) in Lisp
- IMHO "same" is one of the most dangerous words in the designer's lexicon

OBJECT MANAGEMENT GROUP

What to do?

- Designers should include application-level equality tests in their abstractions
  - System can't provide universal application equality test
- It means a round-trip invocation delay to establish equality
  - This will always be slower
  - But why should "equal" be different from "add"?
- Possible research topic to design controlled access to == test, so it can be used only where application programmer allows?
  - Full disclosure: my example objects lack updateable state

OBJECT MANAGEMENT GROUP

Argument 2: Conclusion

- Pointer comparison easy & efficient in non-distributed systems
  - Even though it's not clear what the semantics are
- Impossible to provide in realistic distributed environments
  - ... but non-distributed programmers continually ask for it
- Unrestricted pointer comparison breaks encapsulation
- Designers should include equality test as part of abstraction
  - As Java string designer did
- Access to == should be blocked to users
  - Though system implementers need to manipulate obj ref

OBJECT MANAGEMENT GROUP

End of lecture one

OBJECT MANAGEMENT GROUP

Argument the Third

Type compatibility

(and a few words on versioning)

OBJECT MANAGEMENT GROUP

What is a "type"? (1)

- Another interesting question to look up in textbooks
  - In non-OO languages, types classify data by representation
- For objects, encapsulation is a fundamental property
  - Separates interface from implemetation
  - To preserve encapsulation, two objects with same interface should be interchangable, regardless of how implemented
- So objects can have two, almost- orthogonal classifications
  - By impementation
  - By interface

OBJECT MANAGEMENT GROUP

What is a "type"? (2)

- In Smalltalk, implementation types are called "classes"
  - Objects from different classes can have same interface
  - Basis of inclusion polymorphism we find so useful
  - "If interface includes print, it's printable"
- For now, let's just talk about interface types
- Interface types can be defined in two ways
  - Extensional Type is a name for a collection of objects
  - Intensional Type is a predicate
  - Theoretically these are duals

OBJECT MANAGEMENT GROUP

Extensional view

- Typically used in Object databases
  - Assume complete knowledge of extension of a type
  - "Give me the complete set of objects with this property"
- Implicit closed world assumption
  - Impossible to implement in large distributed systems
  - Population may change faster than information about population can cross system

OBJECT MANAGEMENT GROUP

Intensional view

- Classically, define types as a predicates
  - For every type x, there exists a function:
    is_of_type_x(object) -> bool
  - Some OOLs with run-time types provide such predicates
- Since substitutability is the raison d'être for interface types, better formulation uses substitutabilty function
  - is_substitutable(object, object) -> bool
- If predicates make you happy, curry the function
  - Substitutability function has fewer concepts
  - Occam's Razor

OBJECT MANAGEMENT GROUP

Which substitutability relation?

- Strict equality
  - Not very helpful - no inclusion polymorphism
- Extension
  - Create subtype by extending base type's list of operations
  - Must not redefine any of the base type's operations
  - Advantages: Easy to implement and understand
  - Disadvantages: Still excludes some safe substitutions
  - Usally implemented via inheritance on interfaces

OBJECT MANAGEMENT GROUP

Aside - conformance

- Weakest (least restrictive) substitutability relationship to guarantee static type safety
  - Used in Emerald by Black et al
  - Not generally found in commercial statically-typed OOLs
- Informally defined via "no surprises" rule
  - Caller must not invoke any operation object doesn't support
  - Object must not return any exception caller doesn't handle
  - Apply recursively to parameter and result object references
- Conformance has a property called "contravariance"
  - Parameter types conform in opposite direction to results

OBJECT MANAGEMENT GROUP

A conformance example

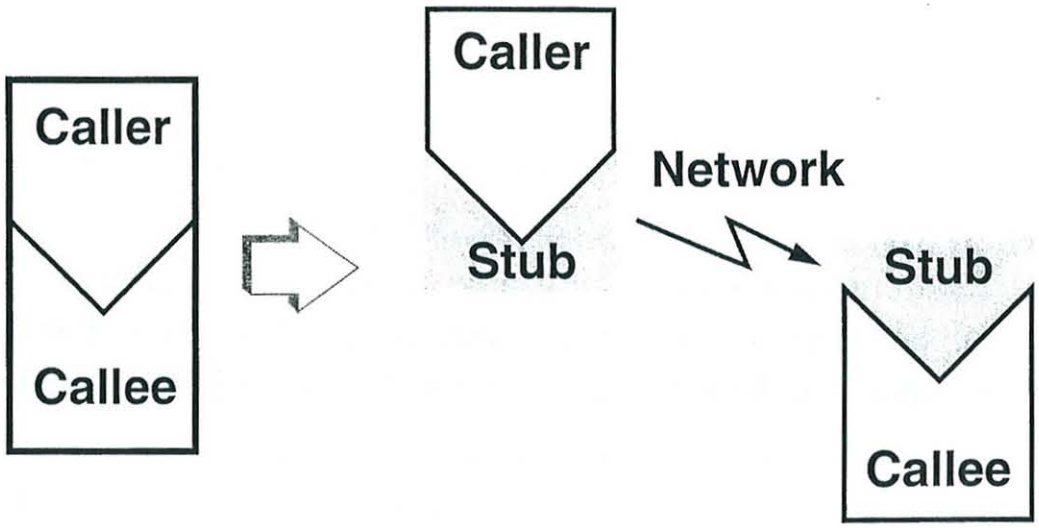
- Consider these types:

```
A = type (p(F):(Boolean))
B = type (p(G):(Boolean)
         q() :(Boolean))
F = type (r():())
G = type (r():()
         s():())
```

- Plainly, G conforms to F, but does B conform to A?
  - No: If object of type B substituted for object of type A, client may try to invoke p(F); object (of type B) thinks it has been passed parameter of type G, may try to invoke operation s

OBJECT MANAGEMENT GROUP

Back in the distributed world ... stubs



OBJECT MANAGEMENT GROUP

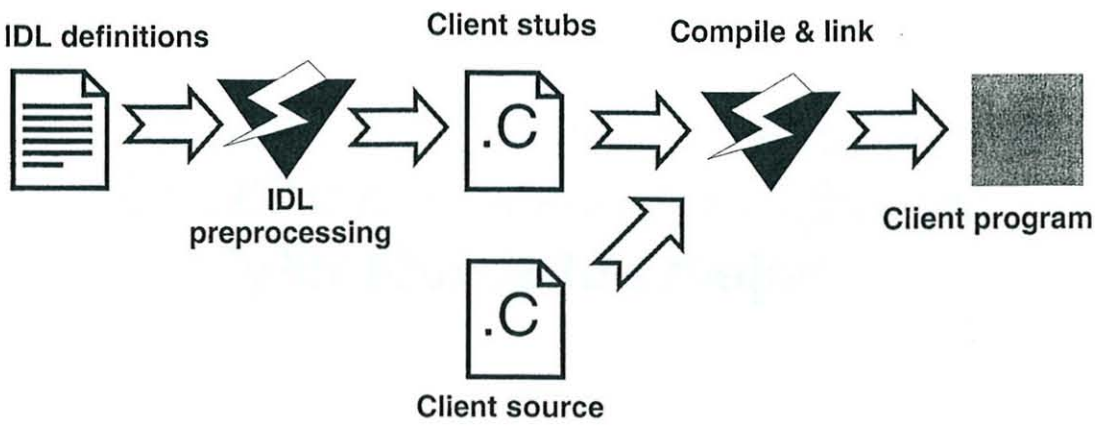
IDL defines interfaces

- Used to generate the stub (marshalling) code
  - "IDL compiler" generates marshalling code for particular language and patform
  - Provides language & data representation independence
- Compiled stubs linked with user code
  - IDL author does not control with exactly what code

```
interface example1
          { float op (in int arg1,
                      in string arg2)
          }
```

OBJECT MANAGEMENT GROUP

Using IDL



OBJECT MANAGEMENT GROUP

Inheritance

- Interfaces can be derived from other interfaces by extension
  - Rules are specified so this extension creates a subtype
  - An interface can be derived from multiple ancestors - but illegal if operation names conflict
  - CORBA calls this derivation "interface inheritance"

```
interface A {void f {in float x}}
interface B {long g {in long x}}
interface C: B, A {void h {in long x}}
```

  - Object of interface C can be used where client wants A or B

OBJECT MANAGEMENT GROUP

The question

Is interface inheritance a necessary, or merely a sufficient, condition for object substitutabilty?

OBJECT MANAGEMENT GROUP

Is this the right room for an argument? (1)

```
To: orbos@omg.org
Subject: Type Equivalence in CORBA
Date: Fri, 14 Jun 1996 14:29:07 +1000
From: Kerry Raymond <kerry@dstc.edu.au>

One of the issues that keeps cropping up on various OMG
mailing lists is the issue of equivalence of interface
definitions.

Look at the following examples and answer the question
"is X equivalent to Y?":

Example 1:

interface X { void A (); };
interface Y { void A (); };
```

OBJECT MANAGEMENT GROUP

Is this the right room for an argument? (2)

Example 2:

```
interface X { void A (); void B (); };
interface Y { void B (); void A (); };
```

Example 3:

```
interface Z { void A (); };
interface X : Z { void B (); };
interface Y { void A (); void B (); };
```

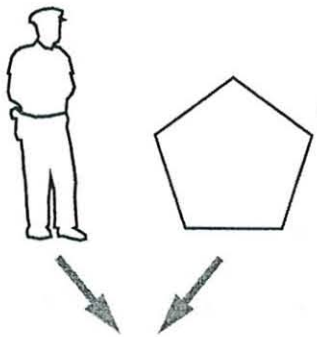

OBJECT MANAGEMENT GROUP

The discussion

- Many assert that inheritance is a necessary condition
  - i.e Objects whose interfaces appear compatible should not be substitutable unless one interface inherits from other
- Arguments seem to revolve around unwritten semantics that author associates with an interface with a particular name
  - If names & signatures should "accidentally" match, substitutability should (apparently) still be prevented
  - ... even though Smalltalk programmers have been doing this for decades
- "Accidental Conformance" argument

OBJECT MANAGEMENT GROUP

"Accidental Conformance" argument



interface Policeman

{ void draw ()
  void shoot (in victim x)
}

interface Polygon

{ void draw ()
  void move (in float x,
             (in float y)
}

?

interface Wombat

{ void draw () }

OBJECT MANAGEMENT GROUP

How do we find objects?

- "Accidental conformance" is an advantage, not a disadvantage
  - Allows more flexible versioning
- Could be a problem if we search for objects solely by type
  - An unbelievably silly idea
  - Types don't denote semantics in local code, why here?
- Traders, name servers locate objects by name or other criteria
  - Types tell us if invocation is safe, not if its meaningful
- CORBA IDL authors have no direct control over semantics of code linked with stubs generated from their IDL

OBJECT MANAGEMENT GROUP

Aside - type evolution and versioning

- Inclusion polymorphism provides some scope for evolution
  - New operations can be added to a server, but old clients needn't care
  - Contravariance (if available) permits altering parameter types in controlled ways
  - BUT semantics of the 'old' operations must be unchanged
- What if new version isn't a structural subtype?
  - i.e. lots of implementation code in common, but no substitutability between interfaces

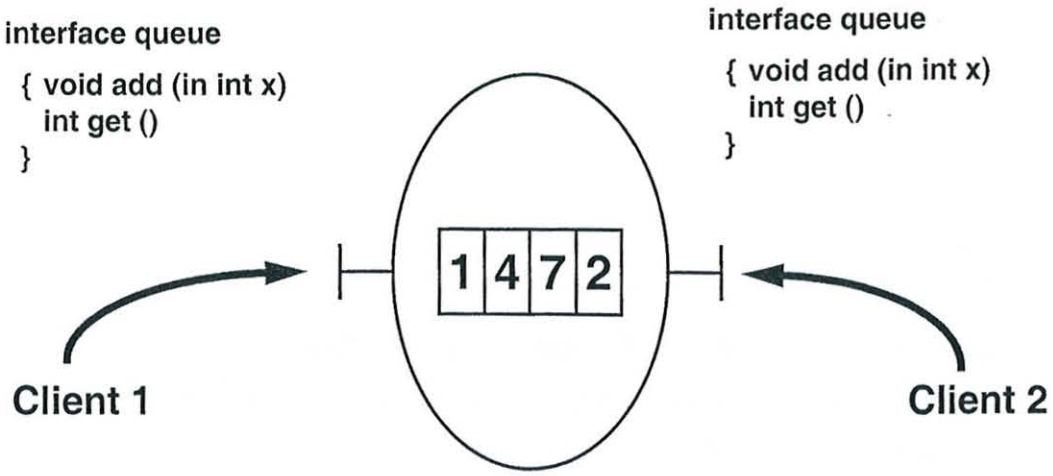
OBJECT MANAGEMENT GROUP

Multiple interfaces (1)

- Separate two functions of object
  - Encapsulation
  - Provision of abstraction
- Allows one object to present multiple abstractions
  - When they have unrelated types
  - ... or when they have operations with clashing names
  - ... or even when they are structurally identical (multiple interfaces of same type)

OBJECT MANAGEMENT GROUP

Multiple interfaces (2)



OBJECT MANAGEMENT GROUP

Multiple interfaces (3)

- We implemented it in full at ANSA in late 80s
  - Just call me smug
- Microsoft COM/DCOM has limited form
  - Only allows object to present one interface of each type
  - Used purely for versioning
- CORBA Component Model also now implements it
  - Again for versioning

OBJECT MANAGEMENT GROUP

Argument 3: Conclusion

- Structural type compatibility is anathema to many
  - They don't like it, even though they can't quite explain why
  - Wrapped up in the (ludicrous) belief that structure of an interface, interface names etc somehow encode semantics
- Some CORBA implementers apparently haven't noticed that spec forces them to implement structural compatibility
  - Client can sidestep stubs and dynamically build invocation
  - ... so can invoke any operation of that name on any object
- Most fiercely argued of the non-terminating arguments
  - And the most relevant one for building scalable systems

OBJECT MANAGEMENT GROUP

Overall Conclusion

- Things I wish people understood better before coming to OMG:
- Closed World Assumption
  - As touched on here
- End-to-end argument
  - Not discussed here
  - Read: J.H. Saltzer, D.P. Reed, D.D. Clark,
  - "End-to-End Arguments in System Design"
  - ACM Transactions on Computer Systems,
  - Vol 2, N. 4, p 277-288, Nov 1984
  - ... or at http://web.mit.edu/Saltzer/www/publications/

DISCUSSION

Rapporteur: Dr P D Ezhilchelvan.

Lecture One

While the speaker was presenting the Generic model, he pointed out that the methods can be aware of the concrete representation of the multiple objects on which they operate. Professor Sloman was of the view that representation can be hidden; the speaker indicated instances where methods can know the representation.

When the system property encapsulation was being discussed, Professor Henderson wondered whether too much flexibility is being admitted in assuming that the boundary of encapsulation can be known. The speaker replied in the affirmative.

When the speaker pointed out that permitting object sub-typing and multiple class inheritance increases the complexity of method selection, Mr Peine opined that the algorithm for method selection is in general complex, even in C++. Dr Waldo believed selection can be simplified by relying on 'globally unique' identifiers. The speaker refuted such a notion by saying that 'global uniqueness' exists only with 'closed world' assumption - a mentality that must be shed off in the OMG's sphere of activities.

As the differences between '==' and '=' were being discussed, Dr Waldo observed that the former is a system notion and the latter an object notion. The speaker reiterated his point that the application designers should be left with a clear idea of what equality means between objects.

Lecture Two

When the speaker was asserting that every new language that has emerged recently has C++ syntax, Professor Henderson observed that visual Basic is an exception.

While the speaker was dwelling on the question of finding the objects, he saw no apparent benefits in using types for that search. The reason was that types (names) say little about the semantics. Professor Randell had the impression that the names generally do carry much semantic information. The speaker explained that names carry semantic information – only small enough to help remember them, but not large enough to allow the entire semantics to be derived. Dr Thompson observed that if types were to reveal more information on the behaviour of the objects, this would compromise on the principle of encapsulation. The speaker emphasized the need to have good description of objects; he recalled interfaces being described in multiple spheres and here he is focusing only on type based descriptions which he believed could be enriched by experts. Dr Thompson likened the interfaces attached with descriptions of themselves, to proof carrying codes.

Dr Waldo referred to the example the speaker used to explain the problem of 'accidental conformance', and wondered whether the problem was truly as problematical as it was made out to be. The speaker put forward further convincing arguments and re-expressed his view that the issue of what an interface means should be addressed, perhaps treating interfaces as objects themselves. DrThompson wondered whether languages like Java permit interfaces to be treated as objects, and the speaker replied positively.

At the end of the discussions, Professor Randell sought the speaker's (by extension OMG's) view on the rule for sequencing of calls on methods. The speaker replied that this is one of the issues being investigated.