Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Type systems for distributed, object-based programming

Andrew Watson

Abstract

This briefing covers the fundamentals of conformance-based typing, as advocated by ANSA. It takes around 45 minutes, provided you don't dawdle, and was originally written to be presented to the OMG Technoical Committee - hence some of the examples are in pidgin IDL.

The slides are in colour, but should be legible if printed on a monochome printer.

APM.1148.01 **Approved** 8 February 1994
Briefing Note

Distribution:
Supersedes:
Superseded by:

Type systems for Distributed, Object Based programming

(and why you should care about them)

Andrew Watson

APM

ajw@ansa.co.uk

Talk objective
(or "Why am I telling you this?")

- Typing in OO systems is an oft-neglected and misunderstood topic
- Much of the literature is inaccessible, obscure or downright confused
- I'd like to introduce to you:
  - A self-consistent (and useful) view of types in distributed object systems
  - Some thoughts on why type checking helps
  - An insight into polymorphism
  - My dogma on inheritance
  - Relationship of static/dynamic typing to static/dynamic binding
  - Gaping holes in our current knowledge of typing polymorphic objects

What is a “type”?

- All sorts of definitions floating around
  - (and plenty of use of the term without definition at all!)
- A general definition is “categorisation by suitability for (some) purpose”
  - Procedural languages have concrete data types: purpose is to be passed to function that manipulates the representation
- In OO we divorce interface from implementation
  - If this encapsulation is to be preserved, users may only categorise objects by how they interact with them
  - Abstract data types
  - Categorisation by implementation also useful, but not to clients, and is orthogonal
  - “templates” ≈ ST80 “classes”

Categorisation by interaction

- **“Interaction” could cover a number of things**
  - **What information flows between client and server**
  - **How that information is represented**
  - **...**
- **One of the historical problems with OO type system work is this spectrum of meanings for “interaction”**
- **In this talk I shall be limiting myself to the narrowest possible definition of interaction, based on the names and parameter types of operations:**

A (computational) type is a list of the operations (including signatures) to which an object responds

Looks familiar?

- An IDL description of an object is a list of operations and signatures
- By my definition, an IDL description is a type
- IDL is a language for describing (computational) types

So what does this buy us?

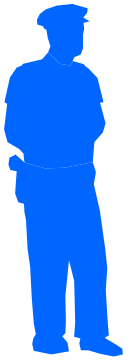
- **Comparison of the computational type of an object and the computational type desired by the client allows us to predict at bind time whether an interaction error is possible**
  - **Computational type** — **List of operation names and signatures. “Computational” because there could be (are!) other type classifications**
  - **Bind time** — **The point where the client acquires a reference to the service-providing object**
  - **Interaction error** — **“Method not understood”**
- **Corollary: IDL descriptions can be compiled into:**
  - **Stub code to package up and transmit parameters from client**
  - **Skeleton code to unpack transmitted parameters and call object method on client’s behalf**
- **That’s all**

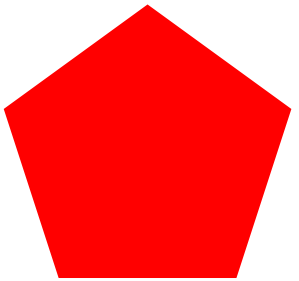
What it doesn’t buy us

- Doesn’t help answer the question “what is the extent of this type”
  - A computational type is not a set, it’s a description
  - A type no more lists all objects it describes than the colour Blue lists all blue things
  - Types are not “created”, but are implicitly defined by objects and clients
  - Existence of (IDL for) a client that needs to interact in a particular way defines a type just as much as the existence of an object that can interact in that way
- Doesn’t tell us anything about the implementation of an object
  - Such as the order of the methods in the dispatch table
- No notion of what an object represents or what an interaction “means”
- In short, doesn’t prevent building large, adaptable, heterogeneous distributed systems

“Accidental conformance problem”

```
interface Policeman
 {  void draw ()
    void shoot (in victim x)
 }
```



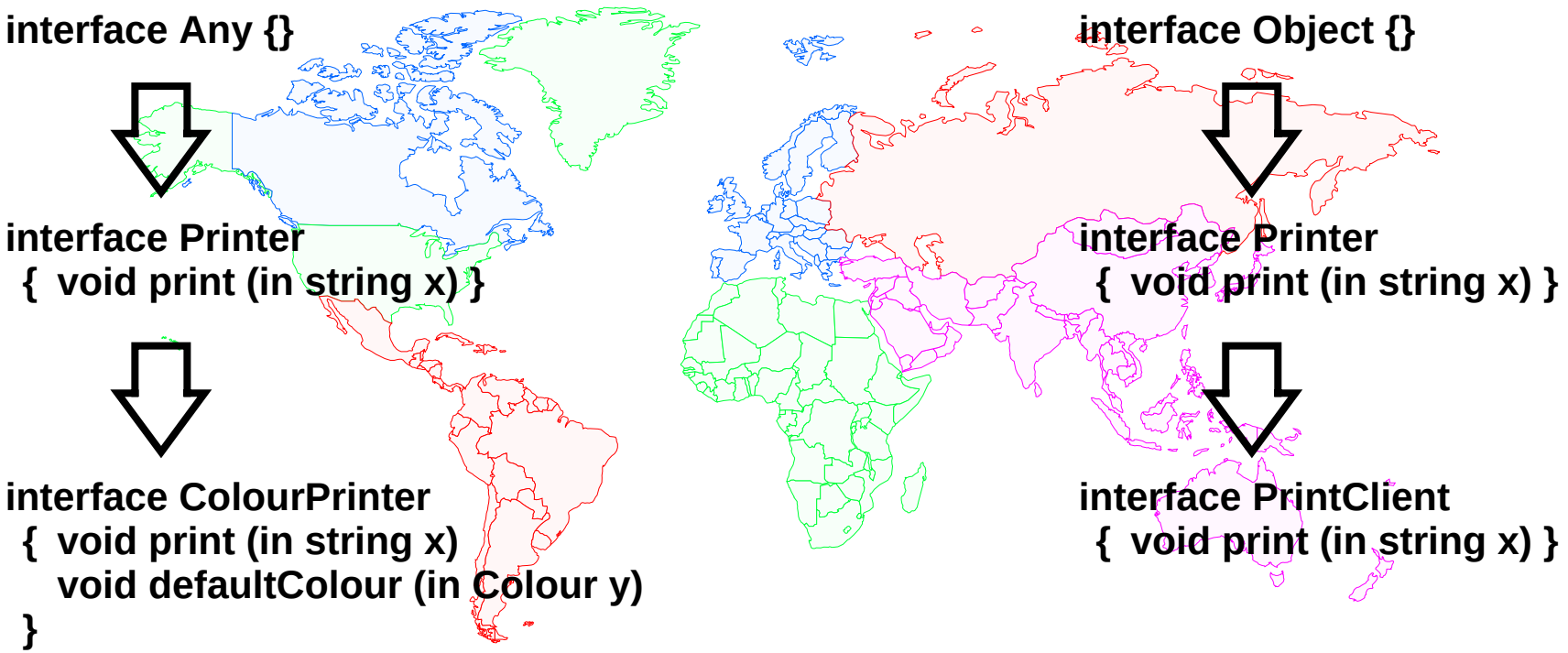


```
interface Polygon
 {  void draw ()
    void move (in float x,
               in float y)
 }
```

?

```
interface Wombat
 {  void draw ()}
```

Federation


interface Any {}
interface Printer
{ void print (in string x) }
interface ColourPrinter
{ void print (in string x)
void defaultColour (in Colour y)
}
interface Object {}
interface Printer
{ void print (in string x) }
interface PrintClient
{ void print (in string x) }

“Accidental conformance” & Federation

- Computational type doesn’t say what an object represents or what an interaction “means”
- “Accidental conformance” isn’t a problem, it’s the solution to a problem
  - It enables federation
  - On the information highways of the future, we must be able to advertise a service to clients we’ve never heard of, allowing them bind to it and interact with it
  - Must be able to merge previously isolated networks and have their applications interwork
- Other classifications of objects, according to ownership, information represented, etc., are needed
  - BUT they are separate & orthogonal
  - Simple type graphs are not rich enough for complex knowledge representation
  - Clyde the Royal Elephant

Accidental conformance and the OM

- **Computational type = OMG glossary “interface”**
- **OM defines type = interface + (unspecified) magic token**
  - **Magic token exists solely to allow objects with same interface to have different “types”**
  - **i.e. to prevent “accidental conformance”**

Type checking

- **Avoidance of interaction errors**
  - **Are there any invocations that this client might make which could cause that object (or others referenced) to issue “Method not understood”?**
  - **Is the type provided by the server substitutable for that expected by the client?**
  - **Of course, just because you can interact successfully with something doesn’t mean that it does what you expect**
- **Several substitutability relationships that fit the bill, including:**
  - **Same type - may only substitute object whose interface is the “same”**
  - **Extension (c.f. the CORBA inheritance relationship)**
  - **Conformance (no surprises)**
- **“Same interface” very safe, but obviates inclusion polymorphism**

Polymorphism

- The ability to treat different things as though they are the same
- A vitally useful technique used every day by every OO programmer
- Comes in (at least) two varieties
  - Inclusion polymorphism
  - Parametric polymorphism
- Inclusion polymorphism used every time an object is implicitly cast to a supertype
  - e.g. whenever an object is passed as a parameter to an operation that expects one with a subset of its operations
  - Pretty well understood
- Parametric polymorphism more complex

Subtyping by extension

- What CORBA IDL does
- Create a subtype by extending base type's list of operation signatures
  - Must not redefine any of the base type's operations
  - Only works in the absence of explicit self-reference (e.g. "self" keyword)
- Advantages
  - Easy to implement and understand
  - Provides some inclusion polymorphism
- Disadvantages
  - Too restrictive - some safe substitutions that are ruled out

IDL and the Inh*r*t*nce word

- CORBA IDL has a mechanism for deriving one IDL description from another
- Termed inheritance - unfortunately, probably the most abused word in OOP
  - See Cook *et al*'s paper "Inheritance is not subtyping"
- IDL inheritance certainly creates subtypes
  - Interface B inherits from Interface A is a sufficient condition for B to be a subtype of (substitutable for) A
- Is inheritance a necessary condition for CORBA subtype test to succeed?
  - Are CORBA implementations' invocation mechanisms sensitive to (for instance) signature order in interfaces?
  - Could use interface repository navigate operations to determine if one interface is an extension of another

Conformance

- **Conformance is the weakest (i.e. least restrictive) substitutability relationship which guarantees type safety**
- **Informally defined as the “no surprises” rule**
  - **Caller must not invoke any operation not supported by the service**
  - **Service must not return any exception not handled by the caller**
  - **Apply recursively to object references passed as parameters and results ...**
  - **Out and in/out parameters complicate matters (multiple results would be better)**
- **“Type A conforms to type B” written as A ⊳ B**
- **Conformance has a property called “contravariance”**
  - **Parameter types must conform in opposite direction to result types**
  - **“Controlling argument” view of OOP not really compatible with conformance**
  - **Conformance gives MI-style multiple supertypes for free**

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?**
  - **The answer is no**
  - **This is contravariance in action**
  - **If implementation of B substituted where object of type A expected, caller of p could pass in parameter of type F; implementation (thinking it has been passed something of type G) may try and invoke operation s**

A (possibly disturbing) conformance example

- **Consider these two types:**

```
A = type (equal(typeOfSelf):(Boolean)
          p():())

B = type (equal(typeOfSelf):(Boolean)
          p():()
          q():())
```

- **B seems to be an extension of A (in a system with inheritance, it might inherit from A), but it is not a subtype**
  - **If we substitute object of type B where one of type A expected, caller of equal may pass in parameter of type A, but object expects parameter of its own type (B), and attempts to invoke operation q**

Why is this disturbing?

- **Parametric polymorphism used in typing factories and collection types:**

```
interface SortableArrayFactory
 { ThisSortableArry create (in ElementType e)
      where
         ElementType posesses
            Boolean greaterThan(in ElementType x)
      and
         ThisSortableArray has
            interface
             { ElementType get (in Integer index);
               void set (in Integer i, in ElementType e);
               void sort();
             };
 };
```

- **Problem is specifying that ElementType “possesses” greaterThan operation**

The “contravariance problem”

- At first glance, the answer is to specify that SortableArray takes subtypes of:

```
interface HasGt { Boolean greaterThan(in HasGt x); }
```

- This doesn’t work, since the parameter type must be explicitly “self”
- Extending the type specification language to have *`typeOfSelf`* doesn’t help
  - Extension doesn’t create subtypes in the presence of self-reference
- This is sometimes referred to as the “contravariance problem”
  - Various more-or-less ad-hoc solutions have been proposed, but they miss the point
- This isn’t a “problem” - contravariance is telling us that these types are not substitutable
  - No amount of obfuscation of your language’s subtype rules will change this
  - “Covariance” is not a solution

So what's to do?

- There are two distinct situations where we want to express the idea that one type may be replaced by another
  - To allow server to provide a "richer" type than the client will need
  - To allow server to specify that it will only call a subset of the operations on a parameter passed to it with an operation
- There are experimental approaches to doing this, but they're not completely understood
  - F-bounded quantification
  - It's a research problem

Types and name services

- Name services associates names with object handles
  - Client can go to the service with a name, get back a handle
- Unfortunately, client has only informal assurances about computational type of the thing it gets back
- How to deal with this?
  - Ignore it (but have you ever tried debugging a Lisp program with a type error?)
  - Insert assertions manually (but how can you be sure you've got it right?)
  - Automatically check that service type conforms to client's expectations at bind time
- Third option the safest
- No time to go into details here ...

The Sobering Conclusion

Good News: OOP is a powerful, expressive programming technique, largely because of encapsulation and the consequent polymorphism

Bad News: No production OOL has decent automated type checking

Good News: Conformance-based type checking is well-understood, flexible and usable for building distributed OO applications

Bad News: There are useful kinds of polymorphism with which it can't cope

Good News: There are promising research technique that can

Bad News: It's only research, and precious few people are working on them

Bad News: Meanwhile, we're building a multi-million dollar industry on OOP