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

New Trader - Analysis and Design

Mike Beasley

Abstract

This paper contains the Object-Oriented Analysis and Design of a new modular trader whose pieces can be combined in various ways to build traders with different IDL/RPC and database mechanisms. It also includes some notes on the existing trader implementation.

APM.1110.00.02 **Draft** 14 February 1994
Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

New Trader - Analysis and Design

New Trader - Analysis and Design

Mike Beasley

APM.1110.00.02

14 February 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

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

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

3	**1**	**Introduction**
5	**2**	**Analysis**
5	2.1	Analysis - Object Model
7	**3**	**Existing Implementations**
7	3.1	Trader Interface
7	3.1.1	ANSAware version
8	3.1.2	BNR ITOM version
9	3.2	Context Interface
10	3.3	Type Interface
11	3.4	Federation Interface
11	3.5	Relocation
11	3.6	Type Management
12	3.7	Context Management
12	3.8	Offer Management
13	3.9	Property List and Constraint Program Management
13	3.10	Database Code
13	3.11	Thread Management etc.
13	3.11.1	Threads
14	3.11.2	Locks
14	3.11.3	Memory Allocation
14	3.11.4	Other things in the ANSAware version
14	3.12	Treatment of interface/object references
14	3.13	The main program
15	**4**	**Design**
15	4.1	General Principles
15	4.2	Class Diagram
15	4.3	Layers in the Class Diagram
16	4.4	Top Layer (External Interface)
16	4.4.1	Trader Interface
16	4.4.2	Context Interface
17	4.4.3	Type Interface
17	4.4.4	Federation Interface
17	4.5	Middle Layer (Management of Objects and Collections)
17	4.5.1	Type Management
18	4.5.2	Context Management
18	4.5.3	Offer Management
18	4.5.4	Property List and Constraint Program Management
19	4.6	Bottom Layer (Storage, Scheduling and Memory)
19	4.6.1	Database Code
21	4.6.2	Thread Management

22 4.6.3 Lock Management
22 4.6.4 Memory Allocation
23 4.6.5 Miscellaneous operations
23 4.6.6 Treatment of interface/object references
24 4.7 Outside any Class

1 Introduction

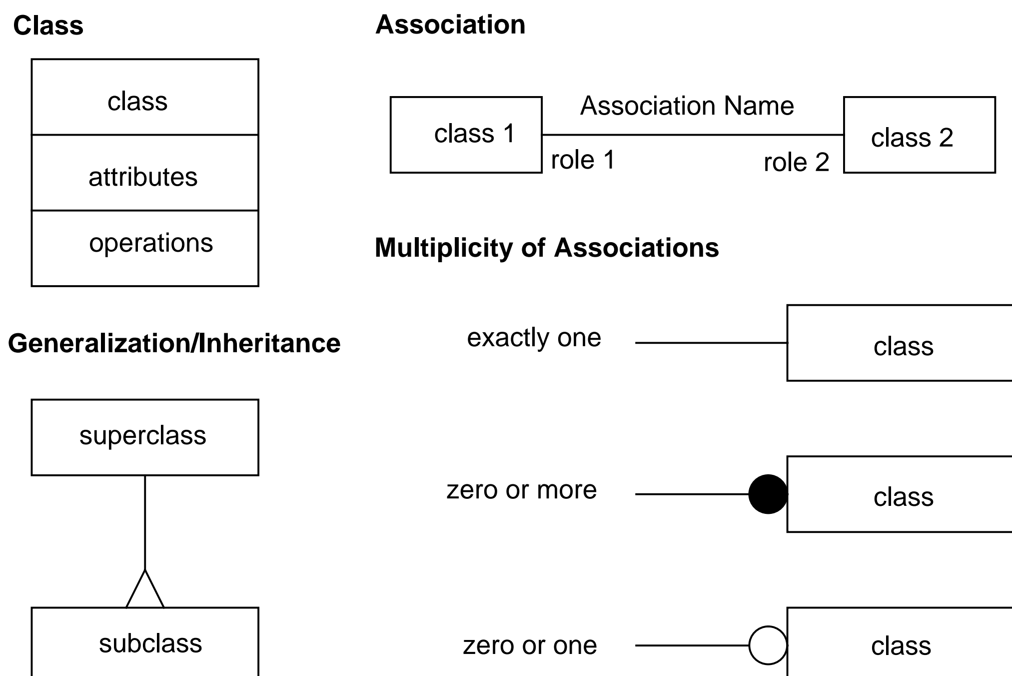
In [APM.1091.0 93] it is suggested that a new modular trader is needed as a starting point on which to build a more advanced trader which includes proper type checking and eventually develops into a Service Specification Repository.

This document develops such a design as follows:

- Chapter 2 presents an analysis, without going into excessive detail.
- Chapter 3 reviews existing trader implementations, to understand in more detail what functionality they provide, so as to design a new trader which is compatible with those that already exist.
- Chapter 4 presents the design.

The diagramming conventions used are those in [RUMBAUGH 91]. A summary of those diagramming features used in this document is in figure 1.1:

Figure 1.1: Diagramming Conventions



Note that some diagrams in this document have associations between a class and itself. In this case the straight lines shown for ‘association’ above end up as polygons or arcs of circles.

2 Analysis

2.1 Analysis - Object Model

Following Rumbaugh [RUMBAUGH 91] we construct an object model of the real-world system. The required classes are fairly clear:

- trader
- context
- type
- offer
- reference
- property

We then identify associations between these classes:

- a trader owns a number of contexts, and a number of types
- there are subtype/supertype and subcontext/supercontext relationships
- there are a multiplicity of offers of any type and in any context
- offers have one interface reference and a multiplicity of properties.

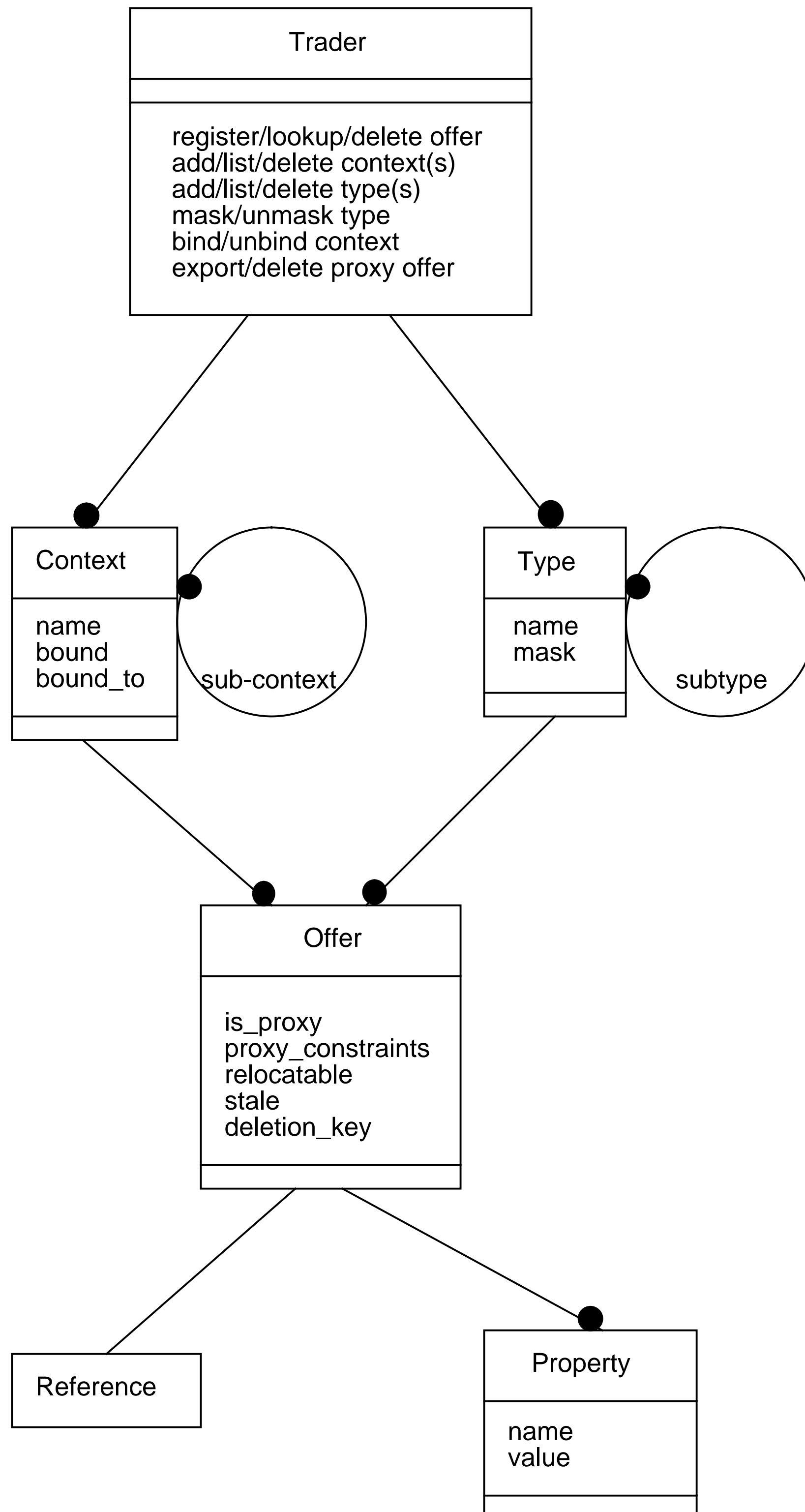
We also identify the attributes of objects.

The operations on the various objects are also identified. Strictly this is part of Rumbaugh's functional modelling.

Figure 2.1 shows this as a diagram in the style of Rumbaughi

The result of analysis is a model showing us what needs to be done, without any reference to how it is to be done.

Figure 2.1: Object Model - Analysis Phase



3 Existing Implementations

This chapter reviews two existing implementations of trading:

- the ANSAware trader, as described in [ARM 93]
- the trader as included in BNR's Integrated Trading and Object Management (ITOM) which formed the basis of a joint ICL/BNR submission to OMG [ICL 93].

Information based on reading the ANSAware code is also included.

The reasons for reviewing existing implementations are:

- to see what external interfaces the re-engineered trader will have to support
- to help to design the class hierarchy of the re-engineered trader
- to understand some of the areas of difference between existing traders which cause us to define abstract classes in the design.

3.1 Trader Interface

The basic operations in this interface are: `register`, `delete` and `lookup`. The interface to the ANSAware trader is defined in ANSAware IDL, and that to the ITOM trader in CORBA IDL as in the original documents.

3.1.1 ANSAware version

```
Trader : INTERFACE =

NEEDS Capsule;

BEGIN

--   non-primitive data types

     PropRecord : TYPE = RECORD [
      p_name: STRING,
      p_value: STRING
     ];

     PropRecords : TYPE = SEQUENCE OF PropRecord;

     IFRecord : TYPE = RECORD [
          ifr_ctxt : STRING,
          ifr_type: STRING,
          ifr_props: PropRecords,
          ifr_ref: ansa_InterfaceRef
     ];
```

```
        IFRecords : TYPE = SEQUENCE OF IFRecord;

        TStatus : TYPE = {F_S, S_S};

        TReason : TYPE = { unknownType, unknownContext, unknownId,
                  erroneousConstraint, floatingPointError,
                  noMatchingOffers, federationError,
                  unknownLookupPolicy, unknownReason };

        Result : TYPE = RECORD [
             r_success : BOOLEAN,
             r_reason : TReason
        ];

        LPolicy : TYPE = {Lookup_Random, Lookup_All};

        LResult : TYPE = CHOICE TStatus OF {
             F_S => TReason,
             S_S => IFRecords
        };

   --   operation signatures

        Register : OPERATION [InterfaceSpecificationName : STRING;
                  NamingContext : STRING;
                  PropList : STRING;
                  Ref : ansa_InterfaceRef;
                  Capsule: CapsuleRef]
              RETURNS [Result];

        Lookup : OPERATION [InterfaceSpecificationName : STRING;
                  NamingContext : STRING;
                  MatchingConstraints : STRING;
                  Policy: LPolicy]
              RETURNS [LResult];

        Delete : OPERATION [Ref : ansa_InterfaceRef;
                  MatchingConstraints : STRING]
              RETURNS [Result];

   END.
```

Note that the `Capsule` argument to the `Register` operation is present for historical reasons concerned with the Notification Service. This argument is therefore to be regarded as obsolescent, and should not appear in the new trader.

3.1.2 BNR ITOM version

In ITOM, trader = context (from a computational point of view). BNR therefore have a `resolve` operation to go from one trader (context) to another. The IDL definition of the ITOM trading service goes like this:

```
interface TradingService: CommonTypes {
     typedef sequence<Object> Objects;

     // Registering an offer returns a ScopedId which
     // identifies the offer
     ScopedId register_offer(
          in Object          object_ref,
          in Type            object_type,
          in PropertyList    properties,
          in RelativeName    context
     ) raises (UNKNOWN_RELATIVE, UNKNOWN_TYPE);

     void unregister (
          in ScopedId        offer_id,
          in RelativeName    context
     ) raises (UNKNOWN_ID, UNKNOWN_RELATIVE);

     Objects lookup (
          in Type            object_type,
          in ConstraintExpr  constraints,
          in RelativeName    context,
          in PolicyList      search_preferences,
     ) raises (UNKNOWN_RELATIVE, UNKNOWN_TYPE, INVALID_POLICY);
};
```

The ITOM interface differs from the ANSAware version above in having the `resolve` operation and in splitting up `constraints` argument into two parts: `constraints`, which has no superlatives, and `preferences`.

3.2 Context Interface

The basic operations in this interface are: `add name`, `list names`, `delete names`. The ANSAware IDL definition is:

```
TrCtxt : INTERFACE =

BEGIN

--   non-primitive data types

     CtxtStatus : TYPE = { CAdded, CDeleted, CNotEmpty,
CNoSuchContext };

     CtxtRec : TYPE = RECORD [
          ctr_name : STRING,
          ctr_bound : BOOLEAN
     ];

     CtxtList : TYPE = SEQUENCE OF CtxtRec;
```

```
--   operation signatures

     AddName : OPERATION [ ContextName : STRING ]
           RETURNS [ CtxtStatus ];

     ListNames : OPERATION [] RETURNS [ CtxtList ];

     DelName : OPERATION [ ContextName : STRING ]
           RETURNS [ CtxtStatus ];

END.
```

3.3 Type Interface

This interface has operations: `add`, `mask`, `unmask`, `delete`, `list`, **defined in ANSAware IDL as follows:**

```
TrType : INTERFACE =

BEGIN

--   non-primitive data types

     TypeStatus : TYPE = { TAdded, TDeleted, TAlreadyExists,
                TNoSuchSuperType, TNoSuchType, TOffersExist,
                TIsASuperType, TMasked, TAlreadyMasked,
                TUnmasked, TAlreadyUnmasked, TInvalidType };

     Vector : TYPE = SEQUENCE OF STRING;

     TypeRec : TYPE = RECORD [
          tyr_name : STRING,
          tyr_sups : Vector,
          tyr_mask : BOOLEAN
     ];

     TypeList : TYPE = SEQUENCE OF TypeRec;

--   operation signatures

     AddType : OPERATION [ TypeName : STRING;
                SuperTypes : Vector ]
           RETURNS [ TypeStatus ];

     MaskType : OPERATION [ TypeName : STRING ] RETURNS [
TypeStatus ];

     UnmaskType : OPERATION [ TypeName : STRING ] RETURNS [
TypeStatus ];

     DelType : OPERATION [ TypeName : STRING ] RETURNS [
TypeStatus ];

     ListTypes : OPERATION [] RETURNS [ TypeList ];

END.
```

3.4 Federation Interface

The operations are: `bind context`, `unbind context`, `proxy export`, `delete proxy` and the ANSAware IDL definition is:

```
TrFed : INTERFACE =

NEEDS Capsule;

BEGIN

--   non-primitive data types

--   operation signatures

     BindContext : OPERATION [
          NamingContext : STRING;
          Ref : ansa_InterfaceRef
     ] RETURNS [ BOOLEAN ];

     UnbindContext : OPERATION [
          NamingContext : STRING
     ] RETURNS [ BOOLEAN ];

     ProxyExport : OPERATION [
          ansa_InterfaceSpecificationName : STRING;
          NamingContext : STRING;
          PropList : STRING;
          Constraints : STRING;
          Ref : ansa_InterfaceRef;
          Capsule : CapsuleRef
     ] RETURNS [ BOOLEAN ];

     DeleteProxy : OPERATION [
          Ref : ansa_InterfaceRef;
          Contraints : STRING
     ] RETURNS [ BOOLEAN ];

END.
```

As for the trading interface (see section 3.1.1), the `Capsule` argument on the `ProxyExport` operation should be regarded as obsolescent and not carried forward into a new trader implementation.

3.5 Relocation

In ANSAware, the relocator is co-located with the trader in the engineering, but is not computationally a part of it.

For details of the ANSAware relocator, which is implemented in the `reloc` library, see the manual page `Reloc(3)`.

3.6 Type Management

In the existing ANSAware trader, a type is represented by a node in a symbol table:

```
typedef struct supertype {
     struct supertype *st_next;
     char *st_name;
} SuperType;

typedef struct typename {
     char *tn_name;
     short tn_mark;
     short tn_mask;
     struct supertype *tn_super;
} TypeName;
```

The `tn_mark` field is used internally to mark what has been processed. There is various code around which uses `g*t*` in conjunction with a strange syntactic construct, known as a 'label', in which a name is immediately followed by a colon. Such programming practices have no place in a new trader !

3.7 Context Management

A context in the current ANSAware trader is represented as follows:

```
typedef struct cxname {
     struct cxname *cxn_next;/* context sibling */
     int cxn_bind; /* bound or not */
     ansa_InterfaceRef cxn_bref;/* i/f ref to bound trader */
     char *cxn_name;/* context name */
     struct cxname *cxn_cdsc;/* context descendants */
     struct ifname *cxn_idsc;/* interface (offer) descendants */
} CXName;
```

3.8 Offer Management

An offer in the current ANSAware trader is represented as follows:

```
typedef struct ifname {
     struct ifname *ifn_next;/* IF sibling */
     ansa_Boolean ifn_fed;/* proxy or not */
     char *ifn_ctxt;/* full naming context */
     char *ifn_spec;/* IF specification name */
     char *ifn_cstr;/* constraint string for proxy */
     ansa_InterfaceRef ifn_expt;/* export stamp for offer */
     ansa_InterfaceRef ifn_capsule;/* capsule making offer */
     ansa_Boolean ifn_notify; /* register with NS or not */
     ansa_Boolean ifn_relocatable;/* set if relocatable */
     ansa_Boolean ifn_stale; /* set if potentially stale */
     ansa_Cardinal ifn_delkey;/* key for stale offer deletion */
      /* 0 means not subject to deletion. */
     Property *ifn_prop;/* property list for offer */
     ansa_NotifyId ifn_obid;/* notify id of object posting offer
*/
} IFName;
```

The fields `ifn_notify` and `ifn_obid` are obsolescent, because they are present only for reasons concerned with the notification service, and the `Capsule` argument to the trader's `Register` operation is likely to be removed in the near future - see the ANSAware manual page `Trader(3)`.

3.9 Property List and Constraint Program Management

There is a considerable amount of code in the existing trader to analyse property lists and constraint programs. This includes a `yacc` grammar to analyse constraint expressions, and code to perform constraint matching and superlative tests on properties.

3.10 Database Code

The current implementation uses the `checkpt` library, as already mentioned. This library is documented in a manual page.

Reading the whole trader database is done by `TableInit` in `tblinit.c`. This effectively applies updates (starting from nothing) to get to where we want to be.

Writing is handled by `checkpoint`, also in `tblinit.c`. This writes out types, contexts and offers using the following functions:

- types: `dumpTypeST`, in `typesubs.c`. The function `PrintSingleType` deals with one type. In `dumpTypeST` there is code which ensures that supertypes are output before any of their subtypes.
- contexts: `PrintContexts`, in `namesubs.dpl`. This also uses `PrintContextNode`, `PrintSingleContext` and `PrintBoundContext`.
- offers: `PrintOffers`, in `namesubs.dpl`. This uses `PrintOfferNode` and `PrintSingleOffer`.

The following are written to the update file:

Table 3.1: The Update File

update	text	arguments
add type	addtype	type name, list of supertype names
delete type	deltype	type name
mask type	mask	type name
unmask type	unmask	type name
add context	addname	context name
delete context	delname	context name
register offer	register	type, context, ifref, notify, (capsule), relocatable, ifproxy, (constraints), properties
withdraw offer	delete	ifref, ifproxy, constraints
bind fed ctxt	bind	context, ifref
unbind fed ctxt	unbind	context

There is some code to convert string forms of interface references into 'the real thing' in the ANSAware trader; this is in the function `chkpt_rIfRef`.

3.11 Thread Management etc.

3.11.1 Threads

The ANSAware trader uses a timer thread. This exists because of the way the `checkpt` library works.

3.11.2 Locks

The ANSAware version has `check`, but this is currently unused and is for diagnostic purposes only (it checks that a lock has been grabbed and freed equal numbers of times).

The implementation uses event counts and sequencers. Note that the underlying implementation uses a lock which is implemented by straight memory reference; this is OK because thread scheduling is non-pre-emptive. Other threading mechanisms may have different characteristics.

3.11.3 Memory Allocation

Memory allocation is quite complicated in the existing ANSAware version, as there can be three sorts of allocation: stub-based, thread-based and normal (the difference is concerned with whether and when an automatic `free` is done). These use `system_allocate`, `thread_allocateMem` or `stub_allocateMem`, but only one corresponding `free` function - stub and thread memory are freed automatically when they are known to be no longer required.

3.11.4 Other things in the ANSAware version

There are various macros like `ansa_Abort`, `ansa_Assert` and `ansa_Warn` in `include/capsule/capmacs.h`. Suitable abstractions need to be produced.

3.12 Treatment of interface/object references

ANSAware has a number of functions for dealing with interface references:

- `ifref_copyRef` (copy a reference)
- `ifref_format` (convert to a string)
- `ifref_freeRef` (free the memory occupied by an interface reference)
- `irgram_parse_Ref` (convert a string to an interface reference)

It also has a ‘compare identity’ function which, despite the theoretical arguments against such things, we may have to include.

3.13 The main program

ANSAware requires that the main program is called `body` rather than `main`, because the name `main` is pre-empted by the library.

4 Design

4.1 General Principles

The general principles underlying the design are as follows:

- separation of concerns: keep things separate when that is appropriate.
- abstraction: cope with differences (e.g. between ANSAware interface references and Orbix object references) by defining a virtual class which has the common features, and use derived classes to implement the differences.
- classes should not have public data members. This makes it easier to distribute things later, if that is thought desirable.
- modularisation: identify a structure of classes encompassing all of the code.

At implementation time (assuming C++), we should:

- write all code as member functions, unless this is impossible (e.g. `main`).
- not use the `friend` mechanism unless there is a very clear justification.

4.2 Class Diagram

Figure 4.1 shows the class diagram as in the analysis phase, with the addition of ‘interface’ and ‘manager’ classes, omitting operations and attributes to keep the diagram more manageable.

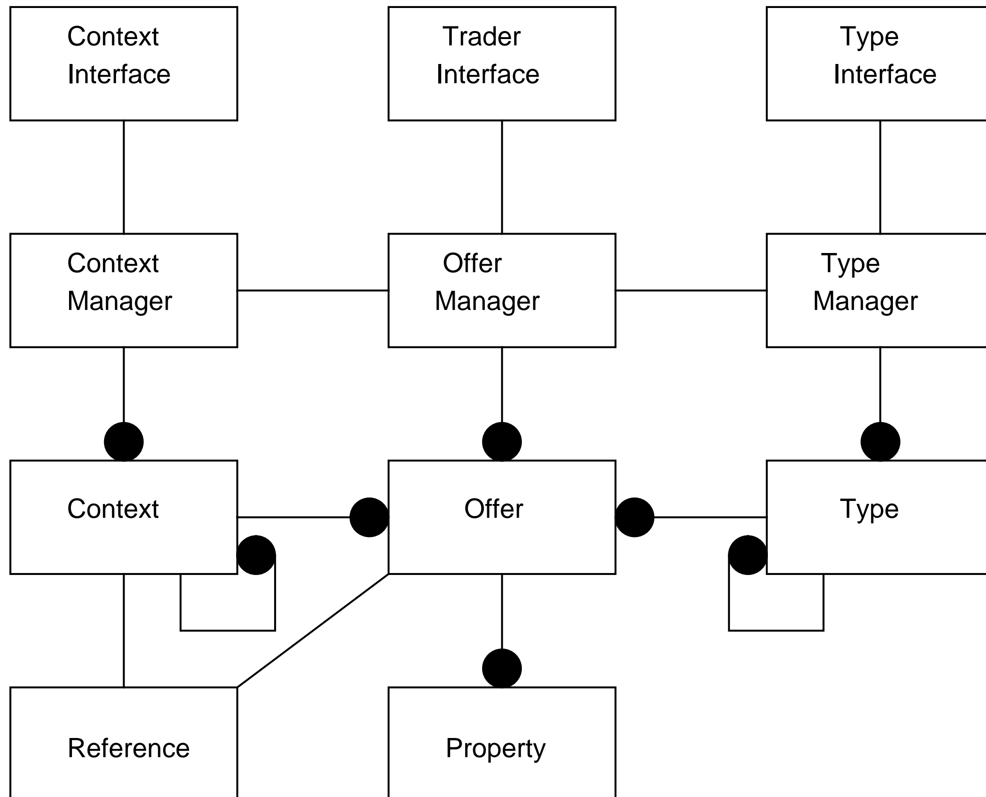
The operations and attributes for each class are included in the figures which follow.

4.3 Layers in the Class Diagram

The classes making up the trader naturally split into three layers:

- classes related to the external interface. These classes will naturally depend on what external interface we need to support (for example, the classes required in an ANSAware trader will be different from those required to support an ITOM trader.
- classes concerned with managing collections of objects (types, contexts and offers) and with individual objects. This layer contains the core trader functionality which is independent of the interface to the outside world and of the storage, scheduling and memory allocation mechanisms.
- classes concerned with storage, scheduling and memory allocation.

Figure 4.1: Class Diagram



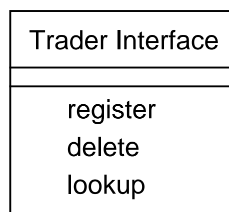
4.4 Top Layer (External Interface)

As stated above, the details of the classes in this layer will depend on what external interface we are supporting. However, certain generalisations can be made about operations and their arguments.

4.4.1 Trader Interface

Figure 4.2 shows the Trader Interface class.

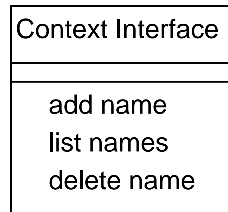
Figure 4.2: Trader Interface



4.4.2 Context Interface

Figure 4.3 shows the Context Interface class.

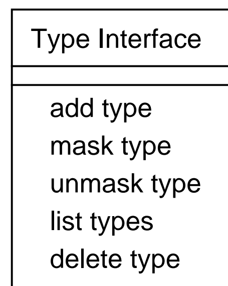
Figure 4.3: Context Interface



4.4.3 Type Interface

Figure 4.4 shows the Type Interface class.

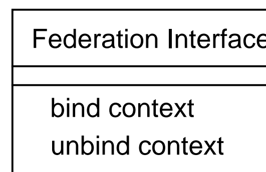
Figure 4.4: Type Interface



4.4.4 Federation Interface

Figure 4.5 shows the federation interface. The registration and deletion of

Figure 4.5: Federation Interface



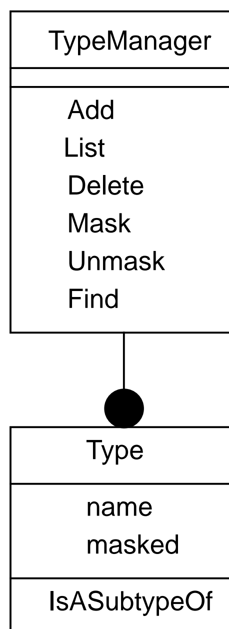
proxy offers and normal offers have much in common, and no differences other than the need to know/remember whether the offer is a proxy or not. It therefore makes no sense to separate them, and so the federation interface consists only of the `bind context` and `unbind context` operations.

4.5 Middle Layer (Management of Objects and Collections)

4.5.1 Type Management

There will be classes `TypeManager` and `Type` as shown in figure 4.6.

Figure 4.6: TypeManager and Type



Symbol table manipulation is an obvious candidate for implementation as an auxiliary class, with operations `lookup`, `enter` and `delete`.

4.5.2 Context Management

There will be classes `ContextManager` and `Context` as shown in figure 4.7.

We may well want to have an auxiliary class to handle splitting names into components.

4.5.3 Offer Management

The classes `OfferManager` and `Offer` are shown in figure 4.8. Proxy offers are distinguished from real offers by an extra boolean argument to the `Add` operation.

Offers will not contain type names; instead, they will contain pointers to types or pointers to interface/object references which refer to types (the latter allows for the type manager not to be co-located with the trader; the former would be the appropriate choice for a co-located implementation).

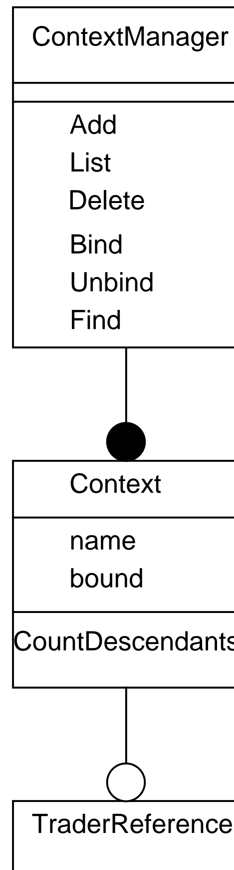
Random Number Generation (used in `Lookup Random`) will be a separate auxiliary class.

4.5.4 Property List and Constraint Program Management

The `yacc` grammar from the existing ANSAware implementation will be retained; this analyses property lists and constraint programs, and performs matching operations on them. It will be encapsulated into a class.

There will be classes for property lists,individual properties and constraint programs - see figures 4.9 and 4.10.

Figure 4.7: ContextManager and Context



4.6 Bottom Layer (Storage, Scheduling and Memory)

4.6.1 Database Code

The database operations which we will need in a trader are these:

- read in the whole of the trader data at startup
- update the database to record any individual update to the trader's internal data, as it happens
- write out the whole of the trader data, to synchronise, at regular intervals and at closedown.

The last of these operations would probably be null in an SQL-based operation, whereas the existing ANSAware trader implementation, using the `checkpt` library, records updates in a temporary update log, and writes out a new file at regular intervals and at closedown.

We will use the abstraction principle here in our design: a virtual class will support an abstract concept of database access, and derived concrete classes will implement the abstraction in terms of various database mechanisms. Examples of derived classes will be:

- ANSAware, using the `checkpt` library as in the existing trader.
- SQL database.

Figure 4.8: OfferManager and Offer

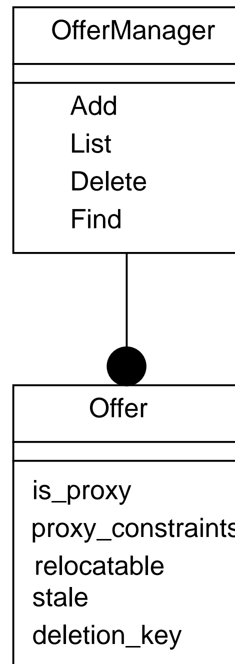
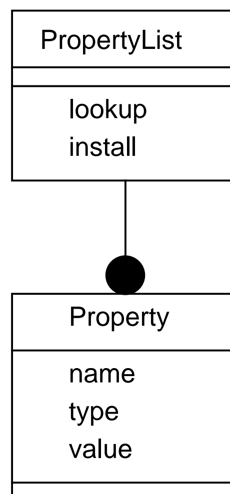


Figure 4.9: PropertyList and Property

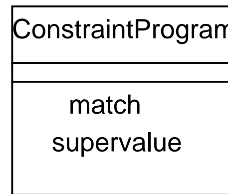


There are two ways in which we might attempt to design the class structure. The problem is that the database access code needs to know about two things: the structure of the objects being written, and how to access the database itself.

The two approaches to the design are:

1. 'Objects write themselves'. This idea fits in well with ANSA approaches to (e.g.) management and security, and superficially seems attractive. We would have a generalised class `PersistentObject`, with:

Figure 4.10: ConstraintList



- — a member function to write an image of itself as a string of bytes, appending this to a buffer and updating a pointer ready for the next object
- — a constructor which takes a buffer and a pointer, and initialises the object, updating the pointer ready for the next object to be initialised

Classes such as `Type`, `Context` and `Offer` would inherit from this class, redefining the above functions. This approach has been used successfully in past projects.

2. Alternatively we might have a `DatabaseManager` class, with member functions to:
   - — read all data (this function will be called at trader startup, and will read the database, calling the various object managers to create the initial set of objects)
   - — write out the current state of all data (this function will be called periodically during running, and at the end; some database managers may not require such a function)
   - — record an update (various kinds on various objects). These functions will be called by the various object managers when they have performed an update, and will update the database to reflect the internal state.

The second alternative seems more likely to provide the solution in the present situation; if we merely wanted to be able to write persistent objects to a file and read them back again, the first approach would be good, but writing to an SQL database (or even the existing ANSAware trader's persistent file) requires the structure of the data to be maintained.

The class `DatabaseManager` is shown in figure 4.11.

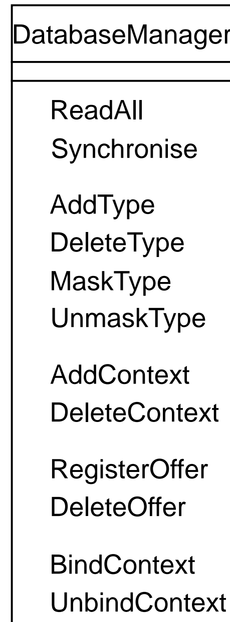
4.6.2 Thread Management

The general approach to management of such resources as threads, memory and locks will be to define abstractions which are implemented by means of virtual classes like `ThreadHandler`, `Lock` and `MemoryAllocator`, and to inherit from these.

So for threads, there will be derived classes for ANSAware threads, Sun lwp, DCE threads, and possibly others. Note that a timer thread will be required in an ANSAware implementation using the `checkpt` library, but not in an implementation using an SQL database.

Note: TBS - details of attributes/operations for thread handling class.

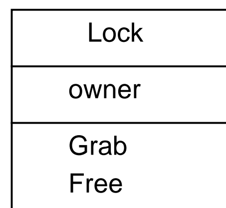
Figure 4.11: DatabaseManager



4.6.3 Lock Management

The `Lock` class is shown in figure 4.12.

Figure 4.12: Lock



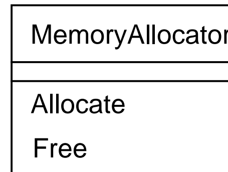
Other private data is implementation-dependent. For example, there may be a need for semaphores - this is dependent on the threading mechanism, and in particular on the thread scheduling.

The `owner` string is present for debugging purposes - it enables a check to be made that the freeing of the lock is done using the same string as in the `Grab` operation.

4.6.4 Memory Allocation

The existing ANSAware trader has to make use of three types of memory allocation: stub-based, thread-based and normal. There will be a single abstraction, with `allocate` and `free` operations; the ANSAware implementation will have three different concrete classes derived from the same abstract class. See figure 4.13 for the abstract class.

Figure 4.13: MemoryAllocator

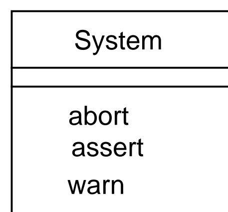


The above shows the design, but the implementation in C++ will probably not actually involve a class called `MemoryAllocator` at all ! All memory allocation in C++ should use the operators `new` and `delete`; you can declare your own operator functions, either global or for a particular class, and do your own memory allocation. The memory allocation operator `new` can be made to take an argument (in our case, of an enumeration type with values `normal`, `thread` and `stub`). It can then allocate the memory as required. Deletion is no problem - any memory which is deleted explicitly can be assumed to have been allocated normally.

4.6.5 Miscellaneous operations

Various miscellaneous operations will be required: currently known examples are: `abort`, `assert` and `warn`. This set will be encapsulated in a `system` class - see figure 4.14; others may be more appropriately placed in other classes.

Figure 4.14: System



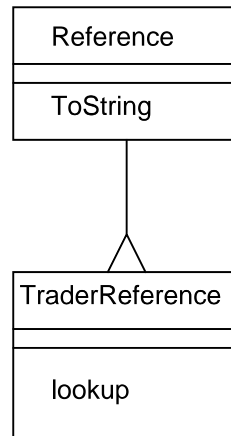
4.6.6 Treatment of interface/object references

There are two types of interface/object references manipulated by the trader:

- general interface references being traded
- interface references to federated traders

Following the principle of abstraction, we have a virtual class for general references, and a derived virtual class for references to federated traders. These virtual classes have no operations on the internals of references; the ‘trader reference’ class adds operations such as `lookup` to the general one. Both these classes have specialisations (for ANSAware interface references and for Orbix object references, for example). The outline of the `Reference` classes is shown in figure 4.15.

Figure 4.15: Reference



4.7 Outside any Class

There will need to be a main program, which will be extremely RPC-dependent (for example, ANSAware requires that the main program is called `body` rather than `main`).

There should be no other code that is outside a class.

References

[APM.1091.0 93]

Beasley, M.D.R., “Advanced Trading”, Architecture Projects Management, Cambridge, 1993.

[ARM 93]

The ANSAware 4.0 manual set, Architecture Projects Management, Cambridge, 1993.

[ICL 93]

“ICL Proposal in Response to OMG Object Services RFP 1”, OMG document number 93-2-10, ICL and BNR Europe, 1993.

[RUMBAUGH 91]

Rumbaugh, J., et al., “Object-Oriented Modeling and Design”, Prentice Hall, 1991.