Object Orientation and Interoperability

William Kent

Database Technology Department, Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, California 94303 USA

1 Introduction

Barriers between various sorts of environments in computational systems are being breached. Independently developed applications need to use each other's services and share each other's data despite a host of differences in the ways they were developed and the ways they operate.

Object orientation offers some tools for solving such problems. Ironically, though, object orientation also introduces interoperability problems of its own. Our examination of object orientation and interoperability will sketch the problems and solutions, focusing in particular on object-oriented database.

2 Dimensions of the Interoperability Problem

Interoperating applications are often developed independently of each other in environments that may differ in the following dimensions:

- Locations.
- Machine architectures.
- Operating systems.
- Programming languages.
- Models of information.

Applications can interoperate along the following dimensions:

- "Horizontal" peer-to-peer sharing of services and information, such as an editor invoking a spreadsheet processor to embed a spreadsheet in a document.
- "Vertical" cascading through levels of implementation. A student registration service may use a database service which in turn uses a file manager which uses a device driver.
- "Time-line" through the life cycle of an application. Enterprise modeling may be done in terms of one set of constructs which are translated into constructs of the application programming language which are compiled into constructs of the run-time environment. Or, a graphical language used to capture a user's conceptual model of a business domain is translated into a computer-executable simulation language, with the results of the simulation then being input either to an analysis tool to allow refinement of the simulation, or to a report generator to produce the final result.

Internal Accession Date Only

- Others, e.g., the “viewpoints” of the ISO/CCITT Reference Model for Open Distributed Processing (RM-ODP) [16]:
  - Enterprise viewpoint
  - Information viewpoint
  - Computational viewpoint
  - Engineering viewpoint
  - Technology viewpoint

Interoperation is concerned with such things as:

- Application interconnection:
  - Finding services and information in a distributed environment.
  - Coping with operational differences between requesters and providers of services, such as interface/communication protocols, synchronization, exception handling, work coordination, resource management, etc.
- Information compatibility.

3 Application Interconnection

Problems of application interconnection are addressed by distributed application architectures such as the Common Object Request Broker Architecture (CORBA), shown in Figure 1, being developed by the Object Management Group [13, 14, 15].

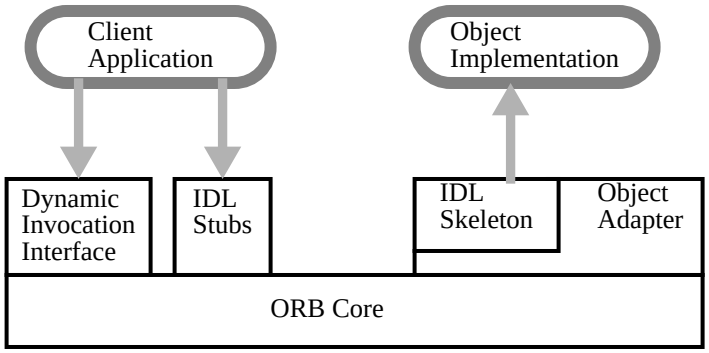

Figure 1. CORBA.

Requests are submitted via a stub or the dynamic invocation interface and interpreted via an object adapter, a skeleton, and an implementation (performer). The ORB Core itself supports a performer model [Section 5.2] defined by its Interface Definition Language (IDL) [15]. Mapping of models is possible on two sides:

- Applications are expected to invoke local routines to generate ORB requests in IDL. Such routines could map various capabilities into IDL form, supported by appropriate interpretations in the application compiler, adapter, skeleton, and implementation to which the requests are routed.

- Specific mechanisms of how an implementation is chosen and activated are determined by specific implementations of ORB cores, adapters, and skeletons.

4 Information Compatibility

4.1 Problems

Problems of information compatibility arise in the following dimensions (many are illustrated in [7]):

- Different data models:
  - Traditional database models: relational, hierarchical, network.
  - File systems.

    Files often don't even have a schema of their own. File format descriptions are buried in applications, allowing different applications to have different views of the same file.
  - Object-oriented models (there are many! [Section 5]).
  - New forms: complex structures, spatial data, text, hypertext, blobs, multimedia.
- Differences in enterprise views of similar domains, reflected in data and/or process differences. These might represent local differences in policies or business rules.
  - At the schema level:

    Some businesses have several salesman per territory, others give several territories to one salesman.

    Some schools have teaching assistants, others don't. TeachingAssistant is a subtype of Student at some schools, a subtype of GraduateStudent at other schools.
  - Differences in abstraction level/granularity:

    One source has motor vehicles, the other has cars, trucks, buses, motorcycles, etc.
  - At the instance level:

    Sales territories are partitioned differently at different companies.

    Different schools offer different courses.
- Different schemas for the same domain, even within the same model and enterprise view:
  - Different names or spellings for schema elements.
  - Single vs. multiple elements for addresses, dates, names, etc.
  - Cross-level mappings:

    Jobs are data values in some sources, distinct tables or subtypes in others.
  - Other structural differences.
- Data element (representation) differences:
  - Data types, unit of measure, precision.

  - Notational differences such as spelling, punctuation, capitalization, or omitted parts in names, addresses, document citations, etc.
- Discrepancies in information values:
  - Errors.
  - Out of date information (different currency levels).
  - Different opinions, e.g., ratings of movies or restaurants.
  - Subtle unspecified differences in definition, e.g., distances as airline miles vs. driving miles.

4.2 A Framework

We focus on peer-to-peer interoperation, and particularly on data sharing, i.e., making data from multiple sources available to an application/user.

A general framework for integrating the information in multiple data sources, such as described in [12], involves zero or more of the following components:

- External sources.
- Exported schema of each external data source.
- Imported schema of each source in a common model.
- Locally managed data.
- Unified schemas, reconciling discrepancies of all sorts.
- End user views, possibly in various models.

A sample framework is shown in Figure 2.

There are many variations on the basic framework, with no clear winners emerging yet. Different situations require different approaches, and it is not clear whether any one paradigm will dominate. Some variations:

- Framework architecture: there might be zero or several of some components.
- Characteristics of external sources:
  - Degree of autonomy.
  - Possible presence of an application serving as an “agent” of the importing system.
  - Degree of heterogeneity.
  - Distribution (physical location and connectivity).
  - Variations in services available, e.g., queries and transactions.
- Inter-component processing:
  - Manual vs. automated import.
  - Query processing/optimization strategies.
  - Transaction management.
  - Propagation of updates, in both directions.

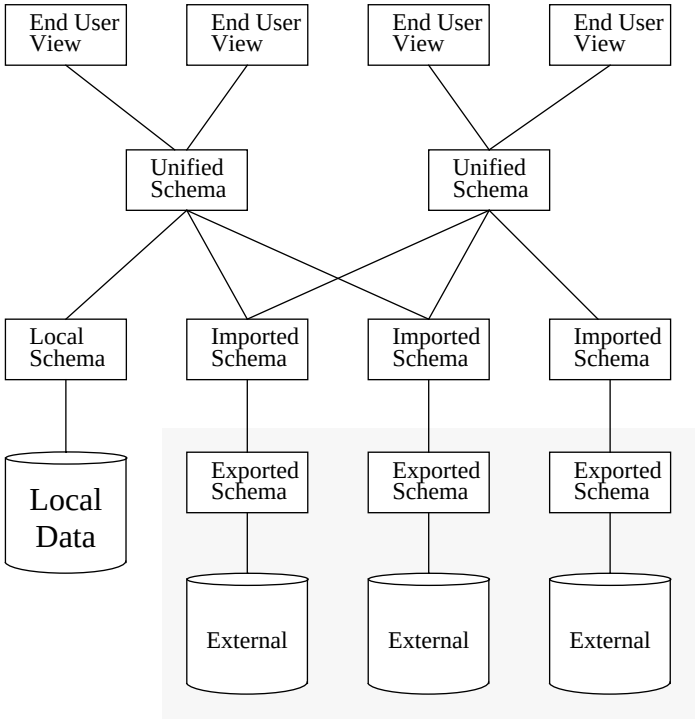

Figure 2. Integration framework.

- Integration scope:
  - Across multiple enterprises (least feasible).
  - Across one whole enterprise.
  - Localized application domain (most feasible).
- When the framework is developed:
  - Statically in advance (not practical for large scope).
  - Incrementally as needed, accumulating an expanding framework. (Both of those have to cope with schema evolution.)
  - Ad hoc: as needed, then discarded.
- End user capabilities:
  - Read-only vs. update of external sources.
  - Current vs. snapshot view of external data.
  - Integration with locally maintained data.

We will focus on information mapping concerns.

4.3 Object Orientation as a Solution

Certain OO features are particularly useful for solving problems of information compatibility:

- Object identity provides a basis for correlating information about "the same thing" from different sources.
- Subtyping allows reconciliation of different levels of abstraction.
- Computationally complete behavior specification (beyond relational view definition) allows complex resolution of information discrepancies.
- The notion of overloaded operators can be extended to help correlate object identities and reconcile information discrepancies.

Note that we are describing language for expressing solutions, not facilities for discovering them. Solutions will be illustrated in terms of a participant model [Section 5.2] in order to use its higher level of semantic abstraction [2, 5].

4.3.1 Stages

Access to external data in multiple sources involves:

- Discovering/identifying relevant external sources.
- Importing their schemas into a common model.

  After importing into a common model, the "home" system has a schema of the external data while the external sources serve as underlying "storage managers" for the data. At this stage the imported data from multiple sources is in a common form accessible by end users, but without identities correlated or discrepancies reconciled. The schema is the simple union of the imported schemas, possibly along with schemas for locally maintained data as well.
- Integrating into a unified view:
  - Integrate the schemas.
  - Correlate object identity.
  - Reconcile discrepancies in information content.
- Executing user requests.

We will focus on import and integration, which are the main areas to which object-oriented techniques can be usefully applied. Import and integration correspond to "levels" in the schema, and need not correspond to a chronological sequence.

Schemas don't have to be statically pre-defined or persistently maintained [Section 4.2]. However, the application/user does need some coherent picture of the information being used. At minimum, what we describe here is a mental model, or some implicit descriptive mechanism transiently created at run time. Renaming (mentioned below) doesn't have to be explicitly specified, but could be inferred from knowledge of synonyms and variant spellings. Of course, the more heuristic the more the risk of error. Algorithms for correlating identities and reconciling information do need to be expressed somewhere.

4.3.2 Import

Suppose we are accessing information about students, teachers, and courses from various universities. One of the university sources might export a relational schema as shown in Figure 3.

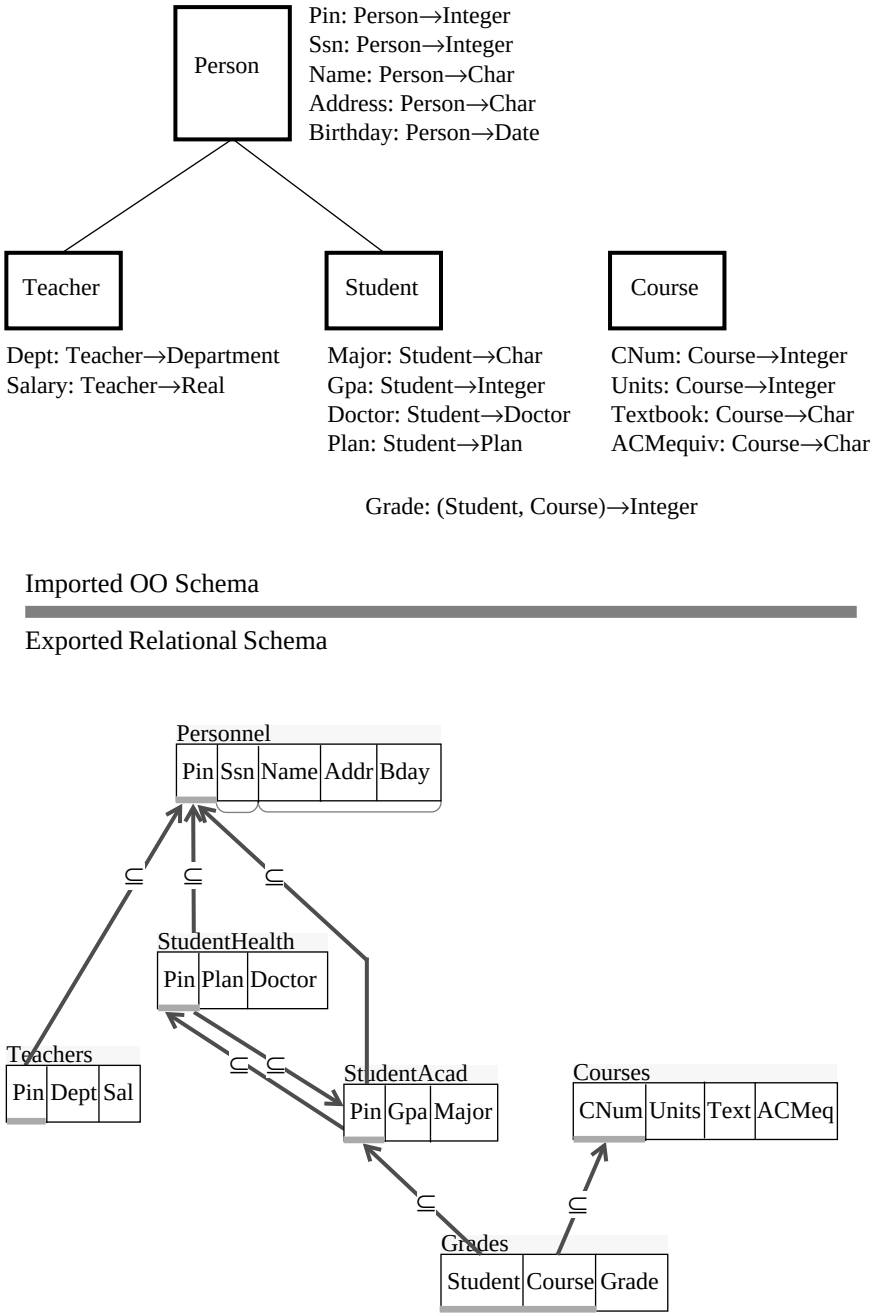

Figure 3. University data example.

Type relationships can be inferred from certain inclusion dependencies [3] (indicated as $\subseteq$ in Figure 3). Simple key inclusions imply subtypes, so the Teacher type is imported as a subtype of Person. Mutual key inclusions imply vertical partitioning, so StudentHealth and StudentAcad are recognized as providing data about the same objects, and hence are imported as a single Student type (also a subtype of Person). Courses are imported as a distinct type. Some foreign keys imply that relations represent relationships rather than types, and are imported into whatever mechanism represents relationships in the object model. Thus the Grades relation is imported as a multi-argument function in a functional object model.

The university uses its own "person identification numbers" (Pin) to identify people, since not everybody has a social security number. In the external relational data, a person corresponds to a Pin value, which is an integer. In the OO model, a person has an identity as a distinct object, separate from the integer Pin value which is one of its properties. Put simply, an instance of Person is not an integer.

In a locally managed OODBMS, objects are explicitly created and deleted. An explicit object creation operation provides a semantic basis for object identity: distinct creation events give rise to distinct objects. (In some implementations, creation generates a unique oid for the object as a "birthmark"). An explicit deletion operation allows the DBMS to manage referential integrity, removing all stored references to the deleted object (or disallowing the deletion while there is such stored data). Such facilities are not available for data imported from fully autonomous external sources. The external data will simply contain different numbers of students at different times, with the OODBMS getting no intervening notification of creation or deletion.

Since there are no creation events on which to base object identity, such identity must be based on data in the external source. Essentially, a set of values has to be defined such that an instance of the imported type exists for each value in the set. We can say that an imported type "produces" an instance corresponding to each value in its "producer set". The producer set might be simply defined, as with the Pin numbers used by this university to identify people. In the absence of such simple keys, it might be something more complex, such as a composite key consisting of Name, Address, and Birthday. (In general, imported object types don't have to correspond to primary keys of relations.)

As mentioned, the values in the producer set are not the same things as the instances of the type. Although Pin integer values are not themselves instances of Person, there is a one-to-one correspondence. Whenever a certain Pin value is detected in the external data, a corresponding Person instance exists. When the Pin value vanishes, the corresponding Person instance ceases to exist. If the Pin value reappears, so does the Person instance. Such "produced objects" have their existence defined by a rule, rather than by creation events (like the "imaginary objects" of [1]).

Object identity has to be carefully managed among different types. Common values in different producer sets may or may not correspond to the same object. In the university example, course numbers (CNum values) might accidentally match Pin values, yet courses and persons are obviously not the same thing. Furthermore, several universities might use Pin numbers independently. Different students at different universities might have the same Pin number — and a student who has attended several universities will probably have a different Pin number at each.

A conservative strategy is to assume that, in the absence of other information, all imported types are disjoint from each other, i.e., they have no instances in common. A plausible implementation strategy would be to construct an object identifier as a concatenation of a producer value with a unique "tag" associated with each imported type.

The disjointness assumption can be relaxed, for example, if inclusion dependencies are known for a relational data source [3]. The disjointness assumption is clearly inapplicable in the case of vertical partitioning, since the underlying relations are imported into a single type. The assumption is also inappropriate for subtypes, as indicated by key inclusion dependencies. It does not make sense to treat Teacher or Student as being disjoint from the Person supertype, since the instances of Teacher or Student are obviously known to be instances of Person. Thus, in implementation terms, these three types should all use the same tag in their produced oid's. (Note that we have no information as to whether or not Teacher and Student are disjoint from each other.)

Naming conventions generally require types to be uniquely named, so types imported from different sources might have to be renamed (e.g., if they are imported from external relations having the same name). Thus, if we are importing data from "Eastern University" and "Western University", the imported types might be named EPerson, WPerson, ETeacher, WTeacher, and so on.

The imported functions (operations) only have to be uniquely named within their argument types, since object systems support overloading (polymorphism). However, we should carefully distinguish between specific and generic functions. Thus, for example, the functions named "Address" which are defined on EPerson and WPerson are two distinct specific functions, corresponding to a single generic function named "Address". Specific functions can be uniquely named by appending their argument types, e.g., EPerson.Address and WPerson.Address.

We will later see that it is very useful to rename functions so that they have the same name if and only if they represent "semantically equivalent" information — a concept which we will not define further. Thus, for example, all functions which provide addresses should be named "Address", and no other function should have that name.

If no further integration is defined, imported data can be made accessible to end users uniformly via the object model of the importing OODBMS. However, it is uncorrelated. Objects imported from different sources appear distinct, so that a person who has attended several universities appears to be several distinct persons. Courses given by one university are different objects from those given at another. The schemas associated with the different sources might be different (though we will assume they are similar for the moment).

For retrieval purposes, imported data can be indistinguishable from locally maintained data. Update may be restricted, depending on the level of interaction supported by the autonomous external source.

4.3.3 Schema Integration

Schema integration is a problem when imported schemas from "similar" domains are structurally different, yet need to be presented as a single schema to the end user [4]. A typical problem is illustrated in Figure 4.

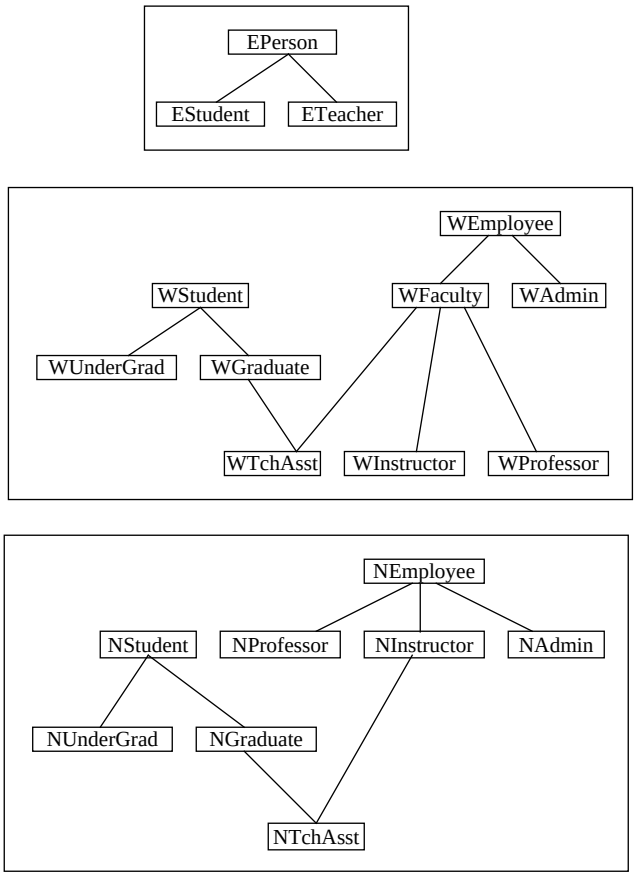

Figure 4. Non-isomorphic imported schemas.

Object technology is generally useful here to the extent that it is capable of representing various levels of abstraction in a type hierarchy. However, the specific problem of unifying these into a coherent schema for the end user is not in itself facilitated by object oriented techniques. We will not pursue this problem in this paper.

4.3.4 Correlating Identities

Correlating identities requires a mechanism for establishing that several imported things should be treated as being the same object. The problem is simpler when the things being made equal are imported from different sources, but we will later consider cases where things imported from the *same* source should be considered equal ("localized equivalence") [Section 4.3.6]. In the absence of localized equivalence, identity correlation can be described in terms of a "single-viewpoint" approach, whereby the equal things are immediately treated as the same object on import. Localized equivalence requires a more complex description, involving an "underlying viewpoint" in which the imported things are distinct objects, and an "integrated viewpoint" in which they appear as one.

Care need be taken to preserve the semantics of equality when treating things as being the same [8]. In the single-viewpoint approach, x=y does not have the sense of two objects which become

one object. It simply means that the variables x and y refer to one and the same object (whether or not they are implemented as having the same oid).

Equality is an *equivalence relation*, hence the following axioms hold:

E1: $\forall x$, x=x (reflexivity).

E2: $\forall x \forall y$, x=y $\Rightarrow$ y=x (symmetry).

E3: $\forall x \forall y \forall z$, x=y & y=z $\Rightarrow$ x=z (transitivity).

Equality implies *substitutability*:

E4: $\forall x \forall y \forall f$, x=y $\Rightarrow$ f(x)=f(y);
also, f(x) and f(y) cause the same conditions and side effects.

Equality implies *singularity*, i.e., there is just one object:

E5: $\forall x \forall y$, x=y $\Rightarrow$ {x,y}={x}, or equivalently, Card({x,y})=1.

E5 might not be an independent axiom, depending on how set construction is defined in terms of equal elements. But it doesn't hurt to emphasize it here.

A typical equality specification might take the form

EPerson.Ssn(x)=WPerson.Ssn(y) $\Rightarrow$ x=y.

An equality specification essentially involves a binary predicate e(x,y) such that

$\forall x \forall y$, e(x,y) $\Rightarrow$ x=y

is added to the list of equality axioms. Note that this is one-way implication, not if-and-only-if.

Oid-based object systems have an implicit axiom that x=y if they have the same oid. In general, x=y is true if it can be deduced from the full set of equality axioms.

Equality is independent of type. An instance of ETeacher might be equal to an instance of WTeacher because of a rule defined on EPerson and WPerson, or even because of a rule defined on EStudent and WStudent if some teachers were also students.

Criteria for judging whether things are the same object can be quite complex, involving various sorts of heuristics, which we will not investigate here. What we will describe are language facilities for expressing equality once the criteria have been determined.

Equality of objects can be described in terms of matching values of some property. The property might be a simple one such as social security number, or a complex one such as the combination of name, address, and birthday. The property might be an artificial one expressly provided for this purpose; for example, a course can be labeled with its equivalent in the ACM/IEEE standard computer science curriculum, with this property then used for establishing identity.

There can be multiple disjunctive criteria, e.g., persons are the same if they have the same social security number (Ssn) or if they have the same citizenship and passport number (Ppn) (not illustrated in Figure 3). Equality criteria often constitute a one-way implication rather than an if-and-only-if condition. Thus persons should be considered equal if they have the same Ssn (even if they have different Ppn's) *or* if they have the same citizenship and Ppn (even if they have different Ssn's).

The general form of an equality specification is thus based on equality of properties, e.g., $f(x)=g(y) \Rightarrow x=y$. It really only makes sense to specify equality based on "semantically equivalent" properties, e.g., comparing social security numbers with social security numbers, or passport numbers with passport numbers. If semantically equivalent properties are given the same name, then equality specifications can be described as uniqueness constraints on generic functions.

A uniqueness constraint means that $f(x)=f(y) \Rightarrow x=y$, i.e., distinct things can't have the same value of f. Such constraints are typically specified for single specific functions, e.g., EPerson.Ssn is unique-valued over all instances of EPerson, so that distinct instances of EPerson cannot have the same value of Ssn.

Declaring generic functions to be unique-valued extends the constraint across all the types on which the function is defined. Declaring Ssn to be a globally unique generic function essentially says that things having the same value of Ssn, via any of the specific functions named Ssn, must be the same object, i.e.,

$$T_i.Ssn(x)=T_j.Ssn(y) \Rightarrow x=y$$

for each of the types T_i and T_j on which Ssn is defined.

4.3.5 Reconciling Discrepancies in Information Content

The essential problem can be described by saying that there are functions which the end user wishes to use whose values may not be consistent in the various external sources. Again, we deal only with facilities for expressing solutions, not for discovering them.

Typically, when only one of the external sources provides a value, this can be taken as the intended result. When there is more than one source, then various strategies might be appropriate:

- If all the sources provide the same value, use it.
- Apply some aggregating operation, such as sum, average, minimum, maximum, etc.
- Establish a priority ranking of the sources, returning the value from the most reliable source.
- If the information is dated, return the most recent value.
- Apply conversion routines to reconcile differences in units, precision, or other representation characteristics.
- Return all the values from the various sources, perhaps annotated as to the source of each. This option is only viable if the function is defined to return results of the appropriate structure.
- Other arbitrary procedure.

It should be obvious that none of these is the natural default for all cases. Computationally complete object-oriented languages generally have the power to express such algorithms. What remains to be defined are mechanisms for specifying and invoking the action to be taken.

If we again assume that semantically equivalent functions have the same name, this problem bears a remarkable resemblance to overloaded function resolution. The situations where discrepancies might arise correspond exactly to the cases where an overloaded function call is ambiguous.

As before, let's assume we are importing data from several university sources such as Eastern and Western. A request such as Salary(x) is an invocation of an overloaded function, involving specific Salary functions defined on ETeacher, WTeacher, etc. If x is not an instance of any of these types, we have a type violation. If x is an instance of exactly one of these types, then the value of the corresponding specific function should be returned, e.g., ETeacher.Salary(x). Otherwise, if x is an instance of several of these types, the overloaded function call is ambiguous and should be treated as an error condition — unless a disambiguating procedure has been provided. (An ambiguity exists when more than one specific function is relevant, even if they all yield the same value.) It is not at all self-evident what the disambiguation ought to be. If the sources are providing data about different universities, it may be appropriate to return the sum of the salaries (even if they happen to be equal). If the sources are different databases containing data about the same university, it might be appropriate to take the average, or to use the source considered most reliable, or to use the most recent value if the dates are known.

A fairly natural approach here is to allow explicit definition of generic functions, providing a mechanism for defining the disambiguating algorithm. This also provides a mechanism for declaring uniqueness of generic functions [Section 4.3.4]. Such a definition might, for example, specify that a user-defined function named BestSal should be invoked to return an appropriate result whenever Salary(x) is ambiguous.

4.3.6 Localized Equivalence

Localized equivalence arises when several things from the same source are to be treated as equal. Most often, these things will be instances of the same imported type. The situation often arises when there is an independently defined set of instances to be seen by the end user, and imported things have to be mapped into this set of instances.

As an example, suppose that courses are to be considered equal if they correspond to the same course in the standard ACM/IEEE curriculum, and a university offers several courses having the same ACM equivalent. All those courses should appear to be the same course to the end user. The same situation might arise if courses were to be considered equal if they used the same textbook. It might also arise if courses are to be mapped into the local university's own set of courses.

Similar localized equivalence might occur in connection with job titles, geographic entities, colors, and other such things.

Formally, localized equivalence can arise when:

- Equality specifications are based on non-unique properties, such as the ACM equivalent or textbook of a course.
- There are cycles among equality specifications. Suppose an equality specification correlates EPerson and WPerson by social number, and another correlates them by nationality and passport number. Then, if an instance of ETeacher had the same social security number as one instance of WTeacher and the same nationality and passport number as a different instance of WTeacher, transitivity would force those instances of WTeacher to be equal.

Localized equivalence makes it difficult to even formulate the problem of information discrepancy. It can no longer be handled directly by the mechanisms of overloaded functions.

As an example, let's say that when courses are considered equal, we want Units of a course to be presented to the end user as the average of the Units in the external data sources. Without localized equivalence, a characteristic problem would be that different specific functions had different values for the same argument, e.g., ECourse.Units(x)≠WCourse.Units(x). We could resolve that with a disambiguator for the overloaded Units function, which would take the average of the values when Units(x) was invoked. The unreconciled values could be seen by the end user by invoking the specific functions ECourse.Units(x) or WCourse.Units(x).

With localized equivalence, we somehow need to have x and y be instances of the same type (say ECourse) with different values of ECourse.Units, and yet also have x=y. This simply doesn't work. Under the substitutability principle [Section 4.3.4], it is not possible for ECourse.Units(x) to be different from ECourse.Units(y) if x=y, since they are logically one and the same object.

This problem requires the introduction of two distinct "viewpoints" (or "spheres" [9]). In an *underlying* sphere, x and y appear to be distinct objects, allowing the possibility of ECourse.Units(x)≠–ECourse.Units(y). This sphere should only be visible to an administrator responsible for defining the reconciliation mechanisms. The end user should only see an *integrated* sphere in which x=y and there is only one reconciled value of ECourse.Units(x).

An equality specification e(x,y) now implies that x=y in the integrated sphere, but not necessarily in the underlying sphere. Under these conditions, a type may have fewer instances in the integrated sphere than in the underlying sphere, since x and y count as one object in the former but two in the latter. Logically speaking, an instance in the integrated sphere corresponds to an equivalence class of instances in the underlying sphere.

Disambiguators for overloaded functions dealt with the values of several specific functions for a given argument. Here we need the complementary notion, having to unify the values of a given specific function for several arguments. That is, we have to unify the values of ECourse.Units(x) and ECourse.Units(y).

The solution is similar to the one for disambiguation of overloaded functions, though now it has to span between the integrated and underlying spheres. A "unifier" function could be defined for ECourse.Units, such that an invocation of ECourse.Units(x) in the integrated sphere would apply this unifier to the set of values of ECourse.Units(x_i), where the x_i are the distinct objects in the underlying sphere which are equivalent to x.

5 Object Orientation is Part of the Problem

Object orientation is plagued by a diversity of interpretations. There is a variety of object oriented models. Readers will probably have noticed different models assumed in different lectures at this Institute. Various experts have very clear and self-consistent ideas, but they are not consistent with each other. Hence there is a new problem of interoperability of object oriented facilities.

5.1 Common Concepts

There seems to be a fairly common core of concepts shared by most OO models [6, 13, 14], but even these are likely to stir some discussion among experts:

- Identity.

- Behavior (messages/methods/operations).
- State.
- Types/interfaces.
- Subtyping.
- Inheritance.
- Overloaded/polymorphic operations.
- Encapsulation.

5.2 Performer and Participant Models

Perhaps the most fundamental distinction between object models is whether objects are perceived as performers or participants. Suppose that a student counseling service (human or automated) wishes to register Sam in the Algebra course. The user might perceive this as a request for a Register operation in which Sam and Algebra *participate*, perhaps written as Register(Sam,Algebra). The requested service might be *performed* by an object corresponding to Sam, by an object corresponding to Algebra, by a Registrar application object, or directly by a DBMS object which records Sam's registration in Algebra. The requester might not even know that the Registrar or DBMS objects exist.

The "participants" in the user's request are Sam and Algebra. The "performer" might be Sam, Algebra, the Registrar, or the DBMS. Thus, in general, the performer may or may not be one of the participants mentioned in the request. Sometimes the application programming language imposes a "performer-oriented" syntax, e.g., Sam.Register(Algebra) or Algebra.Register(Sam), but even then the actual performer might be something else, i.e., the Registrar or the DBMS rather than Sam or Algebra. Similarly, a user at a graphical interface might drag a document icon to a trashcan icon (the two participants), causing the service to be performed by a file manager object.

Some facility in the system may have to transform a user's request of the form Register(Sam,Algebra) into a performer request such as Registrar.Register(Sam,Algebra) or DBMS.Register(Sam,Algebra). This may even cascade to another level, which recognizes different copies of the Registrar application at different sites as being distinct objects, and selects one copy to service the request. There are thus at least two object models involved in the registration scenario, requiring a cascading of interfaces [13, 14] to map between them.

A participant model does not distinguish between performers and participants, while a performer model does. The distinction is relative to a particular request; a performer for one request might simply be a participant in another. In a typical performer model, only certain objects can play the role of performer, and most model facilities are oriented toward those objects.

There are a number of other distinctions which roughly parallel the performer/participant distinction:

Performer Models	Participant Models
Request Message to one target object (the performer). This is the "classical" object model.	Operation applied to one or more participant objects. This is the "generalized" model.
Request Syntax Performer.Operation(Participants)	Operation(Participants)
Objects Represent System components & resources (applications, clients/servers, chunks of code, chunks of storage, files, tables, tuples, data "frames").	Enterprise entities.
Object Granularity Coarse, large.	Fine, small.
Role of Infrastructure Locate performer, deliver request, manage parameters.	Determine performer, then locate it, etc.
Method Ownership Exclusively owned by performer type.	Jointly owned by participant types [10].
State Disjointly partitioned between objects.	Could be shared among objects.
Relationships Separate constructs.	Could be multi-operand operations.
Localization Object is at one node, state is a contiguous "chunk".	Object can be dispersed, with state and services scattered (not just replicas).

This is not a strict partitioning. Various object models exhibit these characteristics in various degrees and combinations.

Note that both performers and participants can have associated behavior.

The role of OODBMS is ambivalent. Some have performer models, some have participant models.

5.3 Other Significant Differences

The following is a sampling of some other significant differences:

- Which of the following are objects?

  Literals, aggregates, tables, applications, users, icons, types, classes, operations, methods, attributes, relationship, factories,…
- Object introduction:
  - Objects are created, made instances of types, and acquire corresponding interfaces by explicit operations of the object system, or
  - Independently existing objects are accessed by the object system, which has to discover their types and interfaces.
- Identity:
  - Based on oid's? Format? Scope of uniqueness?
  - Can oid's be held outside the object system, with guaranteed meaning on future use?
  - Can objects have several (synonymous) oid's?
  - Is x=y testable?
  - How do oid's relate to naming systems?
- Data structure:
  - Expose more complex data structure, or
  - Hide structure behind behavioral abstraction.
- Behavior: is there a distinction between behavior specification and implementation? Which corresponds to methods?
- Types:
  - One level vs. subtyping.
  - Single vs. multiple (immediate).
  - Changeable?
  - Are extents maintained?
  - Various definitions of the difference between types and classes.
  - Basic data types.
- Various inheritance notions (single/multiple, class/type, blocking, instance vs. property, delegation, …)
- Treatment of attributes and relationships.
- Various approaches to containment, aggregates, composite objects.
- Query (yes/no, different query models; is it really an OO issue?).
- Single performer per object?

- All services for a given object are provided by one performer (e.g., a document is associated with one word processor for all services). All requests involving this object can be routed to the same performer.
- Services may be provided by different performers (e.g., independent performers for editing, displaying, and printing a document, possibly at different locations). Choice of performer may depend on service requested.

5.4 Why the Differences?

Object models differ for a variety of reasons. Perhaps the most important is that different goals are being pursued by various object-oriented facilities:

- Interoperability.
- Portability.
- Extensibility.
- Code reuse.
- Communication/distribution.
- Complex applications.
- Persistence of program data.
- Enterprise modeling.
- System resource management.

There are also business, political, and cultural reasons. Vendors have vested interests in their products. Users need to protect their investments in legacy applications, hardware, software, and programmer training. Researchers become committed to a certain line of development. People have learned to think in the metaphors of certain languages (C++, Smalltalk, CLOS, SQL, etc.). Nationalism plays a role. People have different views of the role of database, e.g., as a repository of structured data vs. an interface through which users manage information.

5.5 Solution Approaches

There is even diversity in the solutions to the problem, and also in the organizations trying to solve the problem [11].

One style of reconciliation is to develop a *unified* super-model, in which each model can be found as a subset or restriction of the general model. The other approach is an *interoperability* framework in which distinct models can cooperate.

Diverse agencies are addressing the problems of object technology. There are companies and consortia in the private sector, as well as government standards bodies, both in the United States and internationally. Some are concerned with the object paradigm in specific technologies, such as programming languages or databases, while others seek a uniform approach to object technology as a whole. Two of the organizations addressing these issues are:

- The Object Management Group (OMG), an industrial consortium trying to reach rapid consensus on shared conventions to promote interoperability of applications using currently available technology.

- X3H7 (Object Information Management), an ANSI technical committee intended to foster harmony in the object-oriented aspects of various information processing standards.

The key steps in such reconciliation will include:

- Recognizing the diversity of fundamental assumptions regarding the nature of objects.
- Identifying and characterizing different models.
- Developing a coherent strategy to reconcile diverse models:
  - Describe the models and their differences in common terminology.
  - Converge to a common model as much as possible.
  - Develop interoperability strategies for reconciling the unavoidable differences.

6 Conclusions

Object orientation plays a major role in the interoperation of computational systems. Its basic messaging model, and the associated architecture and infrastructure, enable applications to communicate across hardware and software boundaries and to coordinate the execution of their work. Rich semantic features such as identity, generalization and specialization, encapsulation, polymorphism, and computational completeness facilitate the sharing of common information among the applications.

At the same time, the diversity of object-oriented models is raising interoperability problems of its own.

Acknowledgments

Many thanks to my colleagues in the Pegasus project, with whom many of these concepts were developed, and especially to Stephanie Leichner for helping assemble the material for this paper.

References

1 Serge Abiteboul and Anthony Bonner, “Objects and Views”, Proc. SIGMOD May 29-31, 1991, Denver, Colorado.

2 R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan, “The Pegasus Heterogeneous Multidatabase System”, IEEE Computer, December 1991.

3 Joseph Albert, Rafi Ahmed, Mohammad Ketabchi, William Kent, Ming-Chien Shan, “Automatic Importation of Relational Schemas in Pegasus”, Proc. RIDE-IMS, April 19-20 1993, Vienna, Austria.

4 C. Batini, M. Lenzerini, and S.B. Navathe, “A Comparative Analysis of Methodologies for Database Schema Integration”, ACM Computing Surveys 18(4), Dec. 1986.

5 Umeshwar Dayal and Hai-Yann Hwang, “View Definition and Generalization for Database Integration in a Multidatabase System”, IEEE Trans. on Software Engineering, SE-10(6), Nov. 1984, pp. 628-645.

6 Elizabeth Fong, William Kent, Ken Moore and Craig Thompson (editors), *X3/SPARC/DBSSG/OODBTG Final Report,* Sept 17, 1991. Available from NIST.

7 William Kent, “The Many Forms of a Single Fact”, Proc. IEEE COMPCON, Feb. 27-Mar. 3, 1989, San Francisco.

8 William Kent, “A Rigorous Model of Object Reference, Identity, and Existence”, Journal of Object-Oriented Programming 4(3) June 1991 pp. 28-38.

9 William Kent, “The Breakdown of the Information Model in Multi-Database Systems”, SIGMOD Record 20(4) Dec 1991.

10 William Kent, “User Object Models”, OOPS Messenger 3(1), Jan 1992.

11 William Kent, “The State of Object Technology”, Canadian Information Processing, July/August 1992.

12 Amit P. Sheth and James A. Larson, “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, ACM Computing Surveys 22(3), Sept. 1990.

13 Richard Soley and William Kent, “The OMG Object Model”, in *Database Challenges in the 1990s*, Won Kim (editor), Addison-Wesley/ACM (in preparation).

14 *Object Management Architecture Guide,* Second Edition, Object Management Group, Framingham MA, 1992.

15 “The Common Object Request Broker: Architecture and Specification (Revision 1.1)”, Document Number 91.12.1, Object Management Group, Framingham MA, Dec. 1991.

16 “Information Technology - Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model”, ISO/IEC CD 10746-2.2, 26 April 1993.

Object Orientation and Interoperability

William Kent

Database Technology Department, Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, California 94303 USA
kent@hplabs.hp.com

857-8723

Keywords

interoperability, object-oriented databases, object models

Abstract

Interoperability problems arise when independently developed applications need to use each other's services and share each other's data despite a host of differences in the ways they were developed and the ways they operate. Object orientation offers some tools for solving such problems. Ironically, though, object orientation also introduces interoperability problems of its own. This examination of object orientation and interoperability outlines the problems and solutions.

This paper will be presented as a lecture at the NATO Advanced Study Institute on Object-Oriented Database Management Systems, August 6-16 1993, Kusadasi, Turkey. The lecture notes will be published by Springer-Verlag under the title *Object-Oriented Databases*. The material in this lecture was designed to complement the other lectures, and to focus particularly on the role of object-oriented databases in interoperability.