Atlas is a prototype of an object infrastructure. It is designed to support hundreds of millions of objects distributed in a heterogeneous wide-area network of millions of computers. Atlas supports much finer-granularity objects than does a traditional client/server networking model. An Atlas application will typically consist of a large collection of objects that use method invocation as their primary means of communication.

Atlas provides location-transparent method invocation and dynamic creation, deletion and migration of persistent objects. All Atlas objects are instances of a class that defines which methods they implement. Classes are arranged in a hierarchy that supports multiple generalization as well as multiple specialization.

The notion of class hierarchies from object-oriented programming languages does not scale to large distributed systems as easily as does the object paradigm itself. We shall identify the major reasons for this and explain how the problems have been overcome in Atlas.
1 Introduction

End users of computer systems can potentially access information on thousands of computers via local and wide-area networks, but today's technology does not provide sufficient integration of these computer systems to allow nonexpert users to utilize this possibility. Typically, integration is limited to the traditional client/server model within cooperate networks; beyond these, it is normally limited to electronic mail systems and file transport protocols.

We believe that object-oriented technology can overcome this lack of integration by allowing users to create and share objects and move them around the network. Typical objects in such a system will be pieces of text, spreadsheets, order forms, shared calendars and bulletin boards.

Objects will exist and be distributed because they are created, manipulated and moved around by people who inherently are distributed. An object infrastructure supporting such an environment must therefore support objects of a granularity that relates to end users of the system. In other words, the infrastructure must support object-oriented systems, but not necessarily object-oriented programming.

A typical office worker may handle hundreds of such objects during a day. An object infrastructure that will support such an environment must be able to handle thousands of objects for every user in the network. The kind of support that is needed provides not only inter-object communication but also dynamic creation, deletion and migration of objects.

Wide-area networks are heterogeneous at the programming language level as well as at the hardware level. Some kinds of information are best handled in transaction-based databases, while others are naturally handled by different kinds of programming languages. In the near future, we cannot expect a single language to support all types of applications. Therefore the object infrastructure must be able to handle objects implemented in a wide range of languages and still allow them to communicate freely.

Today's network technology is not flexible enough to allow us to build this type of large scale, integrated information system. Typical client/server or remote procedure call (RPC) models [2] for wide-area networks are not designed to support dynamic creation and deletion of millions of objects that can freely migrate around the network. Single-language systems like Emerald [3] are obviously not designed to support implementation of objects in different languages. They are usually designed to explore the inherent parallelism in local-area networks, supporting automatic migration of objects for load balancing, etc. This may potentially conflict with the end users' models of objects.

Atlas is a prototype of an object infrastructure, designed to support object-based information systems in heterogeneous wide-area networks.
The rest of this paper describes Atlas, focusing on the Atlas class hierarchy. Section 2 gives a brief overview of Atlas. Section 3 discusses why class hierarchies as we know them from object oriented programming languages do not scale to object systems for wide-area networks. As part of this discussion, we present the solutions we have adopted in Atlas to overcome these problems. Sections 4 and 5 describe the Atlas class hierarchy and its implementation. Finally, section 6 summarizes the advantages of supporting explicit class hierarchies in distributed object infrastructures like Atlas.

2 Atlas overview

In this section, we will give a very brief overview of the Atlas object model.

An Atlas object is an encapsulation of state as well as behaviour. Each object is an instance of a class that defines which methods it implements. How these methods are implemented and how the object represents its state is hidden from callers of the object.

The primary means of communication between objects is method invocation. Both synchronous and asynchronous method invocation are supported. In addition, objects can set up different types of stream-oriented communication, which are used for bulk data transmission like video or voice.

An Atlas object is identified by its object identifier (OID), allocated when the object is created. An OID is an unique, immutable identifier that will identify one and only one object throughout the lifetime of Atlas. The importance of a strong notion of identity has been argued for in [10]. Mappings from names in other naming schemes to OIDs have been implemented, but they are not part of the Atlas object model.

Atlas provides location-transparent method invocation. The only information needed to invoke methods on an object is its OID. However, Atlas does not provide failure-transparent method invocations. Programmers must deal with errors caused by partial failures in the network (for example, an object that is temporarily unavailable).

Atlas objects are multi threaded: a new thread of control is created within the object for each arriving method invocation. A thread can spawn other threads that execute within the object. Atlas provides low-level primitives such as semaphores to support concurrency control. Realisation of higher-level concurrency control policies is left to class implementers.

The heterogeneous nature of wide-area networks makes code migration impractical. Atlas only supports migration of an object's serial form, an architecture-independent description of an object's state. The serial form is specified as part of the class. Atlas does not specify how objects represent their state internally. A class implementer must therefore provide a mapping from the state representation of the object within that implementation to the serial form and vice versa. When an object is migrating, Atlas uses a two-phase
commit protocol to ensure that it is not lost and is not duplicated. Atlas provides migration transparency: only the migrating object needs to know that it is migrating; method invocations from other objects will be queued until the migration is complete and will then be forwarded to the new location.

Atlas objects are either active or passive. Active objects execute in memory, and passive objects reside in secondary storage. Whether an object is active or passive is transparent to callers of the object. Atlas ensures that an object is transformed to its active form before any method invocation is delivered to it. Class implementers must provide a mapping from an object's passive form to its active form and vice versa. The object's passive state is dependent on its implementation. Passive forms can be stored on any kind of storage such as disk, videotape or videodisc, or in a database. Atlas provides a persistent mapping from an object's OID to an identifier that can be used to find the passive form of the object.

Security is of utmost importance in wide-area networks. Many different enterprises may be using the network, and all must be able to rely on the integrity of their information. Atlas provides hooks to implement various discretionary access-control policies. The security mechanism provided is based on ECMA standard 138 [1, 6].

An Atlas security zone is a collection of objects that trust each other. An object always belongs to exactly one security zone. Security zones enable optimization of method delivery because security constraints do not need to be observed when two objects in the same security zone communicate. In particular, objects from the same security zone can execute within the same address space, making the cost of method invocations comparable to that of procedure calls.

Atlas is currently implemented on Unix and OS/2, supporting implementation of objects in Iris-OSQL [8], C++, C, and kbProlog [4].

3 Class hierarchies and wide-area networks

Classes and class hierarchies originated in object-oriented programming languages. Atlas has extended this notion to cover object-based computing systems in wide-area networks.

In this section, we identify the most important aspects of class hierarchies in object-oriented programming languages. We also identify the major problems in trying to scale this abstraction mechanism to object systems for wide-area networks.

3.1 Class hierarchies in object-oriented programming languages

Classes and class hierarchies were introduced by Simula [7]. They have proven to be a useful abstraction mechanism and have since been supported by a number of object-oriented

---

1Unix is a trademark of AT&T. OS/2 is a trademark of Microsoft Corporation.
programming languages. Among these are Smalltalk, Beta, Eiffel and C++ [5, 9, 13, 16].

There is no consensus as to exactly what a class defines. Different languages associate different characteristics with classes and thus advocate them for different reasons. We will briefly discuss some of these reasons.

### 3.1.1 Object classification

Object-oriented programming is sometimes characterized as building models of parts of a real or imaginary world [11]. In this view, objects model phenomena in the part of the world that is modeled. Objects are instances of classes, just as phenomena are examples of concepts. The most important aspect of classes is their ability to model concepts and conceptual hierarchies.

In this view, a class can be considered to be a specialization (sub type) of its ancestors in the class hierarchy. In other words, an instance of the class can be used as if it is an instance of any of the class's ancestors. This view gives very simple conformance rules for objects. It allows instances of different classes to be handled in a uniform fashion, as long as they have a common ancestor in the class hierarchy.

### 3.1.2 Encapsulation and type checking

Classes have been used for defining boundaries of encapsulation. That is, how other objects can interact with instances of the class; for example, which methods instances of the class have, and what arguments they take. How the class is implemented is not exposed beyond the interface. Such well-defined interfaces allow for compile-time type checking on method invocations.

### 3.1.3 Implementation inheritance

Traditionally, the class hierarchy has been tied to implementation inheritance. A class automatically inherits all of the implementation of its ancestors in the class hierarchy.

In some object-oriented programming languages, the entire implementation of a class, including representation of state, is exposed to its sub classes. Some have argued that this approach breaks encapsulation and potentially create unwanted dependencies between the implementation of a class and those of its sub classes [15].

### 3.2 Characteristics of wide-area networks

Wide-area networks have properties that distinguish them from the well-controlled environments usually associated with programming languages. In this section, we will discuss the most important of these properties and how they affect the possible design of a class hierarchy for wide-area networks.
3.2.1 Heterogeneity of the network and implementation languages

A wide-area network normally consists of a heterogeneous set of resources. A loadable module that will execute on one computer in the network cannot be expected to execute on another. Even at the programming language level, trying to enforce a homogeneous model for all applications is unrealistic. There are good reasons to implement some objects in databases and others in different programming languages. Objects should be able to invoke methods on other objects, in spite of different architectures and without knowing how these other objects are implemented.

One way to overcome this problem is to separate the interface description for an object from the actual implementation of that interface. This approach has a number of advantages:

- The same interface can be implemented differently on different platforms.
- Objects can migrate between different implementations of the same interface, as long as there exists an agreed way to represent the object state during migration.
- An interface defines exactly the information that is necessary to be able to invoke methods on objects supporting that interface. The caller need not know anything about the actual implementation of objects supporting the interface.

3.2.2 Partial failures and global state

Objects in wide-area networks may be temporarily unavailable due to partial failures or partitioning of the network. The possibility of partial failures and partitioning discourages the use of global state in the implementation of class hierarchies for large distributed systems.

Class variables in Smalltalk provide state that is shared among all instances of a particular class. Not only does the class variable concept require global state, it also reveals how this state is represented. The same is true for nested class hierarchies as used in Simula and Beta. So neither mechanism can be recommended for large distributed systems.

3.2.3 Distribution and code inheritance

Some object-oriented programming languages support code inheritance via the class hierarchy. In such systems, one can consider an instance \( O \) of a class \( C \) to be made up of a set of objects \( o_1, \ldots, o_n \). Each \( o_i \) represents an instance of one of \( C \)'s ancestors in the class hierarchy. When a method is invoked on \( O \) it may delegate the execution of the method to one of the objects \( o_1, \ldots, o_n \). In a distributed system, this model only makes sense if \( o_1, \ldots, o_n \) are guaranteed to reside in the same program execution on the same node as \( O \). Otherwise, the performance penalty and risk of inconsistent states due to partial
failures will be unacceptably high. In other words, implementation inheritance in a large
distributed system should be considered only as a way of organizing the implementation
of a particular class on a single node in the network.

We still believe in code reuse via inheritance or delegation. But the nature of global net­
works makes it undesirable to provide code reuse as an integral part of the abstraction
mechanism used to define the interface of objects that are potentially distributed through­
out the network.

3.2.4 Lack of global control and modifications to the class hierarchy

Most object-oriented environments rely on global control of the environment when classes
are modified. Otherwise, it would be impossible to guarantee that applications continue in
a meaningful way after the modification has been completed. The most common approach,
taken in languages like Eiffel, C++ and Beta, is to modify the class hierarchy and then
restart the entire program execution. Other, less dramatic approaches have been taken to
investigate support of long-lived systems through incremental execution. These approaches
also rely on the ability to control the entire environment in which the objects are executing.

Global control does not exist in wide-area networks, so the implementation of a class
hierarchy cannot rely on it. Instead, stricter rules must be applied as to how classes can
be modified.

A simple and sufficient rule is that existing classes cannot be modified, but new classes
can be added. This approach sounds dramatic approach, but, as will become clear in
section 5.1, it does not impose major restrictions on program development.

3.2.5 Textual name spaces

In a wide-area network, a flat textual name space will be insufficient for naming of clas­
ses, methods and parameter types. It is impossible for programmers to invent guaranteed
unique textual names for every new class, method and parameter type, especially since
separate enterprises or development groups may be developing new classes of objects tar­
geted at the same application area at the same time. The number of name collisions would
make the class hierarchy unusable.

The Atlas solution is to provide a two-level naming scheme with a context-dependent
mapping to and from more meaningful textual names. The low-level naming scheme assigns
system-generated, globally unique names to all named entities in the class hierarchy as they
are created. These names need not be readable or meaningful to humans as long as their
uniqueness is guaranteed. Conflicts in textual names can still occur, but they can be solved
in a context-sensitive way without affecting the rest of the system.
This approach is used in many cases in our everyday life – for example, in mappings between names and employee numbers.

4 The Atlas class hierarchy

Based on the arguments from section 3, the Atlas class hierarchy was designed for describing object interfaces only. It does not support implementation inheritance, as we consider the implementation of an object to be an issue orthogonal to interface definition.

In section 4.1, we describe in more detail what a class defines. The notions of class implementations and class objects are discussed briefly in section 4.2. Finally, in section 4.3, we describe the capabilities of the Atlas class hierarchy and show how name conflicts are resolved.

Section 5 then introduces the Atlas class database, which provides storage for all Atlas classes. It also defines the underlying naming scheme for all named entities in the Atlas class hierarchy.

4.1 Atlas classes

An Atlas class defines two properties for instances of the class:

- The methods that can be invoked on instances of the class and those that can be invoked on class objects for the class. A method is either an instance method or a class object method.

  Each method is defined by a unique identifier, its input and output parameter types, and whether it is to be invoked synchronously or asynchronously. Methods can be inherited from super or sub classes by specialization or generalization.

- The serial form of instances of the class, an architecture-independent description of an object’s state. This is the form an object will take when it is migrating from one implementation of its class to another.

4.2 Class implementations and class objects

Atlas does not dictate how or in which programming language a class must be implemented. Different implementations of the same class can coexist. Currently, Atlas supports implementation of classes in C, C++, kbProlog and Iris-OSQL.
An executable module that implements a class can be installed as a *class implementation* in a security zone. Within that security zone, objects of the class can from then on execute in that class implementation.²

A *class object* is instantiated when a class implementation is created. The class object belongs to and executes in that particular class implementation. Class objects cannot migrate to other class implementations. Apart from that, they can be considered to be ordinary Atlas objects. The same executable can be installed multiple times in the same or different security zones, causing a new class implementation and class object to be created every time.

An Atlas class object serves the same purpose as a Smalltalk class objects: it identifies a particular class implementation and will typically provide services like instantiation of new instances of the class. When an object migrates, it migrates from one class object to another. Migrating between class objects corresponds to migrating from one class implementation to another and enables object migration in a heterogeneous environment. It also provides a graceful path for software enhancements: when an enhanced version of a class implementation is installed, the objects that are executing in the old version can gradually migrate to the new one.

### 4.3 Creation of new classes

A new Atlas class can be created from scratch or by inheriting from existing classes. The inheritance mechanism allows for creation of new classes in two different ways: as specializations (sub classes) or as generalizations (super classes) of already existing classes.

We will use the Atlas Class Definition Language (ACDL) to illustrate all three ways of creating new classes. Other languages can be used as well, as discussed in section 5.

#### 4.3.1 Without inheritance

Figure 1 shows definition of two classes *IntegerDeque* and *Moveable*, expressed in ACDL. *IntegerDeque* defines a class of double-ended integer queues and *Moveable* a class of migrating objects.

#### 4.3.2 Specialization

Figure 2 illustrates how a new class can be created by specialization (often called sub classing in object-oriented programming languages). A new class *MoveableIntegerDeque* is created as a specialization of *Moveable* and *IntegerDeque*.

---

² Atlas allows an executable module that implements several classes to be installed as a class implementation for all these classes at the same time. This facility enables objects of these classes to execute within the same address space, making the cost of a method invocation between them comparable to that of a procedure call.
Class IntegerDeque
ClassMethod
[oid newObject] createObj(integer minsize);

Method [ bool ] empty ( ) ;
Method [ integer val ] pop_front ( ) ;
Method [ integer val ] pop_rear ( ) ;
Method [ ] push_front ( integer val ) ;
Method [ ] push_rear ( integer val ) ;

SerialForm
[integer size,front,rear; array of integer ar];
EndClass IntegerDeque ;

Class Moveable
Method [ ] move ( oid classObject ) ;

/*@ serial form not defined */
EndClass Moveable;

Figure 1: Class definitions without inheritance

MoveableIntegerDeque inherits all the methods of Moveable and IntegerDeque. In addition, it defines one new method (whereAreYou).

A specialization always inherits all the methods of its generalizations (super classes). It can therefore be guaranteed that an instance of the class always can be used as if it is an instance of any of its class’s ancestors.

The methods of a class that has been created by specialization will be the union of the methods of all its generalizations, plus any new ones that have been defined. If the same method is inherited by several paths in the class hierarchy, then instances of the resulting class will only support one such method, since specialization is done by union. In terms of [12], the class hierarchy supports singular attributes only.

The serial form is unique for every class and is not inherited.

4.3.3 Generalization

Classes can be created as generalizations of already existing classes in Atlas. In terms of object-oriented programming languages, this capability corresponds to being able to create a super class for a set of classes. This type of inheritance has traditionally not been supported by object-oriented programming languages. For a more thorough discussion of generalization, refer to [14].
Class MoveableIntegerDeque
   Specializes Moveable, IntegerDeque;

   /* return the class object that identifies the implementation set in which the object currently is executing */
   Method [ oid classObject ] whereAreYou ( ) ;

   SerialForm
      [integer size,front,rear; array of integer ar];
EndClass MoveableIntegerDeque;

Figure 2: Example of multiple specialization

A specialization is guaranteed to have a set of methods that is a super set of the union of all the methods of the classes it specializes. This must also be true when the relationship between the classes has been introduced by generalization. So a class created as a generalization will inherit the intersection of the methods of the classes it generalizes, minus the methods explicitly excluded.

In figure 3 it is illustrated how a moveable integer stack can be defined as a generalization of MoveableIntegerDeque. MoveableIntegerDeque becomes a specialization of MoveableIntegerStack. Instances of MoveableIntegerDeque can be used as if they were instances of MoveableIntegerStack. This relationship is introduced without changing any existing classes (in particular without changing MoveableIntegerDeque).

Class MoveableIntegerStack
   Generalizes MoveableIntegerDeque;

   Excludes pop.rear, push.rear;

   Renames pop.front pop;
   Renames push.front push;

   SerialForm
      [integer size,top; array of integer ar];
EndClass MoveableIntegerStack;

Figure 3: Example of generalization

Generalizations can also be useful for creating abstractions of a set of classes that all support the same methods. Suppose that three classes, Email, TechReport and Book,
have all inherited the two methods `numOfPages` and `print` by different paths in the class hierarchy. We can create a new class, `PrintableDoc`, having only these two methods, as a generalization of the three types of document classes. Instances of `Email`, `TechReport` and `Book` can then all be handled as if they were of class `PrintableDoc` by objects for which this is a sufficient specification.

Atlas allows introduction of new textual names for existing methods by renaming, as shown in figure 3 above. The name `push` is introduced as the new name for the method that was defined as `push_front`. The name `push` can only be used within the context of class `MoveableIntegerStack` or new classes that inherit from it.

### 4.3.4 Name conflicts

Two types of name conflicts can occur through inheritance by multiple specialization or multiple generalization:

- The same method is inherited from two different classes but is referred to by different textual names in these two classes. This can occur as a result of renaming.

- Different methods that are denoted by the same textual name in different classes are inherited.

The rename primitive must be used in both cases to ensure an unambiguous mapping between textual names and methods.

Name conflicts only occur on the textual names. There are never conflicts in the underlying naming scheme, where globally unique names are assigned to all named entities in the class hierarchy. This naming scheme lets different classes and other named entities be referred to by the same textual names in different contexts, or the same entity to be referred to by different textual names in different contexts. The low-level naming scheme will be discussed in more detail in the next section, where we present the Atlas class database.

### 5 The Atlas class database

All Atlas classes are stored in the Atlas class database. It defines the format in which classes are stored, and thereby defines the limitations of possible class definition languages for Atlas. ACDL directly expresses the capability of the class database format, which we denote the Atlas Class Representation (ACR).
5.1 Entries in the class database

ACR defines two types of entries in the database. A method entry is a method identifier (MID) with an associated method description. A class entry is a class identifier (CID) and a class description. Both MIDs and CIDs are globally unique identifiers that are allocated when the entry is created.

A method description describes the parameter lists (input, output) of the method, whether the method is to be invoked synchronously or asynchronously, and in which class it originally was defined. No textual name is associated with a method description. A method entry for createObj as it was defined in figure 1 is shown in figure 4.

mid: 473efc17d55e.02.0f.08.3d.87.00.00.00.01,
defclass: 473efc17d55e.02.0f.08.3d.87.00.00.00,
synchronous,
inparams: ( integer minsize ),
outparams: ( oid newObject )

Figure 4: Method entry for createObj

A class description consists of a textual name for the class, all the classes that it specializes or generalizes, and two method lists, one for ordinary methods and one for methods that can be invoked on class objects. A method list is a set of MIDs, each with an associated textual name that is used to identify the method in the context of this class. Finally, the class description contains a description of the serial form for instances of the class. A class entry for class MoveableIntegerStack (figure 3) is illustrated in figure 5.

cid: 473efc189ada.02.0f.08.3d.87.00.00.00,
classname: "MoveableIntegerStack"
generalizes: 473efc44ec38.02.0f.08.3d.87.00.00.00,
classmethods: ( ("createObj", 473efc17d55e.02.0f.08.3d.87.00.00.00.01))
objectmethods: ( ("empty", 473efc17d55e.02.0f.08.3d.87.00.00.00.02),
("pop", 473efc17d55e.02.0f.08.3d.87.00.00.00.03),
("push", 473efc17d55e.02.0f.08.3d.87.00.00.00.05),
("move", 473efc13f5da.02.0f.08.3d.87.00.00.00.01),
("whereAreYou", 473efc44ec38.02.0f.08.3d.87.00.00.00.01)),
serialform: ( integer size, top ; array of integer ar )

Figure 5: Class entry for MoveableIntegerStack

\[A\] A third type of entry is used for defining new parameter types. It will not be discussed in this paper.
New classes are created in Atlas by installing the corresponding new entries in the class database. Installation of a class will always cause creation of a new class entry. Depending on the way the class was defined, a set of new method entries will be created as well. A method entry for each method of the class will be created if the class has been created without inheritance. Installation of a class that is a specialization will only cause creation of a method entry for each new method defined by the class. If the class is a generalization, its installation will not cause creation of any new method entries, since they all must exist in the database already.

The globally unique CIDs can be used to resolve conflicts involving different classes with the same class names. This use is especially important during development, where changed versions of a class definition may be installed many times. A new class with the same class name is created every time a new version of the class is installed. So what looks to the programmer like modification of a given class is successive creation of classes that are very similar.

Any language that can be mapped to ACR can potentially be used as a class definition language for Atlas objects. Atlas is using NCS [2] as its underlying RPC mechanism. This enable NCS objects to act as Atlas objects. A mapping from ACR to the NCS Network Interface Definition Language (NIDL) has been implemented to support implementation of such NCS objects.

The main purpose of the Atlas class database is to enable objects that are implemented in different languages to communicate, using well-defined interfaces. To support such communication, the class database can be used to generate abstractions in different programming languages. These abstractions can either be an implementer's template or a set of caller stubs for a particular class. The templates and stubs hides details of the Atlas run-time system and maintain the level of abstraction provided by the programming language. An example of details that will be hidden is marshaling of parameters. The implementer's template constitutes a partial implementation of the class, in the language for which the template was generated. The caller stubs can be used to invoke methods on a given class of objects, from the level of abstraction provided by the language in which the template was generated.

The class database can also be used as a run-time resource. This ability is especially relevant for debuggers, agents and other objects that need to obtain information about other classes at run-time.

At the moment, templates and stubs can be generated for C, C++ and Iris-OSQL. Atlas objects can also be implemented in KBMS1 [4]. It is using the class database as a runtime resource to map from textual names to MIDs.
5.2 Implementation of the class database

The Atlas class database is not a centrally kept and administrated database. The main reasons being efficiency, existence of partial failures in the network and the privacy of independent enterprises in the network. Instead, it is realised as a collection of independent databases that are administrated locally. We refer to these independent databases as Local Atlas Class databases (LAC-dbs). Information about a particular class may be replicated in many LAC-dbs. There is no global administration of the Atlas class database. It is simply the collection of all the autonomously maintained LAC-dbs.

When a new class is created in a LAC-db, certain entries must already exist in the LAC-db. Otherwise, creating the new entries and generating templates and stubs for the new class would be impossible. The entries that already must exist are:

- A class entry for all classes that the new class inherits from.
- A method entry for all inherited methods.

Entries in the Atlas class database are immutable; an entry cannot be changed once it has been created. This fact plus the global uniqueness of MIDs and CIDs imply that LAC-dbs can be merged freely, since they all will be consistent. The easiest way to extend a LAC-db that is missing some entries is simply to merge it with another LAC-db that contains the missing entries.

A new LAC-db that defines only a set of chosen classes can be spawned from an existing LAC-db. The new LAC-db will contain a class entry for each of the classes in the set and a method entry for each method of those classes (both those that are defined and those that are inherited by the classes). Spawning new LAC-dbs is the way selected classes will be distributed among enterprises.

6 Advantages of the class hierarchy

The Atlas class hierarchy has been fully implemented as part of the Atlas prototype. We believe it provides a number of advantages for development and interoperability of object-based applications in wide-area networks. We will conclude by summarizing the most important of these advantages.

A class defines the interfaces of objects independent of the platforms on which these objects execute and of the language in which they are implemented. Objects of a given class can execute in databases or can be implemented in some programming language; their implementation is transparent to other objects that invoke methods on them. The only information that is necessary to invoke a method on an object is the OID and the signature of the method.
The serial form defined by the class defines how instances of the class are represented during migration. This is sufficient to allow an object to migrate between different implementations of its class.

The well-defined interfaces defined by the classes in object-oriented programming languages supports development of software in large projects. Atlas classes are especially suited for this purpose since they only define interfaces and do not introduce unwanted dependencies through code inheritance.

The class hierarchy allows for compile-time or run-time type checking on method invocations.

Software enhancements can be provided by installing new implementations of relevant classes and letting objects migrate to them. Migration does not change the OID of an object nor which class it is an instance of. In other words, the migration is transparent to other objects in the system.

A software vendor implementing a class of objects, such as a desktop, can define and publish a class DesktopObject that all objects on the desktop must be a specialization of. This requirement will allow other software vendors to define and implement classes of objects that can exist on the desktop.

A software vendor will publish classes that define the methods that he intends other software vendors objects to invoke on objects he is delivering. He does not need publish the actual class of the objects. He can publish generalizations that only has the methods that wants other software vendors to know about.

**Acknowledgment**

Peter Williams greatly influenced our work. His vision of distributed information systems defined guidelines for most major design decisions taken in Atlas.
7 References


