Using Interface Inheritance to Address Problems in System Software Evolution

Graham Hamilton
Sanjay Radia

SMLI TR-93-21 November 1993

Abstract:

Two specific problems faced in large distributed systems are: (1) evolving and managing different versions of an interface while minimizing the impact on existing clients; and (2) supporting the addition of auxiliary interfaces that are orthogonal to the main interface of an abstraction.

In the context of the Spring distributed system, we addressed both problems using an object-oriented interface definition language. Different versions of an interface are represented as different types, with an inheritance relationship that minimizes the impact on existing clients, and allows easy management of versions.

We distinguish between fundamental and auxiliary properties, each of which is defined as a separate type. Rather than use simple root inheritance, we use a combination of root and leaf inheritance. This provides flexibility in supporting auxiliary properties, and allows us to add new auxiliary properties as the system evolves, without forcing the system to be recompiled.

The solutions have been tested and refined through their use in the Spring system.

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
graham.hamilton@eng.sun.com
sanjay.radia@eng.sun.com

Using Interface Inheritance to Address Problems in System Software Evolution

Graham Hamilton and Sanjay Radia

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

This paper addresses two problems in interface evolution which occur in large distributed systems, and describes how we have attacked these problems by using a strongly typed interface definition language (IDL) with multiple inheritance.

The first problem is managing the orderly evolution of new versions of individual interfaces. Interfaces are rarely perfect when first defined; they are changed and extended as the system evolves. This is fairly easy to solve in a closed system where all components that use or support a revised interface can be updated simultaneously. However, making a simultaneous update is well nigh impossible in any system that is intended to be open, to encompass software from a variety of sources, and to be used by thousands of different customers. Since we cannot update the entire system at once, we have to allow multiple versions of an interface to coexist simultaneously.

The second problem is managing the orderly introduction of new auxiliary properties that complement existing abstractions. The classical approach is to inherit such auxiliary properties at the root of the type lattice. This approach makes the system hard to extend and evolve, since whenever a new auxiliary property is added, the entire system needs to be recompiled.

We use the type system to express and manage both versioning and auxiliary properties. This paper describes our motivation, the solutions we have adopted, and their consequences. We have implemented and used the solutions described in this paper in the Spring distributed system. While our work occurs in the context of a distributed system, we believe our techniques will be useful in managing the evolution of any strongly typed interfaces.

Section 2 briefly describes the Spring system. Section 3 defines the versioning problem, and describes our solution. Section 4 discusses how we deal with the problem of auxiliary properties; we examine three solutions. Section 5 examines and comments on related issues. Section 6 provides some conclusions.

2 Overview of the Spring System

Spring is an experimental distributed environment. Its current incarnation includes a distributed operating system and a support framework for distributed applications. Spring is particularly focused on issues common to large commercial distributed systems—in particular, the need to let the system evolve over time, and the need to let third party developers add their own unique (and unexpected) extensions to the system.

This focus on evolution and extensibility led to a strong emphasis on interfaces. In particular, we had the following goals relating to our use of interfaces:

- There should be a very clear separation between interfaces and implementations. The system should be perceived primarily as a set of interfaces, rather than simply as a set of implementations. It should always be possible for multiple different implementations of a given interface to coexist within a single system.
- There should be no special status for interfaces or implementations that are provided as part of the standard system. A third party application writer should be able to add new interfaces or implementations that are indistinguishable from the functionality that was provided in the standard system.
- It should be possible to evolve the system in an orderly way over time so that new functionality can be added to old interfaces, without disturbing existing applications.
- It should be possible to add new pervasive properties to the system. For example, the current system has no support for atomic transactions or accounting. It should be possible to slowly add new properties like these to the system.

The tactics which we have adopted include:

- An interface definition language supporting multiple inheritance for defining all the major interfaces.
- A strongly typed system, which uses static type checking where possible and dynamic type checking in limited circumstances.
- The use of the object metaphor for representing system resources.

2.1 The interface definition language, IDL

The unifying principle of Spring is that all the key interfaces are defined in an interface definition language. Currently, we use the Object Management Group's IDL interface definition language [OMG 1991].

IDL is object-oriented, and includes support for multiple inheritance. It is purely concerned with interface properties (the methods and their arguments), and does not provide any implementation information. The Spring implementation of IDL includes extensions, so that interfaces can include semantic information. (A related project is using this semantic information to help with automated test generation.)

From IDL type definitions we can generate surrogate objects. A surrogate object provides a language-specific mapping to the Spring interface it represents. For example, in our main implementation language, C++ [Ellis & Stroustrup 1990], Spring objects are represented by C++ objects. When a surrogate object is invoked, it will either perform a local call within the current address space to the local implementation of the object, or forward the call to another address space that implements the object, possibly on a different machine. Calls across address spaces are done using Remote Procedure Calls [Birrell & Nelson 1984].

2.2 Terminology

We use the term *type* to refer to the programming language types that one may define in a typed, object-oriented language such as Eiffel [Meyer 1991] or C++, or to types defined in an interface

definition language such as IDL. We shall use the term *abstraction* to describe the conceptual entities that programmers think about. In general, each abstraction is expressed as a specific type. For example, a programmer's abstraction "file" may be expressed as an IDL or a C++ type of the same name. However, as we discuss below, an abstraction may also be represented by a series of types.

When a type foobar inherits from type foo, we say that foobar is the *derived* type of foo, and that foo is the *base* type of foobar.

2.3 Static and dynamic typing

There are two main styles of type checking in object-oriented systems. In some systems, such as Smalltalk [Goldberg & Robson 1983] or CLOS [Bobrow et al. 1988], type checking is performed dynamically at run time. A program attempts to operate on an object as though the object belongs to a particular type, and if the object is indeed of that type, then the operation succeeds; otherwise, an exception handling method is called. In other systems, such as Eiffel or Trellis/Owl [Schaffert et al. 1986], type checking is performed statically at compile time. A program knows statically that a given object has a particular type, and that it can therefore safely perform particular operations on the object.

The Spring project's focus on formally specified interfaces led us to adopt static typing in our interface definition language and in the C++ stubs generated from the interface language. Given an object of a particular type, there is a well defined set of methods that one can invoke on that object. Each method is defined to take arguments of particular types.

However, we also support a limited amount of dynamic typing. Each object in the system is created as an instance of some particular type, which we shall call its *true type*. At various times, as the object moves around the system, it may be perceived as different base types of this true type. Now, given an object that is statically perceived to be of some type A, Spring programmers can attempt to *narrow* it to some derived type B which inherits from A (see Figure 1). This operation will be dynamically type checked, and it will only succeed if the object's true type either is B or inherits from B. Otherwise, the type-checking mechanism will raise an exception. We also provide a more general analogue of narrow, called *traverse*. Given an object perceived to be of type A, a *traverse* to type C will succeed if the object's true type is C, or if its true type inherits from type C. (Traverse removes narrow's rather gratuitous requirement that C must be a derived type of the currently known type A.)

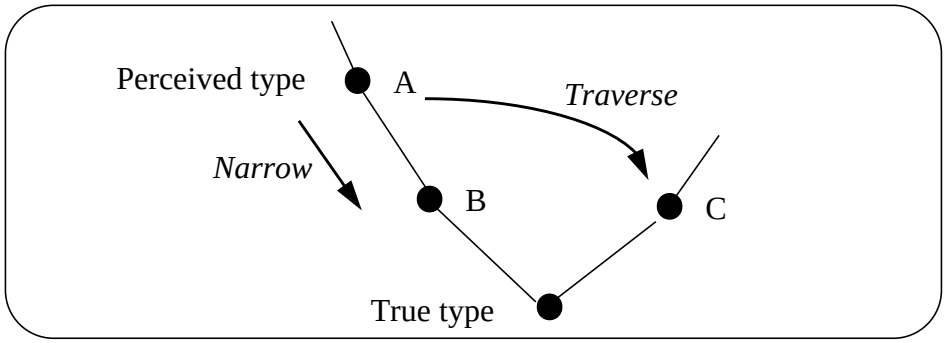

FIGURE 1. Narrow and traverse in the inheritance lattice

In general, we attempt to use static typing wherever possible, and only fall back on limited dynamic typing in situations where the type checks cannot reasonably be performed at compile time. However, Spring always remains a strongly typed system, in that all operations must be type checked, either statically or dynamically [Cardelli & Wegner 1985].

3 Interface Version Management

When an interface is first defined and introduced into a system, it is rarely perfect. As experience is gained in using the interface, and as new requirements are identified, new features may be added, and old features may become obsolete. Occasionally, it may be necessary to perform a fairly major redesign of the interface to reflect new circumstances. Thus, interfaces are rarely static. Instead, they evolve over time.

An interface is a contract between clients and implementations. When we change that contract, we have to consider the effects on both the clients and the implementations.

In general, when an interface is changed, we wish to minimize the impact on existing clients. Wherever possible, old clients should be able to continue to operate against the new interface, without any source program changes or recompilation. When this is not possible, we would like at the very least to get a clean failure as early as possible. Similarly, old implementations should be permitted to continue to support the old interface for their clients.

In a distributed system, where components are supplied by different groups, it is extremely difficult to move the entire system forward to new interfaces at the same time. A period of several years may elapse between the definition of a new interface and its universal adoption. This results in the *versioning problem*, which is that the system must permit multiple, differing versions of the same interface to coexist simultaneously.

While a fair amount of attention has been given to the problems of managing different versions of object instances or object implementations [Putz 83], particularly in the context of object-oriented databases [Skarra & Zdonik 86], [Bjornerstedt & Hulten 1989], comparatively little attention has been paid to the problems of evolving interface types in a general purpose object-oriented system.

3.1 Versioning versus adding new abstractions

This section distinguishes between versioning and adding new abstractions. When a distinctive new piece of functionality is being added to the system, it is often appropriate to express it as a new abstraction, rather than as a change to an existing abstraction. For example, when a set of debug methods is being added to the existing “thread” type, it might be appropriate to express this as a new “debuggable_thread” abstraction, which inherits from “thread.” Clearly, there is no versioning problem here—we have introduced a distinct new abstraction, which merely happens to inherit from an existing abstraction.

On the other hand, new functionality that logically belongs to an existing abstraction is often added to a system. For example, we may wish to add the ability to find the address space object associated with a thread. Now, this could have been expressed by adding a new abstraction “thread_with_get_address_space,” but this seemed rather cavalier. It is better to regard this change as a modification of an existing abstraction, rather than as the addition of a new abstraction to the system. This introduces the versioning problem. The system must allow the two versions of

the thread abstraction to coexist, because some clients may require the later version, while other clients can tolerate the earlier version.

3.2 An example of versioning

To illustrate the issues involved in versioning an interface, let us consider a simple example of extending an existing system abstraction. The exact details of the example (shown in Figure 2) should not be taken too literally—the example is meant to illuminate typical problems, rather than to be an exact description of any real configuration.

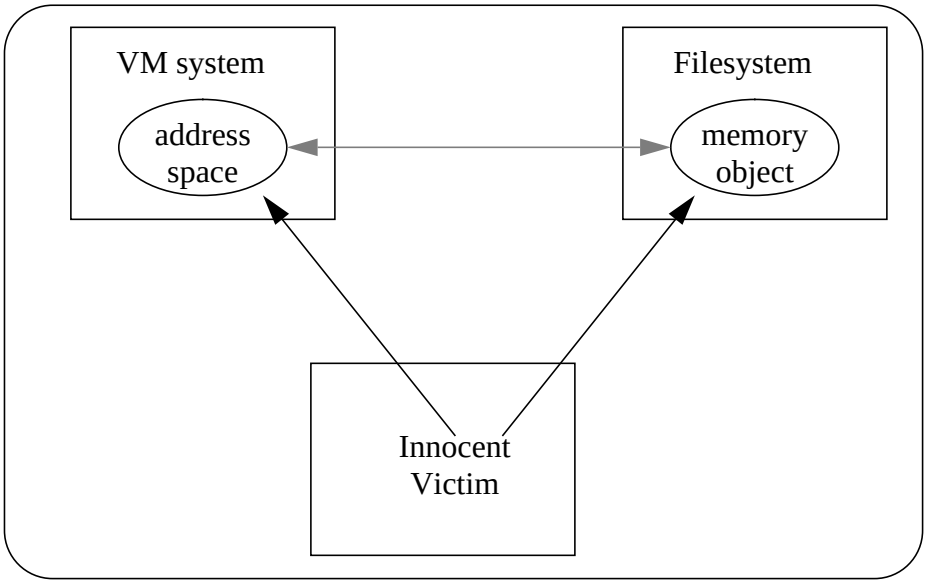

FIGURE 2. The Innocent Victim scenario

There are three players:

1. A Virtual Memory system (VMS) that implements the “address_space” type, which has a map method that takes a “memory_object” type as an argument.
2. A filesystem that implements the “memory_object” type.
3. An Innocent Victim (a client) that happens to use (1) and (2).

In order to map a file into its address space, the Innocent Victim obtains a memory_object from the filesystem, and passes it as an argument to the method of the address_space object, implemented by the VM system.

Initially, the filesystem supports version 1.0 of memory_objects, and the VM system supports version 1.0 of address_spaces. Version 1.0 address_spaces are defined to accept version 1.0 memory_objects.

After some years, the VM interface committee decides to add some new features to the “memory_object” abstraction, which allow the VM system to do a better job of paging. The VM interface committee therefore defines version 1.1 of memory_object.

The following questions arise:

- When and how does the filesystem implementation change?
- When and how does the VM implementation change?
- When and how does the address_space interface change?
- When and how does the Innocent Victim change?

The file system perspective

At some point, the filesystem implementation should be changed to support the version 1.1 memory_object interface. This change will presumably occur in the next release of the filesystem software, whenever that happens to be. However, even then, various users may continue to run old implementations of the filesystem for many years to come.

When the filesystem interface does change, old clients of the filesystem should be able to work against the new improved filesystem without any changes.

The VM system perspective

At some point, the VM system implementation should be changed to exploit the version 1.1 memory_object interface.

There are many different pieces of software that support the memory_object interface. The VM implementation cannot assume that all this software will be converted to use the 1.1 memory_object interface overnight. Therefore, the VM implementation must be able to cope with both 1.0 and 1.1 memory_objects for some period of time.

The Innocent Victim's perspective

The Innocent Victim neither knows nor cares about the details of the memory_objects it is using. It receives memory_objects from the filesystem, and it passes them to the VM system through the address space interface.

We would prefer not to have to change or recompile the Innocent Victim because it does not depend on these changes.

3.3 Versions as types

Since we intend to permit multiple versions of an abstraction to be simultaneously active in the system, programs should be able to name these versions, and determine whether an object implements a particular version of a given abstraction.

In Spring, we have chosen to use distinct types to represent distinct versions of an abstraction. Thus, a given abstraction is represented as a set of types, with each member defining a different version of the abstraction.

Interface changes are categorized into minor and major revisions. If an interface is changed in a way that will prevent old clients from using it, then this is a *major revision*. All other changes are *minor revisions*. For example, adding a required argument to a method would constitute a major revision, but adding a new method would only constitute a minor revision. We expect that most interface designers will attempt to express most changes as minor rather than major revisions, to avoid discomfiting their clients.

Interface versions are identified by a pair of version numbers, one major and one minor. Accordingly, type names take the following form:

<abstraction_name>_<major_version_number>_<minor_version_number>

Our versioning scheme is as follows:

- For each abstraction foo, first create a type foo_0_0. This type does not define any methods—it exists only to be a highly abstract base type for specific major versions.
- Whenever a new major version is defined, create a new type (say foo_1_0) which inherits directly from the base type foo_0_0.
- Whenever a new minor version is defined, create a new type (say foo_1_1) which inherits from the previous revision.

Thus, after a sequence of major and minor revisions, we might end up with a tree as shown in Figure 3.

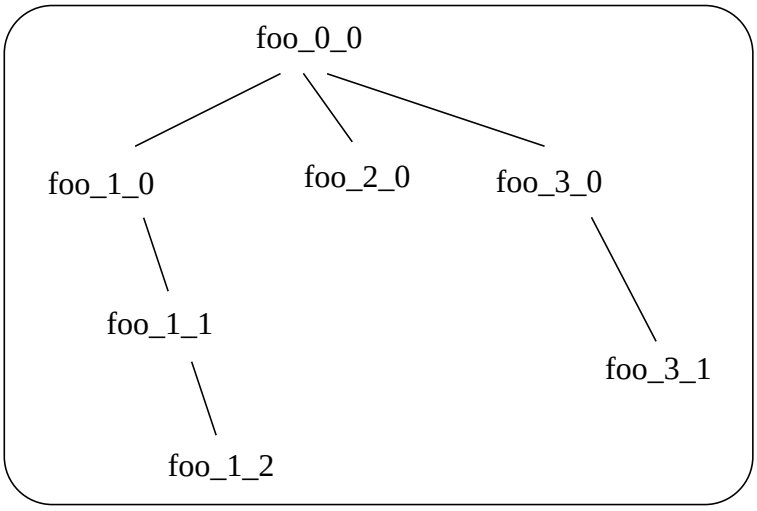

FIGURE 3. Major and minor revisions

This inheritance lattice provides the properties we want. If an interface is defined as requiring an object of type foo_1_1, then we can pass it an object of the minor revision foo_1_2, but we cannot pass it an object of the earlier minor revision foo_1_0, or of the distinct major revision foo_2_0.

More interestingly, we can define interfaces that take objects of type foo_0_0. Any of the type foo_1_1, foo_2_0, etc., can be passed where a foo_0_0 was expected. This is suitable for the "Innocent Victim" scenario described above, where the victim is merely passing the object through, and does not care about the version. Thus, the address_space interface will specify that it takes an argument of type memory_object_0_0.

Spring supports a narrow operation, where a program refines its perception of an object's type to some derived type. Thus, a server implementing an interface with a method that takes an argument of type foo_1_0 can attempt a dynamic run-time narrow of an argument object to foo_1_1, if it would like to exploit some additional functionality provided by that minor revision. Similarly, a server can dynamically narrow an argument object of type foo_0_0 to one of the major versions.

3.4 Propagating version changes to derived types

Types that are versioned may be inherited by other types. How are versioning changes propagated to other derived types? Because a new revision of an existing type is implemented as a new type, it does not directly affect any derived types. These derived types will themselves have to be revised to inherit from the new revision of their base type. We consider this property to be desirable—we want the set of methods represented by the type wombat_1_2 to remain constant, and not be arbitrarily changed by revisions to its base types.

Now if an important base type is revised, then we would expect that ultimately all its derived types will also be revised. If several important base types are revised, then there seems to be a danger of a combinatorial explosion of derived type revisions to reflect each of the successive changes.

Fortunately, human involvement will tend to dampen down the number of revisions that are introduced. For instance, if several base types are due to be revised, then we can expect the human maintainers of each derived type to wait for all the revisions of their direct base types to occur before performing a revision of their own type. Thus, revisions of several types in the top part of the inheritance lattice may result only in a single revision of a derived type.

Unfortunately, this does tend to increase the delay in propagating a new type-revision. If we issue a new revision of an important base type such as *io* (our input-output interface), then it may be some time before all its derived types have been revised to affect this change. Additional periods of time may be required for all the implementations of these derived types to be revised, and for all the old implementations at customer sites to be replaced.

4 Auxiliary Properties

In Spring, there are some object properties that are possessed by many objects in the system, but are really orthogonal to the main interface of each object.

For example, we support the interface “tabular” for providing generic instrumentation of Spring objects. There are standard tools that use the tabular interface to extract and display information about objects. As a result, many object implementations have chosen to support the tabular interface.

For example, the *tty* objects implemented by SpringOS generally support three auxiliary interfaces: the *tabular* interface for displaying statistics, the *acl* interface for managing access control lists, and the *authenticated* interface for implementing a standard authentication protocol. However, each of these three interfaces is quite separate from the main business of supporting the *tty* interface. It would be perfectly reasonable to provide a different *tty* implementation that supported none of these properties. Indeed, these three properties were added after the *tty* interface was originally defined.

We experimented with three different approaches for expressing auxiliary properties in the Spring type system.

4.1 Approach 1: Simple inheritance of auxiliary properties

Originally, we simply defined a type for each auxiliary property, and inherited these types when defining a new type if it seemed appropriate. This is a fairly standard use of mixin inheritance to

define auxiliary properties [Bracha & Cook 1990]. Thus, for example, the standard type *io* inherited from the auxiliary types *acl* and *authenticated*.

On close examination, this approach exhibited two fairly serious flaws.

First, many of the auxiliary properties appeared to be genuinely orthogonal to the main type of the object. Our habit of inheriting standard properties when defining standard types was actually expressing much stronger requirements than were appropriate. It might seem desirable that most implementations of the *file* type should implement the *tabular* interface, but if we express this desire by making the *file* type inherit from *tabular*, then we imply that all implementations of *file* must implement the *tabular* interface. In practice, this turned out to be an undesirable constraint, and many implementations whose type inherited a given auxiliary property in fact failed to implement the property, because it was inappropriate or excessively burdensome. This is clearly an undesirable situation, since it means that the type system is making promises that the implementations are not prepared to meet.

Second, this style leads to a serious discrimination between auxiliary properties that are designed into the base system and properties that are added later by third parties. For example, consider what happens if a third party software developer comes up with a different instrumentation interface, which they want to use for all their implementations. They will be unable to change the definition of widely used types, so they will need to find some other mechanism than the one used for *tabular* to express their new auxiliary property. Since one of our goals is to permit third party interfaces to be on a par with standard interfaces, this was an unacceptable deficiency.

4.2 Approach 2: Auxiliary properties outside the type system

Our next approach was to avoid expressing any linkage between types and auxiliary properties within our type system, but instead provide a way outside the type system for converting between different views of an object.

Thus, the types *file*, *acl*, and *authenticated* were defined as entirely distinct types. An implementation object might choose to support these three interfaces simultaneously, but would support them as three essentially unrelated interfaces.

We added a special operation to our basic object mechanisms, *rotate*, for moving between the distinct interfaces that an object provided. Thus, given an object of type *file*, an application might attempt to perform a *rotate* on the file to obtain an object of type *acl*.

The main problem with this approach is that it steps entirely out of the type system. It forces us to treat an object as implementing N distinct interfaces, rather than as supporting a single interface built from N base types. Special mechanisms were required, both on the client-side to support rotation, and on the server-side to support the gluing together of one piece of implementation state to several distinct interfaces. So, we decided to explore other ways of expressing auxiliary properties in the type system.

4.3 Approach 3: Auxiliary properties as implementation mixins

The main failing of our first attempt at expressing auxiliary properties in the type system was that we had expressed the inheritance high in the type tree, where it affects not just a single implementation, but all implementations of a given type and, worse yet, all implementations of derived types of that type.

An alternative rule involves distinguishing between two kinds of types. Types of the first kind are *pure* types, which express interfaces provided to client programmers, and which are intended as bases for inheritance. Types of the second kind are *concrete* types, which reflect the particular set of properties provided by a particular implementation object.

So, for any given property, such as *file*, *acl*, or *authenticated*, one would define a pure type that expressed this property. This may involve inheriting from other pure types that define properties that are fundamental to the new type. So *file*, for example, might inherit from *io*.

When you actually come to implement an object, you define a concrete implementation type that inherits from the list of pure types whose properties you wish to support. Thus, you might define a concrete type *secure_file* which inherits from *file*, *acl*, and *authenticated*. Figure 4 shows two concrete types used by two different implementations of the *file* abstraction, *foosoft_file* and *acme_file*, which inherit different sets of auxiliary properties.

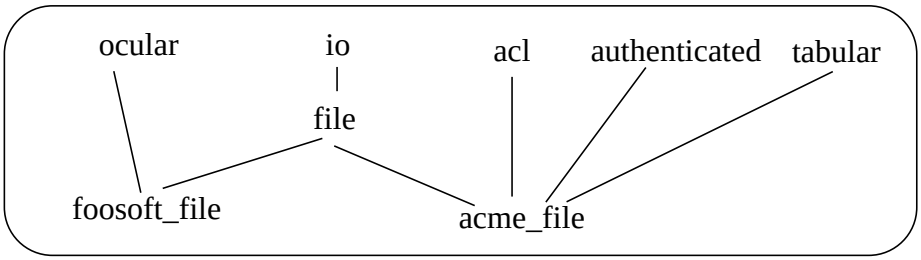

FIGURE 4. Two implementations of file with different auxiliary properties

Clients normally deal with objects entirely in terms of pure types. All method parameters for pure types are required to be themselves pure types, rather than concrete types.

The Spring system provides the ability to move between different points in the type hierarchy when viewing an object, using the traverse operation. Thus, if a program perceives an object to be of type *file*, it can attempt to perceive it to be of type *acl*. If the object's true type (i.e., its concrete type) inherits from *acl*, this operation will succeed, and the program will now be able to operate on the object's access control list.

The key difference between this approach and our earlier approach is that the standard types in our type hierarchy are not polluted by undesirable inheritance. A complete set of types is available, which define only the fundamental properties of their abstraction and do not drag in gratuitous auxiliary properties.

Since client applications deal only in terms of pure types, implementors can add new auxiliary properties as pure types, and define new concrete implementation types that use these properties. Then, these new auxiliary properties can be accessed in exactly the same way as other auxiliary properties supplied with the base system.

One drawback of this scheme is that it leads to a plethora of concrete types. Basically, each implementor of a pure type ends up by defining their own concrete type, which pulls together the particular set of pure properties that they support. However, this is not a particularly serious problem. Types are cheap.

4.4 Comparing the three approaches

Our three approaches lead to type systems with different properties. The use of simple inheritance had the pleasant property that it relied on a standard inheritance model, and was amenable to static type checking. Since *io* inherited from *acl*, the static type system could validate that all *io* objects were also of type *acl*, and an *io* object could be substituted where an *acl* object was expected without any special transformation. Unfortunately, this simple inheritance model imposed undesirable requirements on new subtypes of existing types, and was not amenable to the addition of new auxiliary properties by third parties.

The *rotation* approach satisfied our desire that the standard type hierarchy not be polluted with what should really be optional properties. However, it had the fairly serious defect of being a special case mechanism, which lived entirely outside the type system.

Our final approach of dividing the type tree into *pure* and *concrete* types appears to have satisfied our goals. The pure type tree that is seen by application writers is not polluted by a particular set of auxiliary types. We still use the standard type system and standard type inheritance to express the properties provided by particular implementation objects. The drawback of this scheme is that it relies on dynamic type-checking when moving between standard types, such as *io*, and optional properties, such as *acl*. Unfortunately, this seems to be a natural consequence of our goal of avoiding tying auxiliary properties to standard types. Fortunately, we are able to type-check the narrow operation to maintain strong typing.

[Ossher & Harrison 1992] advocate an interesting technique for merging inheritance hierarchies as a way of adding extensions to an existing tree. However, this technique is primarily focused on merging implementation hierarchies, and has the effect of mutating existing types to reflect merged properties, which is incompatible with our goal of allowing different parts of the distributed system to implement the same types, but support different auxiliary properties.

5 Reflections and future directions

5.1 Type immutability

Our inheritance conventions were designed on the assumption that types should be immutable. Once a type has been defined and exposed as an interface, then its definition should never change. Instead, a new type should be defined that reflects the appropriate changes.

This has the desirable property that throughout the distributed system the definition of a given type is always the same, even between applications compiled at different times against different versions of the system. This means that if an object claims to implement some type T, then the meaning of T will be the same both to the object and to its clients.

5.2 Versioning and static versus dynamic typing

As mentioned earlier, Spring uses a limited form of dynamic typing: the narrow and traverse operations. Our use of versioning tends to encourage some use of dynamic typing.

For example, consider when there is a type T_1_0, one of whose methods is specified to take an argument object of type A_2_0. At some point, we add a new revision A_2_1, which inherits from A_2_0. After a while, some of the implementations of T_1_0 would like to be able to exploit the extra properties of A_2_1, even though the T_1_0 interface says that it takes A_2_0 objects as

arguments. The owners of the T abstraction may choose to add new methods that take A_2_1 arguments, but they cannot remove or upgrade the old methods which currently take A_2_0 arguments, as this would constitute an incompatible revision, a step that they are (commendably) loathe to take.

Thus, in order to exploit the new A_2_1 features, some of the implementations of T_1_0 may use the narrow operation to discover if the argument objects are really of type A_2_1.

As long as the implementation continues to work with objects of type A_2_0, and the narrow to A_2_1 is merely an optimization, it is relatively harmless. Unfortunately, there is a real danger that the implementation may become dependent, in some circumstances, on the new features of A_2_1, and be unable to operate on objects of type A_2_0. This can happen if the A_2_1 type becomes widespread, whereby most or all instances of A_2_0 objects are also of type A_2_1. The implementation code of T_1_0 that deals with the objects that are of type A_2_0 and not A_2_1 may become unused and difficult to support as time goes on; indeed, it may be tempting simply to stop allowing argument objects that are of type A_2_0 and not A_2_1.

Thus, although the T_1_0 interface says it accepts objects of type A_2_0, in practice some implementations may reject arguments unless they can be narrowed to type A_2_1.

Unfortunately, there is no easy solution to this problem. This can be avoided in a purely static type system which insists that those implementations of T which wish to rely on A_2_1 must define a new major revision of T, T_2_0, whose arguments take objects only of type A_2_1.

However, this is in practice a very undesirable solution. It requires that all potential clients of the new T implementations be modified and recompiled in order to work against the new type signature. This may lead to a ripple effect, as these clients must also change their own interfaces to specify that they require A_2_1s rather than A_2_0s. In reality, many (or all) of the clients may be using objects whose true type is A_2_1, even though they only perceive it as A_2_0. In this case, it seems undesirable to break these applications by insisting that this be determined statically rather than dynamically.

So, our solution is to encourage new interfaces to define the most precise version for their arguments and to encourage implementations to deal with old versions even if they would prefer newer revisions. We also recognize that some objects may sometimes, for practical reasons given above, require that they be passed more recent revisions than are actually specified in their type signature. A revision in interface may take place at a later, more convenient time.

We console ourselves with the thought that it is actually quite common for applications to have implementation limitations that cannot be expressed in their interfaces. For example, a fileserver might have an arbitrary limit on the size of a file, or a printserver an arbitrary limit on the complexity of a postscript job. Successful implementors will try to minimize the number of times these limitations actually hamper their clients.

5.3 How are methods removed?

Say that an abstraction T evolves over a period of time through versions T_1_0 to T_1_7. At the time T_1_0 is defined, a method M1 is specified. When T_1_1 is defined, a new and improved method M2 is added, which subsumes the functionality of M1. As time passes, all the clients of T are upgraded to use M2, and there are no longer any clients of M1.

When and how can we remove M1?

Unfortunately, within our versioning rules, M1 can only be removed by creating a major revision of T, since it could theoretically break some client application somewhere. A major revision has the necessary, but undesirable, property of appearing to all client applications as a new and distinct type from the previous minor revisions. So, removing M1 will break existing client applications of T, even though none of these applications actually uses M1.

The practical consequence of this is that obsolete methods tend to stay in the type system, even though some implementations may actually dishonor any calls on these methods, and raise exceptions if they are used. Although this is unfortunate, we consider it preferable to breaking the basic versioning rules.

5.4 Types as versions versus types as abstractions

We are using the type system for two distinct purposes. Firstly, we are using it to express types that are distinct abstractions. Secondly, we are using it to express different versions of particular abstractions.

In many ways, this overloading of a single mechanism is convenient. However, there is also a danger that users may confuse the two properties. It is important to recognize when the type system is being used to define new versions, and when it is being used to define whole new abstractions.

In practice, our syntactic conventions for type names make it fairly easy to distinguish between types which are merely new versions, and types which are intended as new abstractions. For example, it is easy for programmers to recognize that the type thread_1_2 is merely a new version of thread_1_1, rather than a brand new kind of thread abstraction.

However, it may be useful to add some more syntactic support for versioning, so that the different versions of an abstraction can be grouped into a single syntactic unit, rather than being scattered about as apparently distinct entities.

5.5 Managing the version space

Our versioning conventions let us express versions within the type system. However, we are still left with the issue of allocating version numbers. For example, if two distinct groups were both to decide that they wanted to add new methods to the type address_space_1_2, it would be extremely undesirable if they were both to call their new type address_space_1_3.

We expect that this will be solved by social conventions. By default, the group that first defines an abstraction should have the sole right to define new versions of that abstraction. If other groups wish to add methods without agreement from the owning group, then they must invent a new type name, such as foo_address_space_1_3, to reflect their new, distinct lineage.

5.6 Future directions

Currently, our main concern about the type versioning solution is that it requires rather too many keystrokes to create new versions. In order to add a new version, it is necessary to define a new type which inherits from the last revision of the abstraction. While continuing fundamentally to represent revisions as types, we plan to add some syntactic sugar to our interface definition lan-

guage, so that minor revisions can be expressed more conveniently, and are less syntactically isolated from earlier revisions.

We would like to permit implementations to support multiple major revisions of an abstraction. Thus, a given implementation of T might choose to support a type which inherits from both T_1_2 and T_2_3. This can potentially reduce the pain of creating major revisions by enabling implementations to inherit from (and support) two separate major revisions. Unfortunately, different major revisions of an interface tend to use the same method names for some methods, and our server side stubs cannot currently cope with multiple inheritance of a method name—a limitation we intend to remove.

6 Conclusions

Using an object-oriented interface definition language has proven a major asset in defining the Spring system interfaces.

We have been able to express interface evolution using interface inheritance. Interface inheritance has proven a very good match for the kind of properties we require when defining a compatible revision of an interface. Through the limited use of a run-time typing mechanism, we have been able to inquire about different versions at run time, and if appropriate, make different decisions based on which versions of interfaces other modules support.

The use of multiple inheritance at the leaves of our type tree has provided us with a satisfactory style for adding in optional properties fairly widely across the system, without requiring that they be everywhere, and without providing a bias towards a pre-ordained set of properties.

7 References

[Birrell & Nelson 1984] A. D. Birrell and B. J. Nelson. “Implementing Remote Procedure Calls”. ACM Trans. on Computer Systems, 2(1), February 1984.

[Bjornerstedt & Hulten 1989] Anders Bjornerstedt and Christer Hulten. “Version Control in an Object-Oriented Architecture”. In “Object-Oriented Concepts, Databases, and Applications”, edited by Won Kim and Frederick H. Lochovsky, ACM Press, 1989.

[Bobrow et al. 1988] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales and D. A. Moon. “Common Lisp Object System Specification”. ACM SIGPLAN Notices, Volume 23, Special Issue, September 1988.

[Bracha & Cook 1990] Gilad Bracha and William Cook. “Mixin-based inheritance”. OOPSLA’90 Conference on Object-Oriented Programming Systems, Languages and Applications, Ottawa, October 1990.

[Cardelli & Wegner 1985] Luca Cardelli and Peter Wegner, “On understanding Types, Data Abstraction and Polymorphism”. ACM Computing Surveys, 17(4), December 1985.

[Ellis & Stroustrup 1990] Margaret A. Ellis and Bjarne Stroustrup. “The Annotated C++ Reference Manual”. Addison-Wesley, 1990.

[Goldberg & Robson 1983] A. Goldberg and D. Robson. “Smalltalk-80: The Language and its Implementation”. Addison-Wesley, 1983.

[Khalidi & Nelson 1993] Yousef A. Khalidi andMichael N. Nelson. “An implementation of Unix on an object-oriented operating system”. Usenix Conference Proceedings, San Diego, January 1993.

[Meyer 1991] Bertrand Meyer. “Eiffel: The Language and Environment”. Prentice-Hall, 1991.

[OMG 1991] Object Management Group. “Common Object Request Broker Architecture and Specification”. OMG Document Number 91.12.1.

[Ossher & Harrison 1992] Harold Ossher and William Harrison, “Combination of Inheritance Hierarchies”. OOPSLA’ 92 Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, October 1992.

[Putz 1983] Steve Putz, “Managing the evolution of Smalltalk-80 systems”. In “Smalltalk-80: Bits of History, Words of Advice”, edited by Glenn Krasner, Addison-Wesley, 1983.

[Schaffert et al. 1986] C. Schaffert, T. Cooper, B. Bullis, M. Killian and C. Wilport. “An introduction to Trellis/Owl”. OOPSLA’86 Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, September 1986.

[Skarra & Zdonik 1986] Andrea H. Skarra, Stanley B. Zdonik. “The Management of Changing Types in an Object-Oriented Database”. OOPSLA’86 Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, September 1986.

[Wegner & Zdonik 1988] P. Wegner and W. B. Zdonik. “Inheritance as a mechanism for incremental modification”. ECOOP’88 European Conference on Object-Oriented Programming, Oslo, 1988.