Aspects: Extending Objects to Support Multiple, Independent Roles

Joel Richardson Peter Schwarz
IBM Almaden Research Center
San Jose, California 95120-6099

Abstract

The type systems of most object-oriented database systems (OODBSs) descend from traditional object-oriented programming languages. While these systems can capture many interesting relationships among entities, such as classification and subtyping, they do not typically allow an object to change type, and they only partially support the modelling of objects that have many types. However, such characteristics are particularly common among the very entities (e.g., people) that these systems are intended to model. We introduce *aspects*, a new mechanism designed to meet these modelling requirements within the framework of a strongly-typed OODBS. An aspect extends an existing object with new state and new behavior while maintaining the same object identity. In addition to the modelling of roles, aspects have other interesting applications, such as encapsulating the result of a query.

1 Introduction

Objects have become an important influence in computer science. Many database systems researchers believe that object-orientation will help to overcome some of the modelling limitations of relational systems [1, 6, 10, 12, 13, 15]. By combining state and behavior in a single abstraction, objects allow the schema designer to model conceptual entities more directly. Moreover, by supporting the notion of subtypes, object-orientation captures behavioral classification.

1.1 Object Identity and Changing Types

Most object-oriented database systems display serious shortcomings, however, in their ability to model both the dynamic nature and the many-faceted nature of common, real-world entities. The most obvious example of this kind of entity is a person. While existing OODBSs may capture the notion that a student *is* a person, they do not support the notion that a given person may *become* a student. After graduation, that person ceases to be a student, and becomes an alumnus. In the meantime, he or she may also be an employee, a customer, a club member, etc. Throughout his or her life, a person gains and loses many roles.

While researchers in knowledge representation systems have been aware of this problem for some time [4, 11, 20], it has only recently begun to receive attention from the OODBS community [10, 17, 21]. In most OODBSs, the intimate and permanent binding of an object's identity to a single type inhibits the tracking of real-world entities over time. This represents more than an inconvenience; in fact, it is a critical problem, since one is forced to model entities that evolve dynamically with objects that cannot. If one simulates such entities with a collection of individual objects, then there is no longer a direct correspondence between conceptual entities and objects.

It is important to note that many kinds of entities other than persons have roles that change over time. For example, an automobile may at one time be a privately owned vehicle, later be registered as a commercial vehicle in a taxi service, and finally become state's evidence in a hit-and-run trial. The Golden Gate Bridge was first realized as a legal entity, then as a design, and finally as a physical entity. A chip design may have realizations as a specification, a logic diagram, and a silicon layout. A programming environment may represent a single program as a character string, a structured document, a parse tree, and an executable module. In all of these examples, a single conceptual entity has different perspectives that come and go with time. In modelling such an entity, we would like to add and delete interfaces (with supporting implementations) while maintaining a single object identity.

1.2 Multiple Inheritance and Multiple Types

The kinds of entities that we are discussing do not simply evolve from one type into another. At any given moment, an entity may have many different types that are not necessarily related. Earlier, we mentioned persons who may simultaneously be employees, customers, club members, etc. In most OODBSs, one must use multiple inheritance to model objects that have many types at once. This approach can lead to a combinatorial explosion of sparsely populated classes [16]. For example, to create an object that is both a student and an employee, one would first define a new class, StudentEmployee, that inherits from both Student and Employee. The sole purpose of such an "intersection class" [17] is to allow an instance to be of multiple types; it adds no new state or behavior.

Another problem with multiple inheritance is that it provides only a single behavioral context for an object. This restriction leads to name conflicts. For example, both Student and Employee may define methods named department which return, respectively, the student's major department and the department in which the employee works. These two methods cause a name conflict when one defines the StudentEmployee class. All systems that support multiple inheritance define rules for resolving or avoiding such conflicts. At best, this is a burden on the programmer or schema designer who must understand the rules, which can itself be a surprisingly difficult task. At worst, it can lead to unexpected behavior when the object is passed into superclass contexts. For example, some systems, e.g. CLOS [7] and Orion [1], resolve method name conflicts by imposing an order on superclasses; the new class then inherits the method defined by the closest superclass in the ordering. Suppose classes A and B both implement method m, class C inherits from A and B, and A precedes B in the ordering. Then if a C instance is passed into a B context and m is invoked, A's implementation of m will be called.

1.3 Aspects

This paper presents *aspects*, a mechanism designed to address the problems raised in this introduction. An aspect extends an existing object with additional state and behavior while sharing the same object identity. An object may have many aspects that come and go over time. Rather than being an instance of some unique subclass defined through multiple inheritance, an object simply *is* an instance of many types by virtue of having many aspects. Every object reference is to a particular aspect, and the behavior of the object depends on which aspect is being referenced.

As we noted earlier, we are not the first to observe the problems discussed in this section. The main contribution of this work is to demonstrate that a clean, integrated solution is possible in a strongly typed environment. Furthermore, we show that combining aspects with a type system based on conformity [2] leads to a particularly powerful combination having applications beyond the modelling of multiple changing roles. For example, aspects can be used to encapsulate the results of queries in an OODBS or to help evolve software components.

The remainder of this paper is organized as follows. Section 2 describes the basic model of types and objects and provides a framework for the rest of the paper. Section 3 describes our mechanism and shows how it addresses the problems raised in this introduction. Section 4 discusses some open problems in our design. Section 5 reviews other work related to the issues of multiple roles and context-dependent behavior, and Section 6 concludes the paper.

2 The Object Model

Our model of types and objects was inspired by the Emerald language [2]. Like Emerald, we make a sharp distinction between abstract data types and implementations. An *abstract type* is a set of method signatures that specifies an interface. An object is defined by an *implementation*, which specifies a representation for the object and code for its methods. A particular object implements an abstract type if it provides at least the interface specified by that type.

Figure 1 shows the definition of the abstract type Person and an implementation named personImpl.[1] The

```
/* an abstract type specification for Person */
type Person
{
    String name();
    int age();
    String phone();
};

/* an implementation of Person */
implementation personImpl
{
    /* representation */
    String myName;
    int myAge,
    String myPhone;
public:
    /* operations */
    String name() { return myName; };
    int age() { return myAge; };
    String phone() { return myPhone; };

    /* constructor */
    personImpl( String n, int a, String p )
    { myName = n; myAge = a; myPhone = p; };
};

/* Example usage. Declare an identifier. */
Person joeP; /* initially nil */

/* Create an object representing Joe. */
joeP = personImpl("Joe", 72, "555-1234");

/* Print "Joe" */
stdout.putstring( joeP.name() );
```

Figure 1: A Type and Implementation for Person

abstract interface defined by Person includes methods to return the person's name, age, and home phone number. The implementation, personImpl, defines a representation in which the name and phone number are strings and age is an integer. The public interface to an instance of personImpl includes methods for returning the person's name, age, and phone. The implementation also provides a constructor for creating and initializing personImpl instances.

The code following the body of the implementation shows an example of creating an instance and invoking a method. We declare the identifier joeP with the abstract type Person. Next we create Joe by invoking the personImpl constructor, supplying it with initializing parameters. The assignment of the object returned by the constructor to the identifier joeP is legal because the object exports an interface that satisfies the definition of Person. Finally, we print Joe's name.

It is important to understand that personImpl is an implementation of Person not because of any declared rela-

[1] The examples in this paper use a syntax in the style of C++[22]. We emphasize, however, that this is for illustrative purposes only; we have not yet defined a language syntax.

tionship, but because it exports an interface that satisfies Person This is a crucial distinction because it implies that a given type may have many implementations and, conversely, a given object may be an implementation for many types [2]. For example, we can define an alternate implementation of Person that stores the person's date of birth, dob, and then implements the age function by computing the difference between today's date and the dob. Both implementations can coexist in the same system.

While the distinction between abstract types and implementations provides great flexibility to the programmer, our syntax requires extra (and somewhat redundant) coding in order to define a type and an implementation together. As a convenience, we introduce a shorthand form that names an abstract type along with an implementation. In an implementation header, the optional clause with type *name* causes the creation of an abstract type *name* whose interface is the interface exported by the implementation (excepting the constructor). Thus, Figure 1 could be rewritten by removing the definition of Person and changing the implementation header to:

```
implementation personImpl with type Person { . };
```

Note that this construct is merely a shorthand for the purposes of this paper, and does not imply any special binding of the implementation to the type.

We now define more precisely the rules for type compatibility. Abstract types are related to each other and to specific implementations by *conformity* [2, 3]. In the case of an implementation, we refer to the abstract type implicitly defined by its exported methods as the *induced abstract type* of the implementation. Conformity defines an implicit relationship between interfaces. Informally, an abstract type B *conforms* to an abstract type A if B provides all of the method signatures specified by A (and possibly more) and if the corresponding signatures conform in the proper way. Specifically:

1. For each method signature specified in the definition of A, B specifies a signature with the same name, the same number of arguments, and the same number of return values.
2. For each such signature of B, the abstract type of each return value must conform to the abstract type of the corresponding return value of the corresponding signature of A.
3. For each signature of A, the abstract type of each argument must conform to the abstract type of the corresponding argument of the corresponding signature of B.

These rules are very similar to the subtyping rules of Trellis/Owl [18]. A formal definition of conformity and an algorithm for testing the conformity of two interfaces are given in [3].

In Figure 1, the implementation personImpl conforms to the abstract type Person. Figure 2 shows the definition of another abstract type, PhoneOwner. The implementation personImpl and the abstract type Person both conform to abstract type PhoneOwner. PhoneOwner does not conform to Person because it does not include a signature for the age method. In program code, identifiers are declared with abstract types. An expression having abstract type B can be assigned to an identifier of abstract type A if and only if B conforms to A. Because it is possible to determine the abstract type of every expression at compile time, the language is statically typed. In the code of Figure 2, the value of joeP can be assigned to cust1 because Person conforms to PhoneOwner, but the value of cust1 cannot be assigned to someone because PhoneOwner does not conform to Person. In the third example, the induced abstract type of personImpl conforms to PhoneOwner, and thus the assignment is legal.

```
type PhoneOwner
{
    String name();
    String phone();
};

/* example usage */
PhoneOwner cust1;
Person someone;

/* OK: Person conforms to PhoneOwner */
cust1 = joeP; /* the guy from Fig. 1 */

/* Error: PhoneOwner doesn't conform to Person */
someone = cust1;

/* OK: value returned conforms to PhoneOwner */
cust1 = personImpl("Fred", 27, "555-1830");
```

Figure 2: A Phone Owner Abstract Type

Like the more conventional approaches to subtyping found in such languages as C++ [22] and Trellis/Owl [18] and in such OODBSs as O_2 [13] and Orion [1], conformity allows the programmer to define a general type and a set of more specialized types that share the general behavior.[2] Objects that conform to the specialized types may be substituted in any context where an object conforming to the more general type is expected. The difference is that with conventional subtyping, the relationships among types are explicitly declared: type B is declared to be a subtype of type A. With conformity, any type that supplies the required interface is implicitly a subtype, and may be used as such. This capability is especially useful when a more general type is defined *after* several more specific types already exist. Conformity gives the programmer the freedom to introduce new supertypes into the type lattice without having to modify the definitions of existing types.

3 Aspects

Abstract types, implementations, and conformity define our basic object model. We extend this model to support aspects. The combination of type equivalence through conformity and object extension through aspects results in a particularly powerful mechanism that easily supports the modelling of roles.

[2] Here, we are referring only to substitutability. In our model, inheritance is orthogonal to substitutability. We do not consider inheritance in this paper.

3.1 An Example

The language construct for defining an aspect is a variation of that used to define an implementation. Syntactically, the definition of an aspect is an implementation augmented with the abstract type of the base object that is to be extended. Figure 3 shows the definition of an

```
/* an employee aspect definition */
implementation empImpl
    with type Employee
    extends Person
{
    /* added employee state */
    int myEid,
    String myDept;
    String myPhone,
public:
    /* constructor */
    empImpl( int e, String d, String p )
        { myEid = e, myDept = d, myPhone = p; };

    /* operations */
    int eid() { return myEid, },
    String dept() { return myDept, },
    String phone()
    {
        if( myPhone != nil )
            return myPhone;
        else
            return base.phone();
    },

    /* operations exported from Person */
    Person.name;
    Person::phone as homePhone;
};
```

Figure 3: An Employee Aspect of a Person

employee aspect. The extends clause in the implementation header indicates that any object conforming to the abstract type Person may acquire an Employee aspect implemented by empImpl. Invocation of the empImpl constructor creates this new aspect and returns a reference to it.

As mentioned earlier, an aspect may add state, may define new behavior, and may export selected parts of the base object's interface. Our example adds state variables to record the employee's identification number, department name, and office phone number. Note that if a Person implemented by personImpl is supplied as a base object, there is no conflict between myPhone defined in personImpl and myPhone defined in empImpl. The reason is that the employee aspect implementation only sees the *abstract* interface of the base object; it has no special privilege to access its representation.

The methods defined for the employee aspect include those to access the employee's representation as well as two that are exported from the base object. The methods eid and dept are straightforward and need no explanation. The method phone illustrates the ability of methods in an aspect to invoke methods on the base object. In this case, invoking phone on an employee aspect will return the office phone number if there is one. Otherwise, it will return the person's home phone number. In general, an aspect method may access the base object through the implicitly defined reference base. This is in addition to (and analogous to) the implicit reference this[3] that is typical in object-oriented programming. In our model, this refers to the particular aspect on which the method is invoked.

In addition to defining new behavior, an aspect definition also controls how much of the base object's interface "shines through" in the new aspect. A given method of the base object may be exported in turn by the aspect; it may also be renamed or hidden altogether. Our employee example exports two methods from the Person interface, name and phone. Note that name is exported directly, while phone is renamed to homePhone to avoid a clash with the locally defined method. The syntax for exporting and renaming methods of the base object is really only a shorthand for explicitly defining a method that invokes the appropriate method on the base. For example, homePhone can be equivalently defined as:

```
String homePhone(){ return base.phone(); }
```

Figure 4 shows how the aspect implementation of Figure 3 may be used to extend a person with a new employee aspect. First, we create a person named "Joe", referenced

```
/* example usage. Create a Person, Joe. */
Person joeP = personImpl("Joe", 72, "555-1234"),

/* extend Joe with Employee information */
Employee joeE =
    empImpl(10139, "Toy", "555-5678") extends joeP;

/* joeE and joeP are same oid, but not identical refs. */
Bool b1 = ( joeE @= joeP ), /* TRUE */
Bool b2 = ( joeE == joeP ); /* FALSE */

/* Employee does not conform to Person */
Person p = joeE; /* ERROR */

/* Person and Employee both conform to PhoneOwner */
PhoneOwner po1 = joeP; /* OK */
PhoneOwner po2 = joeE; /* OK */

/* Employee exports Person::name method */
stdout putstring( joeP name() ); /* "Joe" */
stdout.putstring( joeE.name() ), /* "Joe" */

/* Employee and Person have different phone methods */
stdout.putstring( joeP phone() ); /* "555-1234" */
stdout putstring( joeE phone() ); /* "555-5678" */
stdout putstring( joeE homePhone() ); /* "555-1234" */
```

Figure 4: Creating and Using an Employee Aspect

by the variable joeP. We then give Joe an employee aspect, assigning him to the toy department. The extends clause in the invocation of the aspect constructor empImpl supplies the reference to the base object that is being extended. The invocation can be statically typed because

[3] This reference is called self or me in other languages.

the abstract type of JoeP, the induced abstract type of emplmpl, and the abstract type of JoeE are all known at compile time. The new aspect becomes an extension of Joe, sharing the same identity. One may interpret the variables joeP and joeE as being references to "Joe as a person" and "Joe as an employee," respectively.

Because of the distinction between different aspects and different identities, we need two operators for comparing references [21]. The == operator means "are identical references" and holds only for references to the same aspect of the same object. The @= operator means "are references to same object" and is true for references to any two aspects of the same object. In our example, joeP and joeE are @= but not ==.

The next few lines of Figure 4 illustrate an important point about the relationship between aspects and conformity. Because of the possibility of renaming or hiding, an aspect may not be substitutable for the base object. In our example, Person exports an age method that is hidden by the employee aspect. Thus the abstract type Employee does not conform to Person, and the assignment of joeE to the Person variable p is flagged as a type error. The following two assignments are legal, however, since Joe can act as a phone owner in both his person and employee roles.

The remainder of Figure 4 shows various invocations of methods on the two aspects of Joe. Because the name method of Person shines through the Employee interface, this method can be invoked through either reference to Joe. However, joeE.phone() and joeP phone() invoke different methods since the employee aspect defines its own phone method. This example illustrates an important difference between extending an object with an aspect and creating an instance of a subtype in a traditional object-oriented system. The addition of an aspect to a base object does not override methods defined in the base object. This is especially important because aspects are dynamically created for long-lived objects, and there may be outstanding references to the base object. In our example, the fact that Joe becomes an employee should not suddenly cause all of his phone calls to be routed to the office.

Aspects also avoid the problems encountered in using multiple inheritance to define intersection classes. Figure 5 shows the outline of a student aspect that defines dept, a method that returns the student's major department. The example shows Joe the Person (who also has an employee aspect) being extended with a student aspect. We did not have to create an intersection class, as we would with multiple inheritance. Joe is *both* a Student and an Employee. Although Employee and Student both export a dept method, there is no name clash since these are independent aspects. We may invoke dept on Joe in his Student role to obtain his major department, and in his Employee role to obtain the department where he works. Finally, note that the three variables, joeP, joeE, and joeS, are all references to the same object, and thus they are all pairwise @=. However, they are all references to different aspects of Joe, so no two are ==.

In general, an object with all of its aspects forms a tree, i.e., an aspect may serve as a base object for further extensions. This is because an aspect, like any implementation, is simply one realization of an abstract interface. For example, the abstract type Manager could be implemented by an aspect that requires an object conforming

```
/* implementation of student aspect */
implementation studentImpl
    with type Student
    extends Person
{
    String majorDept;
    ...
public:
    String dept() { return majorDept, };
    ...
};

/* make joe a Student */
Student joeS = studentImpl("Biology", .. ) extends joeP,

/* print joe's departments */
stdout putstring( joeS.dept() ); /* "Biology" */
stdout putstring( joeE.dept() ), /* "Toy" */
```

Figure 5: Adding a Student Aspect

to Employee as a base. The base object could be implemented by an emplmpl or by any other object conforming to Employee. A given base object may also have multiple aspects of the same "type," i.e., created by the same aspect constructor. Thus, a person could be an employee of more than one company, a student at more than one school, etc. One could also use multiple aspects to represent multiple versions of an object [17].

3.2 Aspects and Conformity

Aspects and conformity allow the modelling of roles that can be played by different types of entities having some common behavior. For example, in modelling a telephone company database, the role of a customer can be played by a person or a business. Persons, businesses, and other phone owners have a limited set of attributes in common, e.g. a name and a telephone number. For each customer, regardless of kind, the database should maintain some additional information (e.g. a record of calls), and support some operations appropriate to customers, such as bill or disconnect.

Figure 6 shows how one could model this example by using aspects. The PhoneOwner abstract type from Figure 2 describes objects that have the minimal set of methods required to act in the role of a customer: name and phone. The Person abstract type of Figure 1 represents one kind of potential phone owner; Figure 6 shows type Company (and an implementation, companyImpl) that describes another kind of potential phone owner. Both Person and Company conform to PhoneOwner. We add another abstract type, PhoneCustomer that defines the interface for the customer aspect, exporting the underlying PhoneOwner methods (since they are generally useful) as well as bill, disconnect, etc. The corresponding aspect implementation, phoneCustImpl defines an aspect constructor that accepts any object conforming to PhoneOwner as the base object. It returns an aspect conforming to PhoneCustomer; the aspect augments the representation to include the call history and supplies code for the added methods.

The combination of aspects and conformity also ad-

```
/* An implementation of a company. */
implementation companyImpl with type Company
{
    ...
public:
    ...
    String name() { ... };
    String phone() { ... };
},

/* An implementation of phone customer. */
implementation phoneCustImpl
    with type PhoneCustomer
    extends PhoneOwner
{
    /* added customer state */
    List<Call> myCallHistory;
    ..
public:
    /* new operations */
    float bill() { ... };
    void disconnect() { ... };
    . .

    /* exported from PhoneOwner */
    PhoneOwner. name;
    PhoneOwner::phone;
};

/* Create a company. */
Company acme = companyImpl( ... );

/* Make the both Joe and acme phone customers. */
PhoneCust pc1 = phoneCustImpl( ... ) extends joeP;
PhoneCust pc2 = phoneCustImpl(  . ) extends acme;
```

Figure 6: Using Aspects to Model Roles

dresses another problem. In a large evolving system, one can expect that multiple programmers will independently develop types that describe similar abstractions. At some point, it may be desirable to interchange instances of such types, but for a number of reasons, they may not conform to each other. The same conceptual method may have inadvertently been given different names in the different specifications, e.g. Display instead of Print. The ordering of arguments for a common method may differ, or one type's definition may specify additional arguments that were not required in the other definition. In any of these cases, an aspect can be used to smooth over the differences between the types. To each base object that conforms to one of the types, one can add an aspect that conforms to the other type's specification. The aspect constructor defines the required methods in terms of those defined by the type of the base object. If necessary, the aspect can also augment the object's representation with additional state to help implement the new interface. Using aspects in this manner allows schemas to evolve in simple ways, e.g., by adding stored or computed attributes, renaming methods, or removing methods from the interface. An advantage of this approach is that existing programs which rely on the old interface continue to work. A disadvantage is that one must explicitly construct the new aspect for each existing instance.

3.3 Aspects and Queries

Aspects provide a convenient mechanism for encapsulating the tuples resulting from a query. Assume, as in ENCORE [19], the existence of a join operator that accepts two sets of objects and a join predicate and returns a set of two-tuples. Each tuple references a pair of joined objects and provides methods to obtain them. For example, suppose Employee objects contain information about an employee's skills, and Department objects contain information about skills required by the department's projects. In considering potential employee transfers, one might wish to create an object for each matchup of an employee's skills and a department's skill requirements. Each tuple produced by the join refers to an Employee object and a Department object, and exports corresponding methods, emp and dept, that return them. Because the only methods for the tuples are the ones that return the underlying objects, the structure of the tuples is exposed. However, aspects can be used to encapsulate these tuples. Each tuple can be extended with an aspect that reveals selected methods of the underlying objects, and perhaps defines some additional methods as well.

Figure 7 shows how this could be done for the example described above. First, we define EmpDept, the abstract type of the tuples created by joining an employee and a department. Next, we define Transfer, an abstract type representing a potential employee transfer, with methods empName and newDeptName for obtaining the employee's name and potential new department, and doTransfer to make the required updates if the transfer is to take place. Thirdly, we define an implementation that extends an object conforming to EmpDept with a Transfer aspect. The final portion of the example shows how the constructor is used in conjunction with two of ENCORE's query algebra operators. The Ojoin operator, which takes as arguments two sets of objects and a join predicate, creates the set of EmpDept tuples from the sets emp and dept. The Image operator applies a function to every object in a set, returning the set of output values. Here, it is used to apply the transferImpl aspect constructor to each tuple produced by Ojoin, creating a set of objects conforming to Transfer.

To the type system, the objects produced by this query are indistinguishable from conforming objects produced by other queries or methods. For example, one might create other objects conforming to Transfer by matching employees and departments based on some entirely different criterion.

4 Open Problems

In addition to the obvious questions of implementation, there are a number of other issues that we have not yet addressed, both regarding the aspect mechanism and also concerning the larger topics of the overall type system and data model. We consider some of these issues briefly in this section.

4.1 Aspect Discovery

Given that an object may have many independent aspects, each of which is intended to model a particular role,

```
/* Type created by joining employee and department */
type EmpDept
{
    Employee emp(),
    Department dept();
}

/* An abstract type representing an employee transfer */
type Transfer
{
    String empName(),
    String newDeptName();
    void doTransfer();

};


/* An aspect constructor for a transfer  */
implementation transferImpl extends EmpDept
{
public:

    String empName() { .. },
    String newDeptName() { .. };
    void doTransfer(), { . . },
},

/* Join employees and departments */
set<EmpDept> skillMatchTuples =
    Ojoin(emp, dept,
        λeλd. Intersection(e.Skills, d.RequiredSkills) != {}
        ),

/* Encapsulate tuples by creating Transfer aspects */
set<Transfer> skillMatches =
    Image(skillMatchTuples, λs. transferImpl() extends s),
```

Figure 7: Encapsulating the Result of a Query

it is valuable to be able to determine what aspects an object currently carries. For example, when a person checks into a hospital, the receptionist will ask whether that person is insured. In our aspect model, this is equivalent to being handed a reference to a Person and asking if that object carries the aspect Insured.

There are at least two ways in which a language might provide this capability. The more general mechanism would allow programs to ask the question "What aspects does this object carry?" Since the possible answers are not known at compile time, the program must receive and interpret a runtime descriptor of the object and its aspects. A simpler construct would allow a program to ask if the given object carries a specific aspect. If the answer is yes, then a reference to that aspect of the object would be returned. Since the abstract type of the desired aspect is stated explicitly in the code, the program can receive a typed reference to the aspect, avoiding the need to interpret runtime descriptors.

4.2 Dropping Aspects

The aspect mechanism permits an existing object to acquire new context-dependent behavior. Support for evolving abstractions also requires a mechanism by which aspects can be dropped. Whereas adding a new aspect has no effect on existing programs, arbitrarily dropping an aspect could potentially cause dangling references in existing programs or persistent data structures. This problem is not unique to aspects; it arises in any system in which entities can be explicitly deleted. The most conservative approach would be to forbid explicit aspect deletion; an aspect could only be garbage-collected when no outstanding reference to the aspect exists. Since an aspect discovery mechanism would make it possible to access any aspect of an object given a reference to any other aspect (or to the base object), one must decide whether such implicit references should be considered for garbage collection purposes.

At the opposite end of the spectrum, one could permit aspects to be dropped arbitrarily, leaving behind a "tombstone" that would cause an exception whenever a program tried to refer to the deleted aspect. If one deletes an aspect that has been extended with additional aspects, the dependent aspects must also be deleted, because their implementation may depend on the methods of the aspect that they are extending. These aspects may in turn serve as bases for further extensions, which must also be deleted. Similarly, when a base object is deleted, all of its aspects must be deleted.

Because we believe that explicit deletion is needed in large systems to allow resources to be reclaimed in a timely manner, we favor the explicit deletion approach. To reduce the impact of deletion on unsuspecting programs, however, we are considering a mechanism that permits explicit deletion but allows programmers to specify when aspects can be dropped. Consider the following example. When a parent wishes to obtain a babysitter for an evening, he or she consults a list of babysitters and begins to make telephone calls. If one of the potential babysitters informs the parent that he or she is no longer interested in babysitting, the parent simply moves on down the list. That is, the parent is able to handle the "I'm no longer a babysitter" exception at that point. On the other hand, once the parent has retained a particular babysitter for an evening, he or she would be very displeased to return home and find that the babysitter gave up on the profession sometime during the evening (turning, perhaps, to witchcraft or politics instead). To prevent this, the parent enters into a contract with the babysitter for the duration of the evening.

We believe that this paradigm is applicable to many kinds of programs. At certain times, the program is prepared to respond to an exception indicating that an aspect no longer exists. Defining an appropriate exception handler for each reference, however, would be cumbersome and there might well be circumstances under which the program would be unable to recover from the exception. Analogous to the babysitter example, we are exploring mechanisms by which a reference to an aspect can be placed under a *contract* that prevents the referenced aspect from being arbitrarily dropped. Just as there are many ways in which real contracts can be terminated, one can envision various ways in which a contract between an aspect and a reference could be terminated, including explicit release by the holder of the reference, asynchronous notification of the holder, timed expiration, etc. We intend to explore various contract mechanisms as we continue to refine our type system.

4.3 Aspects and Authorization

Another area for future work concerns the interaction between aspects and authorization. Aspect creation, aspect discovery, and aspect deletion must all be subject to authorization constraints. For example, in the real world, becoming a student requires an agreement between the person and the university; neither party can unilaterally cause that event. Furthermore, the student information belongs to the university, even though the role is an aspect of the person. While the student has some rights to the information, he or she is not allowed to perform certain updates, say, to change a grade. Discovery of aspects must also be controlled. Unconstrained discovery could reveal information about an entity that should be protected, e.g. that an Employee is also a Patient or a Defendant. As argued in the previous section, dropping an aspect must be controlled so that clients of an aspect can operate with some guarantee of continuity. Finally, authorization over aspects is needed from a systems administration point of view, since creating an aspect consumes resources.

4.4 The Melampus Data Model

We believe that multiple, changing roles are typical of long-lived entities and that primitives to model such behavior are crucial to a coming generation of "omniscient" information systems. These geographically-distributed multi-user systems will provide a universal repository for both long- and short-lived data. All information in such a system will be accessed and managed in a uniform way, and the system will provide both navigational and associative access to the objects it contains. Guided by this vision, we have begun to design such a system, which we call Melampus [5]. Aspects are only one part of the overall data model for Melampus. We have also been developing a query algebra and designs for authorization and naming.

While the type system will be based on conformity, there are additional areas that must be addressed before its design is complete. As we have noted, conformity makes substitutability implicit. We believe, however, that programmers should also have a mechanism by which they can explicitly indicate their intent that a type being defined is to be substitutable for a previously-defined type. When this mechanism is used, the compiler can assist the programmer by verifying that the interface of the new type conforms as intended.

Furthermore, although conformity provides a mechanism for subtyping, it does not provide any support for inheritance. That is, each implementation that conforms to an abstract type is potentially independent. Any code shared by more than one implementation must be repeated in each implementation. An inheritance mechanism would allow the programmer to make the relationship between implementations explicit, with shared code specified only once.

It may also be useful, under some circumstances, to make the relationship between a type and a particular implementation explicit and exclusive. A mechanism that allows a programmer to designate a specific set of implementations that can conform to an abstract type provides users of that type with a stricter guarantee that all instances will behave in a specified way.

5 Related Work

As we noted in the introduction, the problems that motivated aspects have been observed by other researchers, and several proposals have appeared in the literature. To the authors' knowledge, the earliest example was work by Bobrow and Winograd on perspectives in KRL [4]. These ideas also appeared later in PIE [11] and Loops [20]. Recently, these issues have sparked a renewed interest among researchers in OODBSs. In this section, we review some of the more relevant work and compare it with our own.

5.1 Iris

Iris [10] was the first OODBS to be based on a data model recognizing the importance of entities that can change type, and it provided the original inspiration that led to our aspect mechanism. In Iris, an object is fundamentally nothing more than an identity. An object may gain or lose types over its lifetime, but it retains its identity throughout. An object x is considered to be of a type T if the predicate T(x) evaluates to true. An explicit add-type operation adds a new type to an object; thereafter, any of the methods defined for that type can be invoked.

Since one can add arbitrary types to an object, Iris clearly supports the modelling of multiple, changing roles. However, Iris does not support context-dependent behavior for an object, and this leads to some difficulty. As in classical object-oriented models, a method defined by a type may be redefined by any of its subtypes. Method invocation normally chooses the implementation defined by the object's most specific type. However, since arbitrary types may be added to an object, there may not be a single most-specific type.[4] The semantics of such cases is not defined in the model and requires user-specified rules to resolve the ambiguity [10]. The problem is not that an object may have multiple types, but that the entire set of types that an object has is visible in every context. By contrast, aspects partition an entity into independent views, and each reference is *in* a particular context. Thus, without conflict, two aspects may provide different methods having the same name.

5.2 Object Hierarchies

Recently, Sciore proposed *object hierarchies* [17] as models for entities in the real world. Each object in the hierarchy represents some information about the entity. An object may contain some state variables and it may implement some behavior. In addition, it inherits behavior from its parent in the hierarchy. When a message is sent to an object, it either responds to the message directly, or it delegates [14] the message to its parent. Thus, the effective behavior observed by a client of an object depends on the particular arrangement of the object's hierarchy. In other words, inheritance is determined on a per-object basis. Various predefined methods are provided for manipulating object hierarchies, including methods for cloning an existing hierarchy and for extending one object with (part of) the behavior of another. *Template hierarchies* are introduced as a way to impart uniformity on the world

[4] This problem is essentially the name-clash problem of multiple inheritance.

of object hierarchies; each template plays the part of a prototype hierarchy from which instances may be cloned.

Although object hierarchies and aspects share similar motivations, there remain important differences. One distinction concerns types in the language. Sciore's model is based heavily on prototype languages [14, 23]. There are no types, and every object hierarchy is potentially unique in structure and behavior. Moreover, the structure of a given object hierarchy can change dynamically and arbitrarily. While the use of template hierarchies is an elegant way of ensuring uniformity of newly cloned objects, they do not completely address the problem. For example, the use of templates is purely advisory; one may still clone an arbitrary object. One may also arbitrarily change the structure of an object after it has been cloned from a template. For example, an object can change its parent to some other object, perhaps in a different hierarchy. Furthermore, there are no restrictions on which objects can become the parent. While such flexibility may be useful in some circumstances, it is not clear that a large computer system shared by many users can cope with such lack of structure. It would be very difficult to predict with any confidence that a message sent to an object will be understood.

By contrast, our model is strongly typed yet has significant flexibility. A given object may be extended only if it satisfies the abstract interface specified by the aspect definition. In terms of Sciore's model, this restriction would limit the parent of an object in a hierarchy to be one that responds to a known set of messages. Furthermore, each new aspect also exports a known abstract interface. Thus, each view of an object supports a predictable set of methods.

Another important difference between aspects and object hierarchies concerns object identity. An object hierarchy comprises many independent objects, each with its own identity, that collectively model a real-world entity. There is no one object identity that serves as the name for the entity being modelled. By contrast, an aspect is part of a single object that shares a single identity while supporting multiple, independent views. We believe this is an important distinction from a modelling standpoint, since one of the primary advantages of object-orientation is that it allows the structure of a program to reflect more directly the entities being modelled. This distinction between the models also has practical consequences. For example, in Sciore's model an object may detach itself from one hierarchy and reattach to another. Given that a hierarchy is a model for an entity, such an operation means that a role of one entity can become a role of a different entity. Such operations are disallowed for aspects, since they are intended explicitly to be views of a single object.

5.3 Clovers

Stein and Zdonik have proposed a mechanism called *clovers* [21] that is similar to both object hierarchies and aspects. This proposal is based on traditional notions of subtyping and inheritance. An object is created with a particular type T and may be extended to any subtype of T, say, T' The resulting object can be visually represented as a core of type T with an attached chunk containing the instance variables in (T' - T), i.e. those variables in the representation of T' that are not in T. An object having several such leaf-like extensions "looks" like a clover, hence the mechanism's name. The operator for extending an object is called become, and initial values for the instance variables in (T' - T) must be supplied as parameters.

Coercion operators are provided for moving among the various views of an object that exist at a particular time. The coerce- operator moves up the type hierarchy. For example, given a reference to the T' part of our clover, we can use coerce- to obtain a reference to the T part. The inverse, coerce+, moves down the hierarchy. Since every T' is also a T, coerce- is a type safe operation. However, coerce+ may produce a runtime error. As a way around this problem, the authors suggest a simple predicate, has, to test whether an object has an extension of a particular type before attempting the coercion.

Like object hierarchies and aspects, clovers were motivated by the need to model entities that have multiple, independent views. Consequently, a clover shares the basic semantic that different references to the same object may observe different behavior, and that duplicate method names in different "leaves" of a clover do not clash. As with aspects, an entire clover has a single object identity. And, like an object hierarchy, the behavior of any particular view is defined automatically through inheritance, although inheritance is at the object level in an object hierarchy and at the type level in a clover. The key difference between clovers and aspects is that each "leaf" of a clover must be a subtype of the type of the original object, whereas an aspect can be used to extend an object with arbitrary new behavior.

5.4 Summary

All of the work discussed in this section started from the same observation: existing object-oriented systems cannot capture the dynamic nature of the real-world entities that they seek to model. The Iris mechanism for changing types provided the original inspiration, although it proved to be too unstructured for our purpose. Sciore's object hierarchies are close relatives of aspects, although they lack identity semantics and strong typing. Stein and Zdonik's clovers mechanism casts many of these ideas into a traditional object-oriented type system. While they include identity semantics, their mechanism can only extend objects with subtypes. Aspects can extend objects with arbitrary behavior.

6 Conclusions

Modern object-oriented data models can capture two important properties of real-world entities: classification and identity. However, these models are not able to capture a third property: the evolution of entities through time. We have presented aspects, a mechanism that allows instances to gain and lose behavior dynamically, and we showed that this mechanism is consistent with both strong typing and object identity. We also showed how aspects, combined with a conformity model of types, is a powerful construct having many applications beyond the modelling of roles.

Acknowledgements

We would like to thank the other members of the Melampus project for many helpful discussions. Thanks especially to Jim Stamos and Felipe Cabrera for all the embarrassing questions and to Mike Carey for helping to improve this presentation.

References

[1] Banerjee, J., Chou, H.T., Garza, J.F., Kim, W., Woelk, D., and Ballou, N., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems,* 5(1), January, 1987, pp. 3-26.

[2] Black, A., Hutchinson, N., Jul, E., and Levy, H., "Object Structure in the Emerald System," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 78-86, Portland, OR, September, 1986.

[3] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Trans. on Software Engineering,* pp. 65-76, 13(1), January, 1987.

[4] Bobrow, D., and Winograd, T., "An Overview of KRL, A Knowledge Representation Language," *Cognitive Science,* 1(1), 1977.

[5] Cabrera, F., Haas, L., Richardson, J., Schwarz, P. and Stamos, J., "The Melampus Project: Toward an Omniscient Computing System", IBM Research Report RJ7515, June, 1990.

[6] Carey, M.J., DeWitt, D.J., and Vandenberg, S.L., "A Data Model and Query Language for EXODUS," *Proc. ACM SIGMOD Conf. on Management of Data,* pp. 413-423, Chicago, IL, June, 1988.

[7] Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., and Moon, D.A., "Common Lisp Object System Specification," *X3J13 Document 88-002R,* June, 1988. Printed in *SIGPLAN Notices,* vol. 23, special issue, September, 1988.

[8] *Eiffel: The Language*, Eiffel Language Manual, version 2.2, Interactive Software Engineering, Inc., December, 1989.

[9] Ellis, M. and Stroustrup, B., *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, MA, 1990.

[10] Fishman, D. H., et. al., "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Information Systems,* 5(1), January, 1987, pp. 48-69.

[11] Goldstein, I., and Bobrow, D.G., "Descriptions for a Programming Environment," *Proc. AAAI-1,* 1980.

[12] Hudson, S.E., and King, R., "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System," *ACM Trans. on Database Systems,* 14(3), September, 1989.

[13] Lecluse, C., Richard, P., and Velez, F., "O_2, an Object-Oriented Data Model," *Proc. ACM SIGMOD Conf. on Management of Data,* pp. 424-433, Chicago, IL, June, 1988.

[14] Lieberman, Henry, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 214-223, Portland, OR, September, 1986.

[15] Maier, D., Stein, J., Otis, A., and Purdy, A., "Development of an Object-Oriented DBMS," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 472-482, Portland, OR, September, 1986.

[16] McAllester, D., and Zabih, R., "Boolean Classes," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 417-423, Portland, OR, September, 1986.

[17] Sciore, E., "Object Specialization," *ACM Transactions on Office Information Systems,* 7(2), April, 1989, pp. 103–122.

[18] Schaffert, C., Cooper, T., and Wilpolt, C., "Trellis Object-Based Environment: Language Reference Manual," Digital Equipment Corporation Technical Report DEC-TR-373, November, 1985.

[19] Shaw, G.M., and Zdonik, S.B., "An Object-Oriented Query Algebra," *Proc. 2nd Int'l Workshop on Database Programming Languages,* pp. 111-119, Glenedon Beach, OR, June, 1989.

[20] Stefik, M., and Bobrow, D.G., "Object-Oriented Programming: Themes and Variations," *AI Magazine,* January, 1986.

[21] Stein, L.A., and Zdonik, S.B., "Clovers: The Dynamic Behavior of Types and Instances," Brown University Technical Report No. CS-89-42, November, 1989.

[22] Stroustrup, Bjarne, *The C++ Programming Language,* Addison-Wesley, Reading, MA, 1986.

[23] Ungar, D., and Smith, R., "Self: The Power of Simplicity," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 214-242, Orlando, FL, October, 1987.