JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 10, November-December 2004

Diamond Inheritance and Attribute Types in Timor

J. Leslie Keedy, Christian Heinlein and Gisela Menger, University of Ulm, Germany
Mark Evered, University of New England, Australia

Abstract

In Timor multiple inheritance of methods from a common abstract ancestor (e.g. Collection) and of separate "parts" (possibly repeatedly) from distinct supertypes (e.g. a Radio, a Cassette Player) are handled in different ways. The paper shows that neither technique is suitable for cases where a common concrete ancestor (e.g. Person) is specialised in different subtypes (e.g. as a Student, an Employee) and then brought together in a new subtype, possibly with repeated inheritance (e.g. a Doubly Employed Student). For such cases a new kind of type ("attribute types") is proposed, which provides an alternative programming paradigm to inheritance, based on the idea of adjectives and their use in noun phrases in natural languages.

1 INTRODUCTION

Timor[1] is a new OO language currently being developed at the University of Ulm with the primary aim of facilitating the development of programs and applications using components which have been separately designed and developed without knowledge of each other. In earlier papers two different ways of handling multiple inheritance in Timor have been presented.

The first paper [5] describes how subtypes in the Timor Collection Library (TCL), such as `Set`, `Bag` and `List`, can be derived from a common abstract ancestor (`Collection`). Multiple inheritance arises when orthogonal properties (here the ordering and duplication of collection elements) are combined. In this case it is natural to merge methods (e.g. `insert`) when they are inherited in a subtype from multiple supertypes at intermediate levels in the hierarchy.

The second paper [10] addresses a completely different kind of multiple inheritance, whereby a subtype (e.g. a type `RadioCassettePlayer`) can inherit "parts" from independent supertypes (e.g. a type `Radio` and a type `CassettePlayer`). Repeated inheritance can play a significant role (e.g. for a subtype `RadioDouble-`

Cite this article as follows: Leslie Keedy, Christian Heinlein, Gisela Menger, Mark Evered: "Diamond Inheritance and Attribute Types in Timor", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 121-142. http://www.jot.fm/issues/issue_2004_11/article2

CassettePlayer), creating a naming problem which Timor handles by allowing the supertypes to have "part identifiers" in the subtype definition, thus giving them an appearance analogous to variables declared by aggregation. The paper also discusses in detail the differences between this kind of parts inheritance and aggregation.

The present paper discusses the relationship between these two fundamentally different approaches to multiple inheritance, using examples which involve a common ancestor (e.g. Person) that can be specialised in orthogonal ways (e.g. as a Student and an Employee) and then brought together in a subtype involving diamond inheritance (e.g. EmployedStudent), and possibly also repeated inheritance (e.g. a DoublyEmployedStudent). It will be shown that neither of the approaches described in the earlier papers provides a satisfactory way of handling such examples.

A new kind of type, called an *attribute type*, is then presented. Attribute types support a programming paradigm which is based on noun phrases in natural languages rather than on inheritance. This approach leads to more modular units than conventional subtypes, and is especially useful in cases which would otherwise result in diamond inheritance problems. It is shown how different attributes can be flexibly mixed and matched with a base object and can be composed in different combinations (including cases which would otherwise lead to repeated inheritance) into static type definitions.

The paper assumes a knowledge of the papers mentioned above [5, 10] as well as an earlier paper which describes the fundamentals of the Timor approach to single inheritance [4]. The reader is advised that the code re-use technique as described in [4, 5] has been revised to take repeated inheritance into account. The new technique is described in [10], and knowledge of this technique is assumed here.

Section 2 describes how types involving diamond inheritance can be defined in Timor using a conventional OO style, and shows that with the standard Timor re-use technique it is only possible to re-use an implementation of one of the supertypes (cf. Java). Section 3 illustrates that there is also a serious problem at the type level as soon as repeated inheritance is considered. Section 4 then presents an alternative approach, showing how attribute types (which can be compared with adjectives in natural languages) can be defined and implemented. Then section 5 shows how these can be combined, using the analogy of adjectives in noun phrases, to compose types involving diamond inheritance and repeated inheritance (e.g. DoublyEmployedStudent). Section 6 shows that the implementation difficulties encountered in section 2 do not occur when attribute types are used, and section 7 shows that even diamond inheritance types defined in the conventional way can be implemented using implementations of attribute types, with full code re-use. Section 8 discusses the peculiarities of value and reference variables for attribute types and section 9 describes casting of types containing attributes. The paper then discusses related work in section 10 and provides some concluding remarks in section 11.

2 DIAMOND INHERITANCE

We begin by considering a standard example of diamond inheritance, showing how a type `Person` might serve as a supertype from which types such as `Student` and `Employee` can be derived and then combined into an `EmployedStudent`. Type inheritance and code re-use are considered in turn. The approach adopted in this section is based on the Timor equivalent of the standard OO paradigm, as described in [5] (i.e. without using part identifiers, but merging methods from a common ancestor).

Diamond Inheritance at the Type Level

A type `Person` might be defined as follows:

```
type Person {
instance:
 String name, address;
 Date dob;    // the date of birth
 enq int currentAge();
 enq String toString();
maker:
 init(String name);
}
```

This might have subtypes `Student` and `Employee`:

```
type Student {
extends:
 Person;
redefines:
 enq String toString();
instance:
 String uni;
 Date matriculationDate;
 enq int ageAtMatriculation();
maker:
 init(String name, uni; Date matriculationDate);
}
type Employee {
extends:
 Person;
redefines:
  enq String toString();
instance:
  String employer, employeeNumber;
  Date startingDate;
maker:
 init(String name, employer; Date startingDate);
}
```

As would be expected, `Student` and `Employee` inherit all the public methods and abstract variables [6, 10] of `Person`, and redefine the method `toString`.

A new type can be derived from the above types, using diamond inheritance, to model an `EmployedStudent`, as follows:

```
type EmployedStudent {
extends:
 Student;
 Employee;
redefines:
 enq String toString();
maker:
 init(String name, uni, employer;
      Date matriculationDate, startingDate);
}
```

Because the type `Person` is inherited as a common ancestor via multiple derived types (cf. `Collection` in [5, 6]) its inherited instance methods and abstract variables are merged in `EmployedStudent`. The method `toString` is redefined, again as expected. If other `Person` methods were inherited in different forms (as a result of method redefinitions in `Student` and/or `Employee`) a further redefinition of the affected methods would also be necessary in `EmployedStudent`.

At this point it appears that an adequate mechanism for handling diamond inheritance from a common concrete ancestor exists at the type level.

Diamond Inheritance at the Implementation Level

Given an implementation of `Person`, the types `Student` and `Employee` might be implemented along the following lines:

```
impl StudentImpl of Student {
state:
 ^Person p;           // reuses any implementation of Person
 // the Person methods (except toString) are matched from p.
 String uni;           // set and get methods are
 Date matriculationDate;// automatically implemented, cf [6]
instance:
 enq String toString() {
  return (p.toString() //equivalent to super in OO languages
          + ... /* code to print Student details */ );
 }
maker:
 init(String name, uni; Date matriculationDate) {
  p.name = name;
  this.uni = uni;
  this.matriculationDate = matriculationDate;
 }
}
impl EmployeeImpl of Employee {
state:
 ^Person p;
 String employeeNumber; // set and get methods are
 Date startingDate;     // automatically implemented, cf [6]
instance:
 enq String toString() {
  return (p.toString()
```

```
            + ... /* code to print Employee details */ );
  }
 maker:
  init(String name, employer; Date startingDate) {
   p.name = name;
   this.employer = employer;
   this.startingDate = startingDate;
  }
}
```

It might be thought that the combined type `EmployedStudent` can then be implemented as follows:

```
impl EmployedStudentImpl of EmployedStudent {
state:
 ^Student s;
 ^Employee e;
instance:
 enq String toString() {
  /* code to combine the toString methods */
 }
maker:
 init(String name, uni, employer;
     Date matriculationDate, startingDate) {
  s.name = name;
  s.uni = uni;
  s.matriculationDate = matriculationDate;
  e.employer = employer;
  e.startingDate = startingDate;
 }
}
```

This would provide an implementation which syntactically matches the type definition: the methods of `Person` (except `toString`, which is implemented explicitly in the `instance` section) and of `Student` are all matched from `Student s`, and the remaining `Employee` methods are matched from `Employee e`.

But semantically the implementation would not achieve the intended result, because it contains two separate `Person` implementations, associated respectively with `s` and `e`, because each of these is a separate variable with its own complete implementation (which by definition includes an implementation of `Person`). In cases where subtype methods access the supertype implementation (e.g. where `Employee` accesses `Person`), the "wrong" `Person` state would be accessed.

We might attempt to solve this problem by including an explicit variable for the "top" of the diamond, e.g.

```
state:
 ^Person p;
```

Even then an efficient implementation is not easy with the means described in earlier papers, because any attempt to re-use `Student` and `Employee` implementations encounters the problem that these each still include a separate state for its `Person` supertype.

A semantically correct implementation of `EmployedStudent` is always possible in Timor, because each implementation can be a fresh implementation: there is no requirement that re-use variables must be used. But to recode each such case from scratch, or to base a new implementation on the re-use of the code of only one supertype (cf. the Java approach) is hardly satisfactory. Before we present a solution for this (implementation level) problem we consider repeated type inheritance involving a common ancestor.

3 REPEATED INHERITANCE WITH A COMMON ANCESTOR

Suppose the above example is changed so that it involves repeated inheritance from a common ancestor, e.g. a `DoubleStudent` (i.e. a `Person` enrolled at two universities) or a `DoubleEmployee` (i.e. a `Person` with two jobs). This might be approached by using part names (the Timor technique for achieving repeated inheritance [6]). Here is a possible definition:

```
type DoubleStudent {
extends:
 Student s1, s2;
redefines:
 [s1, s2] enq String toString();
}
```

For this definition to be semantically appropriate (i.e. such that it defines a single person with repeated `Student` - but not `Person` - attributes) there would have to be a rule requiring that a common ancestor inherited *in multiple parts* leads to the merging of methods.

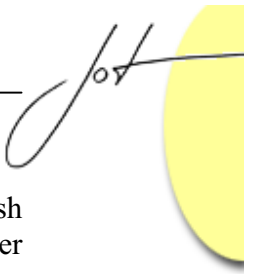
However, such a rule would be undesirable in other cases. For example, suppose we extend the type `CassettePlayer` (defined in [6]) to become a `CassetteRecorder`, as follows:

```
type CassetteRecorder {
extends:
 CassettePlayer;
instance:
 op void startRecording();
 op void stopRecording();
}
```

It would seem appropriate to define a `DoubleCassetteRecorder` (by analogy with the `DoubleCassettePlayer`) as follows:

```
type DoubleCassetteRecorder {
extends:
 CassetteRecorder cr1, cr2;
}
```

But the intended interpretation, that there are two separate parts - each including its own supertype `CassettePlayer` - conflicts with the interpretation appropriate for `DoubleStudent` where the supertype `Person` should only be present once.



It would no doubt be possible to devise mechanisms which could distinguish between such cases, but these would all have at least one disadvantage. Either the definer of the repeated type would have to be aware of the way the individual supertype(s) are defined, or (as with C++ virtual inheritance) a decision would have to be made by the designers of all the second level types, perhaps even before the diamond inheritance case is considered. The deeper the hierarchy involved, the more evident it is that such approaches are unsatisfactory.

The Timor aims of supporting the information hiding principle and of being able to use components without a knowledge of their inner composition led to the decision to adopt the interpretation of the above examples which replicates an *entire part* without consideration of its inner structure or common ancestor(s) (the interpretation relevant for `DoubleCassetteRecorder`), i.e. when multiple subtypes are inherited as parts, methods of a common ancestor are not implicitly merged. According to this rule the type `DoubleStudent` defines a "schizophrenic", with two `Person` elements.

An important additional advantage of this decision is that it creates no implementation difficulties, as the following implementation shows:

```
impl DoubleCassetteRecorderImpl of DoubleCassetteRecorder {
state:
 ^CassetteRecorder cr1;
 ^CassetteRecorder cr2;
}
```

On the other hand all the difficulties associated with implementing `EmployedStudent` would still arise for a rule which favours merging a common ancestor in types such as this.

4 INHERITING ORTHOGONAL ATTRIBUTES

The above discussion leaves at least two questions unanswered:

a) How can repeated inheritance from a common (shared) ancestor (such as a `DoubleStudent`) be appropriately *defined* at the type level?
b) How can diamond inheritance and repeated inheritance involving a common (shared) ancestor be conveniently *implemented* (even where it can be appropriately specified, as in `EmployedStudent`)?

At the heart of these issues is the fact that a (usually concrete) base type serves as a common (shared) ancestor which can be orthogonally extended in a potentially infinite number of ways to specialise objects of the base type. Such orthogonal attributes can then be combined (and might occur repeatedly) in particular objects. `Person` is not an exception in this respect. The same principle would apply to a hierarchy defining ships or vehicles, and many other cases.

Such specialisations are usually incremental extensions which are behaviourally conform with the base type (cf. [11]), i.e. they typically add new state and new methods, without changing the definition of existing methods or state. In fact they are normally not

only behaviourally conform with their supertype but, for a given supertype, they are usually compatible with each other. (In the above example the method `toString` appears to be an exception, but this will be taken into account in the following discussion.)

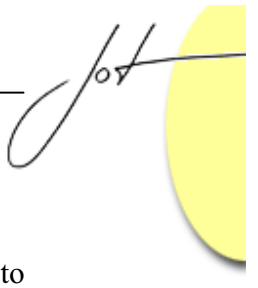
To handle such extensions Timor breaks with the standard OO paradigm, by providing a mechanism for defining behaviourally compatible type units as *add-on* attributes, which can be orthogonally combined with each other in association with a base type. This alternative paradigm is inspired by the use of adjectives in natural language rather than by inheritance concepts. Adjectives (cf. attributes) can be added to nouns (cf. objects) to augment their meaning (e.g. a student is a *studying* person, an employee is an *employed* person, an employed student is a *studying*, *employed* person). In contrast inheritance simply works in terms of nouns (i.e. objects, e.g. a student is a person), with the consequence that in the object oriented paradigm the attributes represented by adjectives simply disappear as separate units. This is unfortunate, since adjectives are especially flexible: the same adjective can often qualify many different nouns, and many different adjectives can (where appropriate concurrently) qualify the same noun. The basic idea behind attribute types in Timor is to introduce a similar level of flexibility into programming. This involves constructs for defining and implementing attribute types (i.e. the adjectives) and further constructs allowing them to be composed into more complex types (i.e. the noun phrases).

In this section it is shown how such attribute types can be defined and implemented in Timor, and in section 5 how they can then be composed into new types.

Attribute Type Definitions

An attribute type definition is characterised by the keyword `for`. The `for` clause nominates a base for the attribute type (known as an *attribute base*[2]). This can be defined in terms of a type, or a view, or the special keyword `any`, and indicates the type(s) of object which can be qualified adjectivally. In the following examples a specific type is nominated as the attribute base.

```
type Studying for Person {
instance:
 String uni;
 Date matriculationDate;
 enq int ageAtMatriculation();
 enq String toString();  // returns a string describing
                         // only the student details
maker:
 init(String uni, Date matriculationDate);
}
type Employed for Person {
instance:
 String employer, employeeNumber;
 Date startingDate;
 enq String toString();  // returns a string describing
                         // only the employee details
maker:
```



```
    init(String employer; Date startingDate);
  }
```

Although attribute types have a similar appearance and serve a similar purpose to subtypes, they are by no means the same. Here are some key differences from the viewpoint of type definition and implementation:

a) An attribute type can have only one attribute base[3].
b) The methods of the attribute base cannot be redefined in a `redefines` clause of an attribute type[4].
c) The attribute type's makers, if any, are responsible only for initialising their own state. They cannot invoke the makers of the attribute base.
d) The instance methods of an attribute type can access the *public* methods of the associated base object via a pseudo variable `base`. This promotes both behavioural conformity with its attribute base and independence of the latter's implementation.
e) The instance methods of an attribute type should confine their activities to the attribute's own state. Thus methods such as `toString` should produce results which can be used to add to (rather than already include) those of the attribute base in a modular way.

In accordance with the philosophy behind attribute types we typically use *adjectival* names in examples.

Implementing Attribute Types

Like other Timor types, attribute types can have multiple implementations. Their implementations differ from implementations of other types only in that the code of their methods can use the pseudo variable `base`, which provides access to the public methods of the attribute base object. Here is an implementation of `Studying`:

```
impl StudyingImpl of Studying {
state:
 String uni;
 Date matriculationDate;
maker:
 init(String uni; Date matriculationDate) {
  this.uni = uni; this.matriculationDate= matriculationDate;
 }
instance:
 enq int ageAtMatriculation() {
  return (Date.yearDifference(matriculationDate, base.dob));
 }
 enq String toString() { // confined only to own state
  return ("Commenced at " + uni
          + " on " + matriculationDate.toString());
 }
}
```

The code of the method `ageAtMatriculation` illustrates how the pseudo variable `base` can be used to gain access to the public methods of the base object (here the get method of `dob` from `Person`, using the abstract variable notation, cf. [6, 10]).

Because of its add-on nature, an implementation of an attribute type does not include state variables describing its attribute base.

Inheriting from Attribute Types

Like other Timor types, attribute types can inherit (in the conventional sense) from other types. Inherited bases may (but need not) be attribute types. If one or more of the inherited bases is an attribute type, then the subtype must have an attribute base which is either the same type as, or is a common subtype of, all the attribute bases of the inherited types.

5 USING ATTRIBUTES IN TYPE DEFINITIONS

In concrete situations adjectives are used to qualify nouns, typically in noun phrases. Similarly, instances of attribute types ("attributes") can be used in association with their base objects to compose new types. In this section we consider how such types are composed.

Composing Types from Attributes

The following is an alternative definition of `EmployedStudent` which uses attribute types.

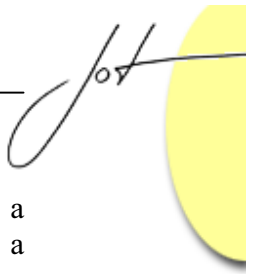
```
type AttributedEmployedStudent {
extends:    // or includes where polymorphism is unwanted
 {Studying; Employed;} Person;
redefines:
 enq String toString();//returns all EmployedStudent details
maker:
 init(String name; String uni; Date matriculationDate;
                        String employer; Date startingDate);
}
```

The `extends` clause defines one or more inherited bases, as usual. Those bases which serve as attribute bases are preceded by a bracketed list of dependent attributes. The methods of the attribute base (here `Person`) and of the individual attributes (here `Studying` and `Employed`) are all "inherited" as separate methods of the new type.

The syntax can be understood in terms of the following EBNF fragment:

```
derivationClause = "extends:" qualifiedList |
                   "includes:" qualifiedList.
qualifiedList = {[qualifyingList]  qualifiedItem ";"}.
qualifyingList = "{"  {qualifiedList}  "}".
```

This syntax is more fully explained in [9], which also describes how it can be used in static definitions that include qualifying types with bracket methods [7, 8]. The basic idea is that a qualified item can be qualified by a qualifying list (here of attributes). Because the qualifying list is recursively defined in terms of a qualified list, qualifying items (here attributes) can themselves be qualified.



Because each of the attributes and the attribute base in this example all have a method `toString`, a name collision occurs, which, if unresolved, would lead to a compile time error. Hence this appears in a `redefines` clause to indicate that it is a common method (with an informal specification indicating what it does)[5].

The instance methods and abstract variables of `AttributedEmployedStudent` are identical to those of the conventional `EmployedStudent`, as follows:

```
instance:
 String name, address;  // from Person
 Date dob;
 enq int currentAge();
 String uni;            // from Studying
 Date matriculationDate;
 enq int ageAtMatriculation();
 String employer, employeeNumber; // from Employed
 Date startingDate;
 enq String toString(); // redefined in the redefines clause
```

Should the designer of the type wish to make the individual `toString` methods of the various parts publicly available as separate methods, this can be achieved by adding part names (which may be defined as optional, cf. [10]), e.g.

```
type EmployedStudentByParts {
extends:
 {Studying [s]; Employed [e];} Person [p];
instance:
 enq String toString();//returns all EmployedStudent details
}
```

In this case there are separate public methods `s.toString`, `e.toString`, `p.toString` and `toString`. Because the part names have been provided in the optional form, non-colliding methods can be invoked by the client either with or without the part name (cf. [6]). This type is not equivalent to `EmployedStudent`.

Repeated Attributes

The modular structure of attribute types allows diamond and repeated inheritance to be simulated in a straightforward manner, e.g.

```
type DoublyEmployedStudent {
extends:
 {Studying; Employed e1, e2;} Person;
redefines:
 [e1, e2] enq String toString();
 //combines all the toString methods into a single method
maker:
 init(String name, uni, e1.employer, e2.employer;
      Date matriculationDate, e1.startingDate, e2.startingDate);
}
```

This is defined by analogy with repeated inheritance of parts. Repeated attributes must have a part name; others (including the attribute base) may, but need not be explicitly named. Part names must be provided whenever a type occurs more than once in the derivation clauses of a type definition. These must be unique within all the derivation

clauses of a type definition and cannot be hierarchical (i.e. their uniqueness must be independent of the dot notation).

Part names are used by clients to invoke methods, as in simple repeated inheritance (except in cases where they are optionally defined using square brackets [10]). Where a part name is used, the names of the object's members are compounded from the part name and the normal method name, using the dot notation, e.g.

```
DoublyEmployedStudent* des =
                        new DoublyEmployedStudent.init(...);
des.e2.matriculationDate = Date.init(1,10,2000);
```

In `DoublyEmployedStudent` the `toString` methods of all the constituent types are merged into a single method. As in the case of repeated parts inheritance, any named parts must be explicitly named in the `redefines` clause if their methods are to be merged. The effect of omitting the part names from the `redefines` clause would be that the methods `toString` from `Studying` and `Person` would be merged, but the methods `e1.toString` and `e2.toString` would remain separate methods for objects of type `DoublyEmployedStudent`.

The parameters of makers may (but need not) use the dot notation to name parameters, where this corresponds to a name as seen by the client. This facility allows simple makers with parameters to be implemented automatically.

Multiple Attribute Bases

The `extends` clause is a normal extends clause. Consequently an attribute base is simply an inherited type (or part), and types can be defined to extend or include multiple attribute bases. Here is an example defining a "schizophrenic", whose first personality thinks he is a student while the second thinks he is doubly employed:

```
type SchizoDoublyEmployedStudent {
extends:
 {Studying s;} Person p1;
 {Employed e1, e2;} Person p2;
redefines:
 [s] enq String p1.toString();
            // returns all details of the first personality
 [e1, e2] enq String p2.toString();
            // returns all details of the second personality
maker:
 init(String p1.name, s.uni, p2.name,
             e1.employer, e2.employer;
      Date s.matriculationDate,
           e1.startingDate, e2.startingDate);
}
```

In this example there is no method `toString`, but there are methods known to the client as `p1.toString` and `p2.toString`.

Bases for Attribute Types

A `for` clause can nominate a type as its attribute base, as in the above examples, or it can nominate a view (cf. [6]), in which case an instance of any type which implicitly or explicitly contains this view can serve as a base type. In both cases the compiler checks that instances of the attribute type are only used in conjunction with bases which contain the view or type named in the `for` clause.

Alternatively an attribute type can be defined to have the special base `any`, which indicates that it can have any type as an attribute base. In this case its implementations may *not* use the pseudo variable `base`. Here is an example:

```
type Loanable for any {
instance:
 op void putOnLoan (Person* toWhom, Date loanDate);
 op void returnFromLoan(Date returnDate);
 boolean currentlyLoaned;  // an abstract variable
 Date dueDate;             // an abstract variable
 enq int daysOverdue();
 enq Person* borrower();
 enq Person* previousBorrower();
 enq Date dateLastReturned();
 ...
}
```

The `for any` clause indicates that instances of this type should only be used in conjunction with an attribute base, even though implementations may not use the pseudo variable `base`.

Attribute Types as Attribute Bases

An attribute type can qualify another attribute type. This is equivalent to qualifying an adjective with an adverb. Thus an attribute `PartTime` might be defined as follows:

```
type PartTime for any { // for Studying would also be OK
                        // and base could then be used
 int fullTimeHours;
 float fractionPartTime;
}
```

This could be used in a type:

```
type SchizoPartTimeDoublyEmployedStudent {
extends:
 {{PartTime pt1;} Studying;} Person p1;
 {{PartTime pt2;} Employed e1; Employed e2;} Person p2;
maker:
 init(String p1.name, p1.uni,
             p2.name, e1.employer, e2.employer;
      Date p1.matriculationDate,
           e1.startingDate, e2.startingDate);
}
```

Here the first personality is a part-time student, while the second has a part-time employment `e1` and an employment `e2` not defined as part-time. The items `PartTime`,

`Employed` and `Person` must have part names, because of the type repetition, but `Studying` need not. In the latter case the member names "belong to" the next named higher part in the hierarchy. In this example there is, for instance, an abstract variable in `Studying` named `p1.uni`. The client refers to the methods in the `PartTime` items as `pt1.fractionPartTime` and `pt2.fractionPartTime`, e.g.

```
SchizoPartTimeDoublyEmployedStudent* schizo =
        new SchizoPartTimeDoublyEmployedStudent.init(...);
float hoursWorked = schizo.pt2.fractionPartTime;
```

6 IMPLEMENTATIONS USING ATTRIBUTE TYPES

Automatic Implementations

Assuming that implementations for all the types used in derivation clauses (see section 5) already exist, it would be tedious to require the programmer to provide explicit implementations of types into which they are composed, especially if no methods are being overridden and no new methods are being added. For such cases, if there are no explicit makers (or if the makers conform to certain requirements), the compiler can provide an automatic implementation of the type in question. It transforms the type definition using the following basic rules:

a) Change each `extends` or `includes` clause into a `state` clause.
b) For each type which does not already have a part name in the type definition, add a part name (the same as the type name, but beginning with a small letter) to form a variable declaration.
c) For each type which already has a part name in the type definition, use that part name to form a variable declaration.
d) Prefix the hat symbol to each type name of a variable declaration which provides public methods.
e) Add a parameterless maker or makers which conform to the requirements for producing automatic makers.

The following is an automatic implementation of the type `SchizoPartTimeDoubly-EmployedStudent` (cf. the last subsection of section 5):

```
impl SchizoPartTimeDoublyEmployedStudentImpl of
                    SchizoPartTimeDoublyEmployedStudent {
state:
 {{^PartTime pt1;} ^Studying studying;} ^Person p1;
 {{^PartTime pt2;} ^Employed e1; ^Employed e2;} ^Person p2;
maker:
 init(String p1.name, p1.uni,
             p2.name, e1.employer, e2.employer;
      Date p1.matriculationDate,
           e1.startingDate, e2.startingDate) {
 p1.init(p1.name);
  studying.init(p1.uni, p1.matriculationDate);
   pt1.init();
 p2.init(p2.name);
```

```
    e1.init(e1.employer, e1.startingDate);
     pt2.init();
    e2.init(e2.employer, e2.startingDate);
   }
  }
```

This automatic implementation sets out a pattern which can be used in explicit implementations. The key points which it illustrates are as follows:

a) Qualified lists can be used in the state section of an implementation to express the relationships between attributes and their attribute bases.
b) Each variable has a unique name.
c) Makers of attributes are invoked from within the maker for the composed type to instantiate the necessary attributes.
d) Makers conforming to simple rules with respect to their formal parameter names can be implemented automatically.

Explicit Implementations

We now show how an explicit implementation might handle a redefined method. To illustrate this we implement the type `DoublyEmployedStudent` (defined in section 5 using attribute types):

```
impl DoublyEmployedStudentImpl of DoublyEmployedStudent {
state:
 {^Studying s;       //re-use any Studying implementation
  ^Employed e1, e2;} //re-use any Employed implementation(s)
 ^Person p;          //re-use of any Person implementation
/* maker:
 init(String name, uni, e1.employer, e2.employer;
  Date matriculationDate, e1.startingDate, e2.startingDate);
 This maker is automatically implemented. It does not need
 to be explicitly coded in the implementation.
*/
instance:
 enq String toString() {
  return (p.toString() + s.toString()
          + e1.toString() + e2.toString());
 }
}
```

In accordance with the Timor strategy that any type can be implemented from scratch, there is no necessity that a type defined in terms of attributes must be implemented using attribute types. However, if implementations of attribute types are used in an implementation of some other type the programmer must define these using qualified lists, in order to clarify the relationships between attribute implementations and their attribute bases.

7 IMPLEMENTING CONVENTIONAL DIAMOND INHERITANCE VIA ATTRIBUTES

In the implementation discussion in section 2, which was based on a conventional approach to inheritance, we did not find a straightforward solution for implementing diamond inheritance with more than one re-use variable. However, the previous sections have illustrated that no substantial problems arise when attributes are used to achieve the equivalent of diamond inheritance.

Because of the relative independence of type definitions and implementations, attribute implementations can be used not only to implement types defined using attributes but also to implement types defined in the conventional diamond inheritance style, such as `EmployedStudent`. The following type definition remains unchanged from section 2. It does not include attribute types.

```
type EmployedStudent {
extends:
 Student;
 Employee;
redefines:
 enq String toString();
init(String name, uni, employer;
     Date matriculationDate, startingDate);
}
```

Here is an implementation, using attribute implementations defined previously:

```
impl EmployedStudentImpl of EmployedStudent {
state:
 {^Studying s;  // any implementation of the attribute types
  ^Employed e;}  // Studying and Employed can be used
 ^Person p;
maker:
 init(String name, uni, employer;
      Date matriculationDate, startingDate) {
  p.init(name);
   s.init(uni, matriculationDate);
   e.init(employer, startingDate);
 }
instance:
 enq String toString() {
  return (p.toString() + s.toString() + e.toString();
 }
}
```

The `Person` methods are matched from the re-use variable `^Person p`, while the *additional* `Student` and `Employee` methods are matched from (any implementation of) the re-use variables `^Studying s` and `^Employed e`. Although these are different types from those used in the type definition (i.e. `Student` and `Employee`) the instance methods match and so are selected.

The fundamental difference from the attempted diamond inheritance implementation in section 2 is that in contrast with implementations of `Student` and `Employee` implementations of `Studying` and `Employed` do not include state for `Person`. Consequently the problems encountered earlier do not arise.

Although the individual `toString` methods have been merged into a single redefined method, they still exist in the re-use variables representing the attributes and their base, and can still be invoked in implementations which re-use them, as is illustrated in the implementation of the redefined `toString`.

8 ATTRIBUTE VALUES AND REFERENCES

The peculiarities of attribute types lead to some special rules with regard to attribute values and attribute references.

Attribute Values

An attribute (instance) relies logically - and if it uses the pseudo variable `base` also physically - on the existence of its attribute base, which implies that an attribute cannot simply be instantiated as a free-standing object or value. Consequently it is inappropriate to allow attributes to be declared as free-standing value variables in implementations (or as abstract values in type definitions). Hence value declarations of attribute types are permitted only as items in qualifying lists. A simple declaration such as

```
Employed e;
```

is treated as a compile time error.

Attribute References

Like a view, an attribute type provides instance methods which are a subset of the instance methods of objects in which it is embedded. Consequently it can be useful to allow reference variables to refer to an existing attribute. Hence it is permitted to declare an attribute reference, e.g.

```
Employed* emp;
```

Such reference variables, like reference variables for supertypes and views, refer to the entire object in which the particular attribute is embedded, and can therefore be the subject of cast statements (see section 9).

Because of the need to guarantee the existence of a `base`, such a reference cannot be used to instantiate an actual attribute as such, i.e. the compiler would treat a statement such as:

```
Employed* emp = new Employed.init();
```

as an error, but it would allow statements such as

```
Employed* emp1 = schizo.e1; // for schizo cf. Section 5 end
Employed* emp2 = new
        (SchizoPartTimeDoublyEmployedStudent.init()).e2;
```

Unlike a view, an attribute type cannot be defined retrospectively, so that a statement such as the latter would not be valid merely on the basis of matching methods. Thus an assignment statement

```
Employed* emp3 = new EmployedStudent.init();
```

is erroneous, because the type `EmployedStudent` is defined in terms of `Employee`, not `Employed`, whereas

```
Employed* emp4 = new AttributedEmployedStudent.init();
```

is valid.

9 THE CAST STATEMENT

An object is considered to be behaviourally conform with any attribute bases which it `extends` (but not `includes`) and it can therefore be used polymorphically, where appropriate using part names to identify which attribute base is intended, e.g.

```
Person* p = schizo.p2; // for schizo cf. section 5 (end)
p.name = "Joe Confused";
```

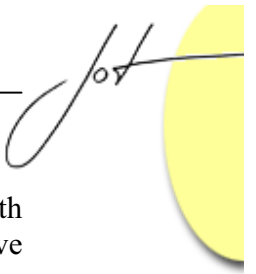
As in the cases of normal inheritance and parts inheritance, such an assignment logically assigns the entire object (not merely the part) to the reference. The Timor conditional downcast statement can then be used in the usual way to gain access to the entire subtype, e.g.

```
cast (p) as {
 (EmployedStudent es) {/* statements using es */}
 (SchizoPartTimeDoublyEmployedStudent sptdes)
                     {/* statements using sptdes */}
  ...
 else {/* optional statements if there is no match */}
}
```

The cast statement can also be used to access *attributes* contained in an object, and where appropriate the square bracket notation can be used to gain access to multiple attributes of the same type (cf. repeated parts [10]), e.g.

```
cast (p) as allof { // allof indicates that all matching
                    // individual clauses are selected
 [Employed e] // square brackets indicate repetition
              // for each matching attribute
   {if (e.employer = "University of Ulm") ...;
    cast e as allof {
     [PartTime pt] {...}
     [FullTime ft] {...}
   }
 [Studying s] {...}
 ...
}
```

As this example illustrates, nested cast statements can be used to access (depth first) all the attributes in an object in succession. Where cast statements are nested in this way all



references in the current hierarchy are accessible. Thus the statements associated with `PartTime pt` can use the current values of `pt`, `e` and `p` as references (unless they have been hidden by other in scope references which use the same identifiers).

Finally a conditional cast can be applied to an attribute reference in order to gain access to an underlying object or, as in this example, other attributes in the object:

```
cast (emp) as { // is this employed person also studying?
 [Studying s] {...}
}
```

10 RELATED WORK

In 1997 members of our group published a paper entitled "Attribute Types and Bracket Implementations" [3] which presented in outline ideas developed for the experimental language L1. The paper outlined in nascent form the basic concepts both of Timor attribute types, presented in the present paper, and of Timor qualifying types (cf. [7, 8]). Although the ideas for both have since been considerably refined and improved for Timor, the idea that a programming language should support not only types based on nouns but also further types based on adjectives, together with a technique allowing new types (corresponding to noun phrases) to be composed from these, was already emphasized in that paper.

Others have also pointed out that the object oriented paradigm could be enriched by taking adjectives more seriously (e.g. [1, 2]) but have not described a technique for doing this corresponding to attribute types.

The need for adjectival types in the object oriented paradigm has become visible partly through Java interfaces and the tendency to name some of these adjectivally, e.g. Runnable, Serializable. However, in contrast with Timor's attribute types, Java interfaces do not provide a solution with full code re-use for diamond inheritance from a common concrete ancestor nor a solution to repeated inheritance involving a common concrete ancestor.

11 CONCLUSION

The paper has presented some aspects of an alternative programming paradigm to inheritance, based on the idea of adjectives in natural language. In doing so we have shown that the technique can easily master issues such as diamond inheritance and repeated inheritance from a common concrete ancestor. This technique can be used in Timor to complement both the conventional object oriented programming paradigm (which can be effectively used for subtyping involving single inheritance and cases of multiple inheritance from a common abstract ancestor) and the parts inheritance technique (which is especially suitable for repeated inheritance and for multiple inheritance from distinct concrete types).

Attribute types have two significant characteristics: they are very modular units and they cannot redefine the methods of their attribute base type. It is these features which allow them to be easily mixed and matched to compose new types, as we have described in the paper. But these characteristics endow them with a further advantage: such mixing and matching need not be limited to static type definitions. In a future paper we will show how individual attributes can be dynamically added at run-time to appropriate attribute base objects, thus allowing, for example, a `Person` object to change its specialisations dynamically over time. In this sense attribute types should make Timor especially attractive for data base applications in which objects need to change over time to reflect changes in the world which is being modelled.

Finally we point out that the other "adjectival" form of type in Timor, qualifying types, can also be statically defined, and in fact the same rules as we saw in section 5 for composing attribute types into new types are used for qualifying types. Because the rules required for qualifying types are somewhat more complex (in view of the existence of bracket methods) we have deferred a full discussion of that syntax until a later paper in which statically defined qualifying types are presented.

ACKNOWLEDGEMENTS

Special thanks are due to Dr. Axel Schmolitzky for his invaluable contributions to discussions of Timor and to the ideas which have been taken over from earlier projects. Without his ideas and comments Timor would not have been possible.

REFERENCES

[1] M. C. Feathers, "Factoring Class Capabilities with Adjectives," *Journal of Object Oriented Programming*, vol. 12, no. 1, pp. 28-34, 1999.

[2] I. Forman and S. Danforth, *Putting Metaclasses to Work*. Reading, MA. Addison-Wesley, 1998.

[3] J. L. Keedy, M. Evered, A. Schmolitzky, and G. Menger, "Attribute Types and Bracket Implementations," 25th International Conference on Technology of Object-Oriented Languages and Systems, Melbourne, 1997, pp. 325-338.

[4] J. L. Keedy, G. Menger, and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 35-43.

[5] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," *Journal of Object Technology*, vol. 1, no. 1, May 2002, pp. 81-106. http://www.jot.fm/issues/issue_2002_05/article2.

[6] J. L. Keedy, G. Menger, and C. Heinlein, "Taking Information Hiding Seriously in an Object Oriented Context," Net.ObjectDays, Erfurt, Germany, 2003, pp. 51-65.

[7] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344.

[8] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 101-121, http://www.jot.fm/issues/issue_2004_01/article1

[9] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Statically Qualified Types in Timor," *(accepted for publication in JOT, September-October 2005)*

[10] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting Multiple and Repeated Parts in Timor," *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 99-120. http://www.jot.fm/issues/issue_2004_11/article1

[11] B. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811-1841, 1994.

About the authors

J. Leslie Keedy is Professor and Head, Department of Computer Structures, University of Ulm, Germany, where he leads the Timor language design and the Speedos operating system design groups. His email address is keedy@informatik.uni-ulm.de. His biography can be visited at http://www.informatik.uni-ulm.de/rs/mitarbeiter/jlk/

Mark Evered is a Senior Lecturer in the School of Mathematics, Statistics and Computer Science at the University of New England in Armidale, Australia. He completed his PhD at the Technical University of Darmstadt in Germany. His research interests include Object-based Systems, Security, Persistence and Programming Language Design and Implementation. His email address is: markev@mcs.une.edu.au.

Christian Heinlein received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently, he works as a scientific assistant in the Department of Computer Structures at the University of Ulm. His research interests include programming language design in general, especially genericity, extensibility and non-standard type systems. His email address is heinlein@informatik.uni-ulm.de.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently she works as a scientific assistant in the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering. Her email address is menger@informatik.uni-ulm.de.

[1] see http://www.timor-programming.org

[2] As will become clear, this differs from the bases from which a type can inherit (in Timor using the keywords `extends` and/or `includes`) in the conventional OO sense. We refer to such bases as *inherited bases*.

[3] It can however be defined to extend and/or include other types in the usual way.

[4] It can however have a `redefines` section in which methods of its *inherited* bases can be redefined.

[5] Although an attribute type may not redefine the methods of its base type, the designer of a type which composes an attribute type with its attribute base can make such redefinitions.