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

Inheriting Multiple and Repeated Parts in Timor

J. Leslie Keedy, Christian Heinlein and Gisela Menger, University of Ulm, Germany

Abstract

The paper describes one aspect of multiple inheritance in the Timor programming language, viz. how "parts" such as a type Radio and a type Cassette Player can be inherited, where appropriate repeatedly, in subtypes such as a Radio Double Cassette Player. Because such types can also be defined via aggregation the paper begins by comparing inheritance with aggregation. It then shows how such cases can be handled first at the type level and then at the implementation level in Timor.

1 INTRODUCTION

Multiple inheritance can be used to model a number of different situations. Some cases involve a common abstract ancestor (e.g. variants of a type `Collection`, such as `Set`, `Bag` or `List`) or a common concrete ancestor (e.g. variants of a type `Person` with multiple roles, such as a type `EmployedStudent`), while others involve inheritance of separate "parts" (e.g. a type `RadioCassettePlayer` inheriting from a type `Radio` and a type `CassettePlayer`). Sometimes repeated inheritance can be appropriate (e.g. a `DoubleDegreeStudent` or a `DoubleCassettePlayer`), and repeated inheritance can be combined with other forms of multiple inheritance (e.g. `DoublyEmployedStudent`, `RadioDoubleCassettePlayer`). The situation can be further complicated by the fact that the behaviour of some methods of one or more base types might need to be redefined, and also one or more of the types might have different implementations. Given this multiplicity of aims it is perhaps not surprising that not all OO languages provide mechanisms which adequately cover all the requirements.

In this paper we describe the mechanisms which handle "parts" inheritance provided in the language Timor[1]. Timor 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 existing components which can be separately designed and developed without knowledge of each other.

Cite this article as follows: Leslie Keedy, Christian Heinlein, Gisela Menger: “Inheriting Multiple and Repeated Parts in Timor”, in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 99-120. http://www.jot.fm/issues/issue_2004_11/article1

As a first step in this direction Timor distinguishes type definitions (introduced by the keyword `type`) from their implementations (keyword `impl`), cf. [3, 4]. An implementation, unlike a class in the conventional OO paradigm, is not a type. A type can have multiple implementations and these can re-use other implementations. There is an implicit mechanism for using default implementations of types.

The separation of types and their implementations leads in turn to a separation of subtyping and the re-use of existing implementations. Subclassing (code inheritance) is abandoned in favour of a more general technique which allows existing implementations of the same or of a different type to be re-used in an implementation of a type. With the help of this technique Timor can reverse the subclassing relationship, such that a base type (e.g. `Queue`) can be implemented by re-using *any* implementation of its derived type (e.g. `DoubleEndedQueue`), cf. [3]. One advantage of this approach is that the information hiding principle [11] is naturally upheld. Timor's approach with respect to multiple inheritance involving a common abstract ancestor is described in [4]. Since [4] and [3] were written some ideas have been refined and syntactic improvements have been made to Timor. However, the essence of the papers remains valid. The new syntax is explained and used in this paper.

Because aggregation can be seen as an alternative to parts inheritance, section 2 describes aggregation in Timor, while section 3 discusses the relationship between aggregation and inheritance. Section 4 then turns to inheritance as such, briefly describing the structure of types and their implementations. Because repeated inheritance is regarded as fundamental, section 5 describes how it is handled in Timor at the type level; the section concludes by describing how colliding methods from unrelated ancestors can be handled. In section 6 implementation inheritance is presented, using the revised Timor syntax. Section 7 discusses related work and section 8 concludes the paper. Diamond inheritance is discussed in a companion paper [6].

2 AGGREGATION IN TIMOR TYPES

In Timor the types `Radio` and `CassettePlayer` might be defined as follows:

```
type Radio {
instance: // introduces public instance methods of a type
 op void switchOn();
  // an op (operation) is an instance method which can
  // modify an instance's state, i.e. it is a "writer"
 op void switchOff();
 enq boolean isSwitchedOn();
  // an enq (enquiry) is an instance method which can read
  // but not modify an instance's state, i.e. a "reader"
 op void setStation(float waveLength);
 op void volumeControl(int volume);
}
type CassettePlayer {
instance:
```

```
 op void switchOn();
 op void switchOff();
 enq boolean isSwitchedOn();
 enq int tapePosition();
 op void fastForward();
 op void fastRewind();
 op void play();
 op void stop();
}
```

To create a type whose instances combine the features of both types aggregation can be used, e.g.

```
type AggregatedRadioDoubleCassettePlayer {
instance:
 Radio r;                  // an abstract variable
 CassettePlayer cp1, cp2; // more abstract variables
}
```

Because Timor rigorously adheres to the information hiding principle, this type definition uses *abstract* variables which can in the present example be considered as a shorthand notation for the following[2]:

```
type AggregatedRadioDoubleCassettePlayer {
instance:
 op Radio r(Radio r);                        // a "set" method
 enq Radio r();                              // a "get" method
 op CassettePlayer cp1(CassettePlayer cp1);// a "set" method
 enq CassettePlayer cp1();                   // a "get" method
 op CassettePlayer cp2(CassettePlayer cp2);// a "set" method
 enq CassettePlayer cp2();                   // a "get" method
}
```

A programmer of the new type does not have to provide an explicit implementation of the methods corresponding to the aggregated items. If the programmer simply includes a corresponding concrete variable in his code, i.e.

```
state: // introduces state (i.e. private instance) variables
 Radio r;  // a concrete variable
```

the compiler automatically adds a standard implementation, along the following lines:

```
instance:
 op Radio r(Radio r) {return this.r = r;}
 enq Radio r()       {return this.r;}
```

If a type definition consists only of aggregated variables (i.e. it corresponds to a *struct* or *record*), the compiler produces a standard implementation for the entire type, e.g.:

```
impl AggregatedRadioDoubleCassettePlayerImpl
                 of AggregatedRadioDoubleCassettePlayer {
state:
 Radio r;
 CassettePlayer cp1, cp2;
instance:  // public instance methods
 op Radio r(Radio r) {return this.r = r;}
```

```
 enq Radio r()       {return this.r;}
 op CassettePlayer cp1(CassettePlayer cp1)
  {return this.cp1 = cp1;}
 enq CassettePlayer cp1() {return this.cp1;}
 op CassettePlayer cp2(CassettePlayer cp2)
  {return this.cp2 = cp2;}
 enq CassettePlayer cp2() {return this.cp2;}
}
```

However, the implementation programmer can, if he chooses, take advantage of the information hiding principle by implementing the set and get methods in some other way, and where appropriate can also use different data structures [5].

Despite this mechanism a client programmer sees no noticeable difference from the normal OO paradigm. He can for example invoke the methods of the aggregated radio part of an instance of this type and he can assign a new value to it, e.g.

```
AggregatedRadioDoubleCassettePlayer* rdcp =
            new AggregatedRadioDoubleCassettePlayer.init();
 // objects are instantiated using "new", which returns
 // a reference value to a new instance
rdcp.r.volumeControl(3);
Radio aRadio = Radio.init(); // variables declared by value
 // are instantiated without "new"
rdcp.r = aRadio;
```

Such statements are interpreted by the compiler as method invocations, e.g.

```
rdcp.r = aRadio;            =>  rdcp.r(aRadio);
```

The compiler can of course make optimisations, so that in the normal case there need be no loss of efficiency compared with aggregation in other languages.

It would violate the information hiding principle to allow references to internal variables of an object to exist outside the object (especially as equivalent concrete variables *might not exist* in a non-standard implementation of the type), so a statement in the C++ style such as

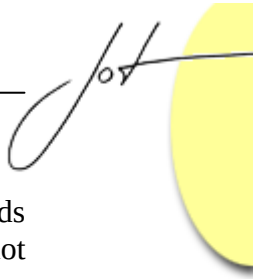
```
Radio* aRadioRef = &rdcp.r;
```

is not supported.

3 AGGREGATION AND INHERITANCE

If individual types can be incorporated into a new type by aggregation, why should one wish to use inheritance as an alternative in such a case? What inheritance allows, but aggregation does not, are the following possibilities:

a) If a subtype object (e.g. a radio double cassette player) has been assigned to a supertype reference (e.g. a radio), a client programmer can use a conditional downcast statement on that reference to gain access to the entire object.



b) With inheritance the implementor of a derived type can override the methods associated with the part, but in the normal object oriented paradigm he cannot override the methods of an aggregated variable.
c) With inheritance the methods of a part become methods of the derived type as a whole. When aggregation is used the aggregated variable also has to be named in order to invoke a method.

On the other hand, if a part is inherited

d) a new value cannot be assigned to it as if it were an aggregated variable and its value cannot be copied to another variable.

In other words, the two forms of modelling are by no means equivalent. How significant are the differences?

Polymorphic Use of Inherited Parts

The advantages of inclusion polymorphism [2] are well known and need not be repeated here. If the present example were defined in terms of inheritance, it would be possible to treat an object of type `RadioDoubleCassettePlayer` as if it were simply of type `Radio` or of type `CassettePlayer` by assigning a reference for the combined object to a reference variable of the appropriate type. Furthermore it would be possible, using an appropriate downcast statement, to examine such a reference with the aim of accessing the device as a whole.

Overriding the Methods

In the example, the radio and the cassette player parts each have methods for switching the device on and off. With aggregation, the two parts remain from this point of view quite distinct components of the combined device, i.e. there are two separate devices under one cover, which have separate switches. With a good inheritance scheme the designer of the new type gains the freedom (without the compulsion) to merge such methods, so that for example a single switch turns both devices on/off. With the usual understanding of aggregation, such freedom does not exist (but see [10]).

Naming Parts

For the kind of example under discussion the fact that modelling by aggregation requires the parts to be individually named when a client invokes methods is not a serious hindrance. In fact the explicit naming of inherited parts can be an advantage, because it solves the naming problems which arise with repeated inheritance and with clashing method names.

Assigning Values to and from a Part

A new value can normally be assigned to an abstract variable in Timor by invoking the associated "set" method. However, this can be prevented by declaring the abstract variable as `final`, which has the effect that there is a "get" method but no "set" method.

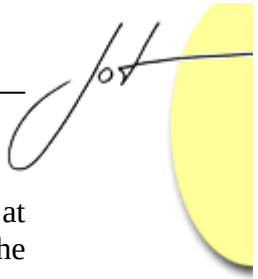
There is no way of preventing a "get" method from being invoked. This is conceptually equivalent to the way public fields are managed in Java, for example.

With inheritance the situation is different, in that inheritance does not normally provide a way of accessing the state of a supertype as if it were a variable with a separate value, and with good reason. To treat it in this way would imply that a supertype has a separable state within its subtype. But this notion frequently does not correspond to the kinds of situation in which inheritance is used. For example if a concrete subtype `List` is derived from an abstract supertype `Collection`, there is no notion that the `List` contains a `Collection` as a separate part. In such a case the inheritance of methods, not of state, is in the foreground. But even when the idea of inheriting state seems relevant (as in the inheritance of two `CassettePlayer` parts in a `RadioDoubleCassettePlayer`), there is no guarantee that these parts can be considered as totally separate. Normally the idea of inheritance is that the supertype becomes an *integral* part of the subtype, so that the copying of "values" of apparently separate parts is a dangerous notion. This is underlined by the fact that methods can be merged and redefined as part of the derived type's definition. What is the separate state of a `CassettePlayer` part of a `DoubleCassettePlayer`, for example, if the switch methods of the individual `CassettePlayer` parts are merged? Redefining methods implies changing their behaviour and this can imply the merging of part states.

4 BASE TYPES AND DERIVED TYPES

Having established that inheritance and aggregation differ, we now turn to the issue of parts inheritance as such. Timor recognises two forms of type derivation, distinguished by the keywords `extends` (which is intended to signal the programmer's intention to define a behaviourally conform subtype [8]) and `includes` (which signals *interface inheritance* without subtyping). A type definition can include both forms of derivation, in which case the resulting type can be used polymorphically as a subtype of those types which it extends but not with respect to those which it includes. Because the structure of `extends` and `includes` sections are identical, `includes` sections are not discussed explicitly. The general structure of a derived type is:

```
[abstract] type typename {
extends:
  list of supertypes
includes:
  list of inherited interfaces
redefines:
  list of redefined public instance methods
instance:
  list of public instance methods (including abstract
  variables) added in this definition
maker:
  list of constructors (known in Timor as makers)
}
```



An abstract type cannot be instantiated. It can have one or more implementations and at the type level it can *predefine* makers for its subtypes; these are distinguished by the keyword `ThisType` [4]. An implementation of an abstract type can also implement makers for re-use, but these cannot be directly invoked by clients.

The `redefines` and `instance` sections include only public methods. There is a fundamental difference between method redefinition (which can appear in `redefines` sections of type definitions and refers to the redefinition of *behaviour*), and code overriding (which is an implementation technique).

An implementation of a type has the following basic structure:

```
impl implname of typename {
state:
 variables which define the state of an instance of the type
instance:
  list of instance methods implemented in this definition
maker:
 list of makers
}
```

The `instance` section includes both public and private methods. The special `overrides` section described in earlier papers is no longer needed. As a result of a further simplification there is no separate `reuses` section. We shall see later how the idea of re-use can be expressed in the `state` section.

The order of the sections in a type definition and in an implementation can vary and any section can appear more than once.

5 REPEATED TYPE INHERITANCE

In OO languages repeated inheritance is often neglected or totally ignored. This is sometimes regarded as unimportant, because in many cases aggregation can be used as an alternative. However, that is not a satisfactory solution, because, as discussed in section 3, aggregation and inheritance are really quite different in their effects. Hence repeated inheritance is treated as fundamental in the Timor design. In this respect it has an advantage over other object oriented languages: by separating types and implementations, these aspects of repeated inheritance can be managed separately. In this section we consider only type inheritance aspects of repeated parts inheritance.

Defining Repeated Parts

The naming issue which arises with repeated inheritance has influenced the structure of the clauses which appear in an `extends` section. Each clause involving repeated inheritance has the same basic form as the declaration of abstract variables in the `instance` section of a type definition (and the declaration of concrete variables in a `state` section of an implementation). Thus a type `DoubleCassettePlayer` defined using multiple inheritance has a similar appearance to aggregation, i.e.

```
type DoubleCassettePlayer {
extends:
 CassettePlayer cp1, cp2; // supertypes with part names
}
```

Syntactically it differs from aggregation only in that the parts appear in an `extends` section rather than an `instance` section. Semantically it differs from aggregation in the ways described in section 3. Because repeated inheritance is involved, the identifiers `cp1` and `cp2` are essential for naming purposes. But they also convey the notion that state and the associated methods are replicated - without implying that the part states are entirely separate (in contrast with aggregation). The degree of integration/separation is defined, from the client's viewpoint, not in the `extends` section, but in the `redefines` section, where, as we shall see shortly, methods can be merged.

When referring to one of the `CassettePlayer` parts, the client programmer uses the dot notation, as he would for an aggregated abstract variable, e.g.

```
DoubleCassettePlayer* dcp =
                      new DoubleCassettePlayer.init();
dcp.cp1.switchOn();
```

Syntactically, the use of the dot notation here deliberately resembles that in the standard object oriented paradigm, but semantically `cp1.switchOn` is not an object with its method, but is rather the *non-decomposable* name of a method of `dcp` as such. Thus (in accordance with the information hiding principle [11]) the use of the dot notation does *not* imply that there is actually a variable `cp1` in all implementations of the type (although there may be in some implementations). Rigorously applying the information hiding principle implies that the implementor of a type can implement the type in any way that he sees fit, provided that his implementation conforms with the specification. Formally the instance methods of `DoubleCassettePlayer` which need to be implemented are as follows:

```
instance:
 op void cp1.switchOn();
 op void cp1.switchOff();
 enq boolean cp1.isSwitchedOn();
 enq int cp1.tapePosition();
 op void cp1.fastForward();
 op void cp1.fastRewind();
 op void cp1.play();
 op void cp1.stop();
 op void cp2.switchOn();
 op void cp2.switchOff();
 enq boolean cp2.isSwitchedOn();
 enq int cp2.tapePosition();
 ...
}
```

and there is a default parameterless maker `init()`.

Merging Individual Methods

As defined above, an instance of the type `DoubleCassettePlayer` can almost be regarded as two separate units which have been put into a single box. These parts can, for example, be switched on and off separately. However, a more integrated unit might be modelled such that its instances are activated by a single switching mechanism. At the type level this involves redefining the switching methods in such a way that the client sees only one such mechanism. But then he might need to have some further methods for determining the mode in which the device should be active. To do this he could define a type as follows:

```
enum DeviceMode {playCassette1, playCassette2}
type IntegratedDoubleCassettePlayer {
extends:
 CassettePlayer cp1, cp2;
redefines:
 [cp1, cp2] op void switchOn();
 [cp1, cp2] op void switchOff();
 [cp1, cp2] enq boolean isSwitchedOn();
instance:
 op void setDeviceMode(DeviceMode mode);
 ...
}
```

The switching methods are here redefined in such a way that there is only one switching mechanism for the combined device. The syntax `[cp1, cp2]` lists the parts affected by the redefinition of a method. The corresponding method then no longer exists as separate methods for the listed parts, but is subsumed into a single method which does not have a "part" name, i.e. the device as a whole is switched on using a statement such as:

```
dcp.switchOn();
```

Hence the instance methods of the type `IntegratedDoubleCassettePlayer` are as follows:

```
instance:
 op void switchOn();
 op void switchOff();
 enq boolean isSwitchedOn();
 enq int cp1.tapePosition();
 op void cp1.fastForward();
 op void cp1.fastRewind();
 op void cp1.play();
 op void cp1.stop();
 enq int cp2.tapePosition();
 ...
}
```

and there is a default parameterless maker `init()`.

Merging all the Methods of a View

Although the above syntax is relatively short, it does involve separately listing each method to be merged. Timor provides an interface definition mechanism, known as a view, which allows related sets of methods to be grouped together, e.g.

```
view Switchable {
 op void switchOn();
 op void switchOff();
 enq boolean isSwitchedOn();
}
```

Views are groupings of instance methods which can usefully be incorporated into different types. They may be defined retrospectively, such that if all the methods defined in a view occur as instance methods of a pre-existing type a match occurs[3]. Thus the view `Switchable` matches the types `Radio` and `CassettePlayer`. In order to shorten redefinitions of methods and make them more easily intelligible, a `redefines` clause can rename all the methods of a view together. Thus the type `IntegratedDoubleCassettePlayer` could have been defined as follows:

```
type IntegratedDoubleCassettePlayer {
extends:
 CassettePlayer cp1, cp2;
redefines:
 [cp1, cp2] Switchable;
instance:
 op void setDeviceMode(DeviceMode mode);
 ...
}
```

Semantically this is equivalent to the method by method definition.

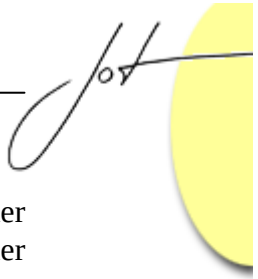
Polymorphic Use of Repeated Parts

If an object defined via repeated inheritance is assigned to a supertype variable which has the type of a repeated part, it is necessary to make clear which part is intended, i.e. which methods of the object are to be invoked polymorphically. To achieve this, the dot notation is used, as follows:

```
IntegratedDoubleCassettePlayer* dcp =
                 new IntegratedDoubleCassettePlayer.init();
CassettePlayer* cp = dcp.cp2;
// the cp2 methods can be invoked polymorphically
```

If a method associated with the part has been redefined, the method into which it has been merged is invoked polymorphically (using the dynamic scheduling mechanism) as if it were the original method of the supertype, as the following code illustrates.

```
cp.switchOn();  // the merged method is invoked
cp.play();      // the method cp2.play() is invoked
```



Invoking merged methods polymorphically may not be appropriate, but the programmer can easily avoid this in relevant cases by defining parts in an `includes` clause rather than an `extends` clause.

The Cast Statement

Semantically objects, not their parts, are assigned polymorphically to variables of supertypes, as in other OO languages, so that although it appears that only a part has been assigned, the entire object `dcp` is actually assigned to the variable `cp`. Hence a conditional downcast can be used to gain access to other parts of the object, e.g.

```
cast (cp) as {
 (CassettePlayer cp1) {/* statements using cp1 */}
 (IntegratedDoubleCassettePlayer idcp)
                     {/* statements using idcp */}
 (RadioCassettePlayer rcp) {/* statements using rcp */}
 else {/* optional statements if there is no match */}
}
```

This is the normal Timor cast statement. In the example the clause `IntegratedDoubleCassettePlayer idcp` would be selected, and the entire object would be accessible using the name `idcp`. However, in this form the programmer would not (easily) be able to determine which part had been assigned to `cp`. If this is important more specific clauses can be used in the cast statement, e.g.

```
(IntegratedDoubleCassettePlayer idcp[cp1]) {...}
(IntegratedDoubleCassettePlayer idcp[cp2]) {...}
```

In the example the second of these clauses would be selected, because the part used to assign the object to `cp` was `cp2`.

Casting takes repeated inheritance into account by supporting repetition, e.g.

```
cast (cp) as {
 [CassettePlayer cp1] {/* statements using cp1 */}
 (Radio r) {/* statements using r */}
}
```

A clause with square brackets indicates that the associated statements are repeated, i.e. if the object to which `cp` refers contains one or more `CassettePlayer` parts, they are executed for each matching part. (The `Radio r` statements are not repeated.)

In such cases it can be useful to execute not simply the code block for the first matching clause, but for example all the clauses which match. Hence the keyword `as` following the reference can be followed by a further keyword: `firstof`, `anyof` or `allof`, with the obvious meanings, whereby `anyof` non-deterministically selects any matching type. The default is `firstof`.

Non-repeated Parts

The idea that a base type can be given a part name is essential for repeated parts, but it can also be essential for identifying which methods of base types are to be merged, even

for cases not involving repeated inheritance. Furthermore, the use of part names for non-repeated parts can sometimes add symmetry. For these reasons Timor allows part names to be provided for base type parts not involving repeated inheritance. To illustrate this, consider an `IntegratedRadioDoubleCassettePlayer` device:

```
enum DeviceMode {playCassette1, playCassette2, playRadio}
type IntegratedRadioDoubleCassettePlayer {
extends:
 Radio r;                  // with part name
 CassettePlayer cp1, cp2;
redefines:
 [r, cp1, cp2] Switchable;
instance:
 op void setDeviceMode(DeviceMode mode);
 ...
}
```

which inherits from a type `Radio` as defined in section 2. Allowing a part name to be used in such a case not only allows appropriate methods to be redefined. It also allows clients to access the different parts uniformly.

Sometimes separate methods with identical signatures might be derived from different base types not involving repeated inheritance. This happens, for example, in a simple type `RadioCassettePlayer` defined without using part names, e.g.

```
type SimpleRadioCassettePlayer {
extends:
 Radio;
 CassettePlayer;
}
```

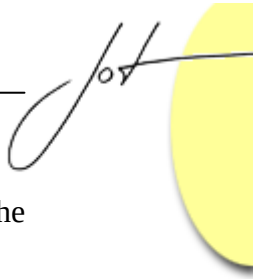
because each base type has the `Switchable` methods, and the intention is to keep them separate. In this case they are not automatically merged as described in [4], because a view is not regarded as a common ancestor. A compile time error occurs if the names are not resolved in the type definition.

One way of keeping the methods separate is to give the two base types part names, but then the client is forced always to use part names. To give greater freedom in such situations Timor allows optional part names to be defined, as follows:

```
type RadioCassettePlayer {
extends:
 Radio [r];                // the syntax [r] and [cp]
 CassettePlayer [cp];      // signifies optional part names
}
```

The client of such a type has the freedom to use part names or not, as it suits him, except in the case where this creates ambiguities. Thus he could write:

```
RadioCassettePlayer* rcp = new RadioCassettePlayer.init();
rcp.r.switchOn();       // part name compulsory
rcp.setStation();       // part name optional
```



If a client assigns such an object to a view reference, this is potentially ambiguous, so he must use the appropriate part name, e.g.

```
Switchable* s = rcp.cp;
```

From the implementor's viewpoint method names have no part name except where this would be ambiguous.

This technique can be used to handle entirely coincidental method collisions, such as arise in a type `GraphicalCardDealer` (cf. [1]), in which each of the base types `CardDealer` and `GraphicalComponent` have a parameterless method `draw`, with quite different semantics:

```
type GraphicalCardDealer {
extends:
 CardDealer [dealer];
 GraphicalComponent [graphics];
}
```

Here part names disambiguate the colliding methods. The result is two separate `draw` methods, which from the client's viewpoint can easily be distinguished as `dealer.draw()` and `graphics.draw()`. It would suffice for only one of the base types to be given a part name: the names of methods of the other would then simply be used without a part name.

6 IMPLEMENTING THE TYPES

Some key issues which must be understood about implementations in Timor are (a) that they are not themselves types, and (b) that each implementation must be a *complete* implementation of its type. However, a complete implementation does not necessarily mean a totally fresh implementation. At an earlier stage in the development of Timor a code re-use technique was proposed [3, 4] which permitted extensive re-use of existing code in a much more flexible manner than subclassing allows.

Following the crystallisation of ideas with respect to repeated inheritance the details of that re-use technique have been revised and simplified, without changing the basic concepts described in earlier papers. We begin with a simple example which was presented in [3] to illustrate how subclassing can be simulated. The extra notion now associated with the re-use technique, initially introduced to accommodate repeated inheritance, is that re-used code has a state, and so can be regarded as a variable in the `state` section. The effect of this change is that the mechanism has an interesting relationship with some forms of delegation, and we therefore motivate the discussion in this direction.

Relationship to Delegation

The standard OO code inheritance technique incrementally extends the code of a class in its subclass(es). This can be a problem if subclassing and subtyping do not match each

other. One way of avoiding an undesirable type relationship while nevertheless re-using code in the standard OO paradigm would be to use delegation. In the following we see how Timor can improve on this approach.

In Timor a type `DoubleEndedQueue` might be derived from a type `Queue` by inclusion (not by extension, as that would imply a behavioural subtype relationship). Given an implementation of `DoubleEndedQueue`, this implementation could be re-used to implement `Queue` by declaring a `DoubleEndedQueue` as an internal variable (cf. delegation). We begin with the type definitions.

```
type Queue<:ELEMENT:> {
maker:
 init(int maxSize);
instance:
 op void insertAtBack(ELEMENT e) throws FullEx;
 op ELEMENT removeAtFront() throws EmptyEx;
 enq ELEMENT front() throws EmptyEx;
 enq int length();
}
type DoubleEndedQueue<:ELEMENT:> {
includes:    // interface inheritance without subtyping
 Queue<:ELEMENT:>;
maker:
 init(int maxSize);
instance:
 op void insertAtFront(ELEMENT e) throws FullEx;
 op ELEMENT removeAtBack() throws EmptyEx;
 enq ELEMENT back() throws EmptyEx;
}
```

Let us assume that an implementation exists for `DoubleEndedQueue` which we want to re-use to implement `Queue`. This could be achieved in Timor as follows:

```
impl QueueImpl<:ELEMENT:> of Queue<:ELEMENT:> {
state:
 DoubleEndedQueue<:ELEMENT:> deq;
 // a normal variable declaration for delegated calls
maker:
 init(int maxSize){
  deq = DoubleEndedQueue.init(maxSize);
}
instance:
 op void insertAtBack(ELEMENT e) throws FullEx {
  deq.insertAtBack(e);
 }
 op ELEMENT removeAtFront() throws EmptyEx {
  return deq.removeAtFront();
 }
 enq ELEMENT front() throws EmptyEx {
  return deq.front();
 }
 enq int length() {
```

```
    return deq.length();
   }
  }
```

This has the nice property that *any* implementation of `DoubleEndedQueue` can be re-used, according to the normal rules of Timor. However, using delegation in this way is not only inefficient at run-time; it is also wasteful of programmer effort, because methods which in effect already exist in the implementation of `DoubleEndedQueue` have to be invoked indirectly via `QueueImpl`. Both these disadvantages can be eliminated, by providing a mechanism which allows the programmer to define that those methods associated with the variable `deq` which match the methods of `Queue` should in fact be treated directly as methods of `Queue`. This is in effect the mechanism which was defined in [3, 4], except that the re-used code was not defined as a variable but as a type. The matching rules are as defined in those papers.

Syntactically re-use variables are distinguished from other variables in the state section by prefixing them with a hat (`^`) symbol. As previously, more than one item can be re-used, and matching occurs in the order of the declarations. Here is how the example would actually appear in Timor:

```
impl QueueImpl<:ELEMENT:> of Queue<:ELEMENT:> {
state:
 ^DoubleEndedQueue<:ELEMENT:> deq; // a re-use variable
maker:
 init(int maxSize){
  deq = DoubleEndedQueue.init(maxSize);
}
 // no instance methods need to be coded
}
```

In this case all the public methods of `Queue` can be matched in `DoubleEndedQueue`. A match is not sought if an interface method of the type being implemented is explicitly coded in the `instance` section of an implementation.

In the present example it would in principle be possible to re-use the maker defined in `DoubleEndedQueue`, but allowing makers to be matched leads to problems in the general case, e.g. when more than one re-use variable is involved. Consequently only instance methods can be matched.

Imitating Subclassing

The above example does not illustrate how subclassing can be imitated in cases where the subclass requires access to the internal variables and methods of the superclass. To achieve this we follow a variant of the same principle, this time allowing a declaration of a re-use variable to be defined in terms of a specific implementation rather than a type. This technique was also described in an earlier form in [3].

To illustrate this approach, we assume that an implementation `ArrayQueue1` of `Queue` exists (cf. [3]). The aim is to extend this incrementally to provide an

implementation of `DoubleEndedQueue` (cf. `ArrayDEQ1` [3]) in the subclassing style, as follows:

```
impl ArrayDEQ1<:ELEMENT:> of DoubleEndedQueue<:ELEMENT:> {
state:
 ^ArrayQueue1 aq; // an implementation is used as a type
instance: // the methods not coded in ArrayQueue1
 op void insertAtFront(ELEMENT e) throws FullEx {
  if (aq.size < aq.maxSize)
   {aq.front--; if (aq.front < 0) aq.front = aq.maxSize - 1;
    aq.theArray[aq.front] = e; aq.size++;}
  else throw new FullEx();
 }
 op ELEMENT removeAtBack() throws EmptyEx {
  if (aq.size > 0)
   {aq.back--; if (aq.back < 0) aq.back = aq.maxSize - 1;
    aq.size--; return aq.theArray[aq.back];}
  else throw new EmptyEx();
 }
 enq ELEMENT back() throws EmptyEx {
  if (aq.size > 0)
   {int i = aq.back - 1;
    if (i < 0) i = aq.maxSize - 1;
    return aq.theArray[i];}
  else throw new EmptyEx();
 }
}
```

As was envisaged for the original re-use technique, nominating an implementation for re-use gives access to the internal methods and state of that implementation. Because in the revised approach it appears as a re-use *variable*, no special "super" keyword or special syntax is necessary for accessing or overriding these. (Hence the earlier `overrides` section has also become redundant.)

To make the code less tedious to write and easier to understand, Timor also supports a Pascal-like `with` statement, e.g.

```
op void insertAtFront(ELEMENT e) throws FullEx {
 with (aq) {
  if (size < maxSize)
   {front--; if (front < 0) front = maxSize - 1;
    theArray[front] = e; size++;}
  else throw new FullEx();
 }
}
```

This allows the programmer to use internal names exactly as they appear in the implementation being re-used.

Implementing Repeated Inheritance

An important advantage of the revised Timor re-use technique is that it takes into account not only *code* re-use but also *state* re-use. This greatly simplifies the implementation of types which use repeated inheritance. We begin with a type `RadioDouble-CassettePlayer`:

```
type RadioDoubleCassettePlayer {
extends:
 Radio r;                  // with part name
 CassettePlayer cp1, cp2;
}
```

which can be implemented as follows:

```
impl RadioDoubleCassettePlayerImpl of
                                RadioDoubleCassettePlayer {
state:
 ^Radio r = Radio.init();  // initialised re-use variables
 ^CassettePlayer cp1 = CassettePlayer.init();
 ^CassettePlayer cp2 = CassettePlayer.init();
}
```

If a part name appears in a type definition, this is significant for interface method matching purposes. If an implementation variable uses an identifier which differs from a part name in the type definition, a match does not occur. But as the `Queue` example illustrates, a base type in a type definition which is unnamed can be matched with a re-use variable regardless of its identifier, as all variables in an implementation must have identifiers.

This example illustrates that a straightforward mapping can exist from each named part in a type definition to an implementation of that part, provided that it has a parameterless maker `init()`. If, as in the above example, the type consists entirely of named parts (and/or abstract variables) the compiler automatically produces a standard implementation of the entire type (named after the type with the standard suffix `Impl`).

Each implementation must be complete in itself. Consequently a derived type need not re-use existing implementations (and on the other hand a base type can re-use implementations of other types). Thus an implementor of `RadioDoubleCassette-Player` can provide a completely fresh implementation. In this case all the public methods of the type have to be completely implemented. The new implementation must name all the methods unambiguously, e.g.

```
impl RadioDoubleCassettePlayerImpl2
                              of RadioDoubleCassettePlayer {
state:
 ...
instance:
 op void r.switchOn() {... /* new implementation code */}
 ...
 op void cp1.switchOn() {... /* new implementation code */}
```

```
 ...
 op void cp2.switchOn() {... /* new implementation code */}
 ...
}
```

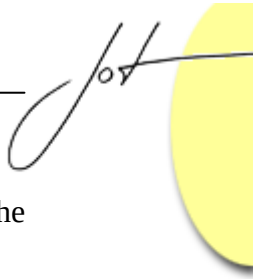
Using Redefined Methods

Even when public methods of a re-use variable have been redefined in a re-using type, these still exist internally as invocable methods. This is illustrated by a partial implementation of `IntegratedRadioDoubleCassettePlayer`, showing how its redefined switching methods might be implemented:

```
impl IntegratedRadioDoubleCassettePlayerImpl
                    of IntegratedRadioDoubleCassettePlayer {
state:
 ^Radio r = Radio.init();
 ^CassettePlayer cp1 = CassettePlayer.init();
 ^CassettePlayer cp2 = CassettePlayer.init();
 boolean isOn = false;
 DeviceMode currentMode = playRadio;
instance:   // a new implementation of the combined methods
 op void switchOn() {
  isOn = true;
  currentMode = playRadio;
  r.switchOn(); // the radio is on by default
  ...
 }
 ...
 op void setDeviceMode(DeviceMode mode) {
  currentMode = mode;
  if (mode == playRadio) {
    r.switchOn();
    cp1.switchOff();
    cp2.switchOff();
  }
  ...
 }
```

7 RELATED WORK

Eiffel [9] supports repeated inheritance primarily by allowing individual features to be renamed. There is no explicit support for the idea of "parts" which can be given separate identifiers. Consequently the replication of a part involves renaming each feature individually. Furthermore the introduction of arbitrary new names for features creates problems (which can be resolved in select clauses) that do not arise in Timor, where arbitrary renaming is avoided in favour of the qualification of existing names by means of part identifiers. Eiffel's use of repeated inheritance to imitate a "super" construct by



renaming methods in the "super instance" and redefining and selecting them in the "subtype instance" is of course unnecessary in Timor.

Sather [13] distinguishes between subtyping and subclassing. At the subtyping level there is no renaming facility, so that methods with identical (or in some cases with contravariantly conform) signatures inherited from different supertypes are automatically merged. Although it appears to be possible repeatedly to inherit from the same supertype, this leads to the effect that the repeatedly inherited methods are merged. At the level of code re-use the code of multiple concrete classes can be included into another class and here renaming is possible. Repeated code inclusion leads to replication, and in this case clashing features must be individually renamed.

Theta [7], like Timor strictly separates types from their implementations. At the type level multiple inheritance is possible, and methods can be individually renamed. There is no concept equivalent to a part identifier. At the implementation level only one base class can be re-used, independently of type relationships. A parts concept does not occur at the type or implementation level.

Although C++ [14] supports multiple inheritance and repeated inheritance the latter can only occur indirectly, i.e. in conjunction with diamond inheritance. As the latter is the theme of a companion paper [6], a fuller discussion of C++ diamond inheritance, including the nomination of base classes as virtual, is discussed there. However, it is of particular interest to the present paper that a client must resolve the names of conflicting methods by qualifying them with the names of the classes in which they are defined. In contrast, by allowing the definer of a type to qualify conflicting methods with freely chosen part identifiers Timor can handle repeated inheritance in an unproblematic way.

Java [1] supports multiple inheritance only at the type level (via interfaces) and provides no explicit support for repeated inheritance. Because multiply inherited methods with identical signatures are automatically merged, even simple arbitrary collisions create a problem.

Timor's separation of types and implementations allows these aspects of inheritance to be handled orthogonally. As was indicated in section 6 re-use variables can be seen as an efficient mechanism for some cases of delegation, and they can be used either to imitate subclassing or to reverse subclassing. The latter has the advantage that it is easier to conform with the information hiding principle, with the consequence that in Timor any implementation of an appropriate type can be "re-used" to implement another, independently of type relationships. This approach derives from the re-use technique proposed by Schmolitzky in [12], but differs from his proposal by adding the idea of state to the re-use technique, thus opening the way for a straightforward implementation of the kind of repeated inheritance discussed in this paper, and simplifying the constructs needed, e.g. by making special support for "super" superfluous (which is especially helpful in the context of multiple and repeated implementation inheritance).

Re-use variables superficially resemble the delegation technique used in prototype based languages. In each case "missing" interface methods are supplied from variables declared within the implementation. However, the differences are overwhelming. In

Timor this is merely an implementation technique, not affecting type relationships; re-use variables are implemented by value, not by pointer; the search for methods occurs at compile-time, not at run-time; the search is not recursive, etc.

8 CONCLUDING REMARKS

This paper has presented the Timor approach to parts inheritance and repeated parts inheritance. This is characterised by the idea that individual inherited parts of a subtype can be provided with part identifiers. This technique, which to our knowledge does not appear in other OO languages, has at least the following advantages:

a) Members of repeated parts can be unambiguously qualified using a part name.
b) Individual members of parts with part names can easily be combined (optionally) into a single member of the subtype.
c) At the implementation level the part names can appear as re-use variable identifiers, thus simplifying the implementation of types defined in terms of repeated inheritance. This also eliminates the need for a "super" construct and the naming problems to which this gives rise for multiple code inheritance.

In [4] we argued that different kinds of multiple inheritance are best handled by different mechanisms. In that paper we then concentrated on multiple inheritance from a common abstract ancestor, which has more in common with conventional type inheritance in OO languages. The relationship between that approach and the parts approach described in this paper, and in particular the issues which appear when they are combined, arise typically in cases of diamond inheritance from a common concrete ancestor. For example a base type `Person` might be specialised in different orthogonal ways (e.g. as a `Student` and as an `Employee`), and these can then be brought together to create a type `EmployedStudent`. This can also involve repeated inheritance (e.g. a `DoublyEmployedStudent`). In such cases some of the elements of both forms of multiple type inheritance discussed earlier occur in combination. These issues are discussed in a companion paper [6].

ACKNOWLEDGEMENTS

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

REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[2] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism," *Computing Surveys*, vol. 17, no. 4, pp. 471-522, 1985.

[3] 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.

[4] 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.

[5] 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.

[6] J. L. Keedy, G. Menger, and C. Heinlein, "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

[7] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers, "Theta Reference Manual," MIT Laboratory for Computer Science, Cambridge, MA, Programming Methodology Group Memo 88, February 1994.

[8] 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.

[9] B. Meyer, *Eiffel: the Language*. New York. Prentice-Hall, 1992.

[10] M. Mezini, "Dynamic Object Evolution without Name Collisions," ECOOP '97, 1997, Springer Verlag, LNCS, vol. 1241, pp. 190-219.

[11] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[12] A. Schmolitzky, "Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen (A Model for Separating Inheritance and Type Abstraction in Object Oriented Languages)," *Ph.D. Thesis, Dept. of Computer Structures*: University of Ulm, Germany, 1999.

[13] D. Stoutamire and S. Omohundro, "The Sather 1.1 Specification," International Computer Science Institute, Berkley, CA ICSI Technical Report TR-96-012, 1996.

[14] B. Stroustrup, *The C++ Programming Language, Third Edition*. Addison Wesley, 1997.

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/

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 Abstract variables are discussed in greater detail in [5], where the implications of nested abstract variables are examined.

3 Although a view can basically be considered as equivalent to a type, it is not regarded as a type with respect to the rules for handling common ancestors in multiple inheritance, see [6].