Support for Subtyping and Code Re-use in Timor

J. Leslie Keedy, Gisela Menger, Christian Heinlein
Department of Computer Structures
University of Ulm
89069 Ulm, Germany
{keedy,menger,heinlein}@informatik.uni-ulm.de

Abstract

Unlike most object oriented programming languages Timor, which has been designed to support component development, replaces the class construct with separate constructs for defining types and their implementations (which are not types). It also distinguishes between behaviourally conforming subtyping and the inclusion of behaviourally deviant interfaces in the definition of derived types. The separation of types and implementations simplifies a separation of subtyping and subclassing, facilitating the re-use of implementations of one type to implement other, unrelated types. A further technique allows a type to be mapped onto an unrelated type with different method names, such that the latter's implementations can also be re-used to implement the former. The paper concludes by outlining a substantial example based on the Timor Collection Library.

Keywords: Software component, type, subtyping, subclassing, polymorphism, behavioural subtyping, base type, derived type, code re-use, code mapping.

1 Introduction

One of the most significant differences between object oriented and other programming languages is their explicit support for subtyping. The basic idea is that a type (the supertype) can be used as a base for defining a new type (the subtype). The subtype can redefine existing methods of the supertype and can add new members. Instances of subtypes can be assigned to variables of the supertype. This use of subtypes is known as inclusion polymorphism (Cardelli and Wegner, 1985).

Because existing methods can be redefined, the behaviour of a subtype instance can be quite different from that of its supertype, even when used polymorphically as if it were an instance of the supertype. It is therefore useful to distinguish between *behavioural* and *non-behavioural* subtypes. Liskov and Wing (1994) have developed a behavioural notion of subtyping which differs from earlier definitions of the subtype relationship in that attention is paid not only to the redefinition of existing methods but also to the significance of new methods, which can affect the behaviour of existing methods in the presence of aliasing and also in a general computational environment that allows multiple users to share mutable objects.

This strong definition of behavioural subtyping excludes some cases of subtyping which at first sight may appear to be behavioural, for example the relationship between a type `Queue` (as the supertype) and a type `DoubleEndedQueue` (as the subtype). If a programmer uses a `DoubleEndedQueue` instance as a `Queue` instance in his program it will, assuming that it has been defined and implemented in a reasonable way, work perfectly well as a `Queue` in his isolated context. But if the same object instance is accessed via other variables as a `DoubleEndedQueue` (e.g. using a method which inserts entries at the "wrong" end) it can exhibit properties which do not conform behaviourally with the usual definition of a `Queue` type.

On the other hand there are many cases where a programmer has no intuitive expectation that a subtype will exhibit the same behaviour as a supertype. For example a supertype `Button` defined for use in a graphical user interface may have an operation `push`, the behaviour of which is specifically intended to vary depending on the kind of button which it actually is, as defined in more detail in particular subtypes.

Definitions of the subtype relationship can also create problems at the implementation level. For example, in class-based languages the class construct is typically used to achieve both *subtyping* (i.e. a type relationship) and *subclassing* (a code re-use relationship), although these are often not compatible with each other, cf. e.g. Cook et al. (1990). For example the code of a supertype `Queue` can easily and effectively be re-used in a subtype `DoubleEndedQueue`, but the fact that this is not a behavioural subtype suggests that in some applications the subtype relationship should be avoided; however this creates the dilemma that the code cannot be re-used.

In this paper we describe how such issues are handled in the programming language Timor, which is currently being designed at the University of Ulm in Germany. The motivation for Timor is to develop an object-oriented language which is suitable for defining and implementing software components, whereby the word *component* here is to be understood in the original sense described by McIlroy (1968) (who used a sine routine as his example). Timor is intended to be suitable for developing software components of various sizes, including quite small components corresponding to fairly trivial classes in typical object oriented applications. This does not

Copyright © 2002, Australian Computer Society, Inc. This paper appeared at the *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia. Conferences in Research and Practice in Information Technology, Vol. 10. James Noble and John Potter, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

preclude an intention also to support the development of larger components, as the word is frequently understood today.

Timor supports not only single but also multiple inheritance, both at the type and implementation levels. However, in the present paper discussion is restricted to single inheritance, which is adequate to illustrate the three main issues which we wish to discuss. In section 2 we describe the separation of types and implementations, which in most object oriented languages can only be simulated via inheritance (at the cost of introducing unintended types). Sections 3 and 4 outline the differing treatment of behavioural and non-behavioural subtypes. Sections 5 to 8 show how code can be flexibly re-used to implement types which are not necessarily related to each other. An example is provided in section 9. Section 10 provides a comparison with related work and section 11 summarises the paper, adding some final remarks. In later papers we shall describe how these ideas are extended to encompass multiple (and repeated) type and implementation inheritance.

2 Distinguishing Types and Implementations

In a component development environment it is important to be able to develop different implementations for the same type (Figure 1). For example a type `List` might be implemented as an array, a linked list, etc. This simple requirement led us to replace the class concept with a concept which distinguishes between types and their implementations.

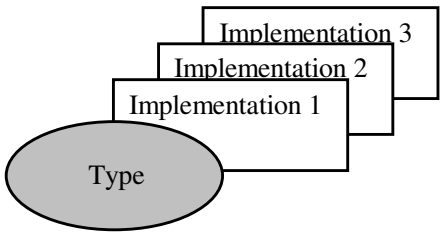

Figure 1: A Type with Multiple Implementations

A Timor type is defined as follows[1]:

```
type Queue {
 maker init(int maxSize);
 op void insertAtBack(ELEMENT e) throws FullEx;
 op ELEMENT removeAtFront() throws EmptyEx;
 enq ELEMENT front() throws EmptyEx;
 enq int length();
}
```

This can have several implementations. Here is an array-based implementation:

```
impl ArrayQueue1 of Queue {
 ELEMENT[] theArray;
 int maxSize;
 int size = 0;
 int front = 0;
 int back = 0;

 maker init(int maxSize){
  this.maxSize = maxSize;
  theArray = new ELEMENT[maxSize];
 }
 op void insertAtBack(ELEMENT e) throws FullEx {
  if (size < maxSize)
   {theArray[back] = e; back++;
    if (back == maxSize) back = 0; size++;}
  else throw new FullEx();
 }
 op ELEMENT removeAtFront() throws EmptyEx {
  if (size > 0)
   {ELEMENT temp = theArray[front]; front++;
    if (front == maxSize) front = 0; size--;
        return temp;}
  else throw new EmptyEx();
 }
 enq ELEMENT front() throws EmptyEx {
  if (size > 0) return theArray[front];
  else throw new EmptyEx();
 }
 enq int length() {return size;}
}
```

There are many other possible implementations of `Queue`, which could be written as separate implementation components and given different names. The important point here is that the behaviour of different implementations of a type must be equivalent to each other. We call this *behavioural equivalence*. It differs from behavioural conformity in that a subtype relationship is not involved, i.e. the behaviour of members cannot be re-specified and new public members cannot be added in an implementation.

3 Defining the Behaviour of Components

In a situation where components are developed and used by different groups of programmers it is important that all concerned have a clear understanding of how components behave. We have just introduced the term behavioural equivalence and have already referred in the introduction to the notion of behavioural conformity as a concept associated with subtyping.

Formally, both of these notions must be defined in terms of a formal specification. We intend to add a specification technique in later versions of Timor, but in the first version behavioural equivalence can only be defined intuitively, as the equivalent fulfilment by different implementations of a type definition (with the help of comments). Similarly behavioural conformity has initially to be understood intuitively, in the spirit of the Liskov and Wing definition.

Suppose, however, that we already had a specification technique (and that it were powerful enough to describe the behaviour of types to a degree of detail and accuracy that we need). This would mean that different implementations of a type could only be described as behaviourally equivalent provided that they fulfil the

[1] The qualifier `maker` introduces an explicitly named constructor. The qualifier `op` introduces an operation (which can modify the state of an instance of the type), `enq` introduces an enquiry (which cannot modify the instance's state). The distinction between `op` and `enq` methods is important for example for defining qualifying types with bracket routines, cf. Keedy et al. (1997), Keedy et al. (2000), but is not significant for the present discussion. The type `ELEMENT` can be thought of as any relevant type. Timor supports a generic concept along the lines described in Evered (1997), Evered et al. (1997), but again this is not directly relevant to our discussion and is not described here.

specification for the type. Similarly subtype specifications could only be described as behaviourally conforming in so far as they fulfil the specification of the supertype. Put another way, an "implementation" of a type would not be a valid implementation if it did not fulfil the specification of the type, and a "subtype" specification would not define a behaviourally conforming subtype if it did not conform with the specification of the supertype.

We assume in this context that a formal specification would be precise in the sense of stating requirements which have to be fulfilled, but at the same time it could leave freedom for different actual behaviours to fulfil these requirements. Thus for example an abstract type `Collection` might specify non-deterministically that following an `insert` operation the collection size would either be increased by one or would not change. In this way a type `Bag` (which accepts duplicates), a type `Set` (which ignores duplicates) and a type `Table` (which throws an exception when an attempt is made to insert a duplicate) could be behavioural subtypes of the type `Collection`. In some cases it can also be appropriate to provide a null specification, in which case *any* syntactically correct implementation fulfils the specification and *any* syntactically correct subtype definition conforms behaviourally with its supertype.

4 Derived Types

Timor supports the definition of types on the basis of other types in a form which resembles the conventional object-oriented style of subtyping. Such types are known as *derived* types (Figure 2).

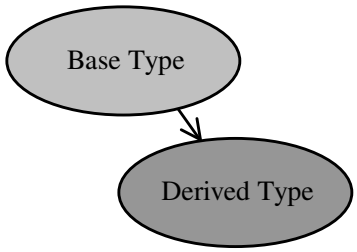

Figure 2: A Base Type with a Derived Type

In definitions of derived types a distinction is drawn between genuine subtyping, based on the behavioural notion, and the use of a base type simply as a mechanism for including interface definitions in a new type without implying a subtyping relationship. Where behavioural conformity is intended the supertype is introduced by the keyword `extends`, e.g.

```
type Collection {
 ...
 op void insert(ELEMENT e) throws DuplicateEx;
}
type Bag
 extends Collection
 redefines {
 op void insert(ELEMENT e);
  // the insert method for a Bag
  // does not throw a DuplicateEx.
  }
{ /* no new methods in this example */}
```

A non-behavioural relationship is introduced by the keyword `includes`, e.g.

```
type DoubleEndedQueue
 includes Queue {
 maker init(int maxSize);
 op void insertAtFront(ELEMENT e) throws FullEx;
 op ELEMENT removeAtBack() throws EmptyEx;
 enq ELEMENT back() throws EmptyEx;
}
```

Because the *inclusion* of a base type in another type does not imply a subtyping relationship, component instances of the derived type cannot be assigned to variables of the base type. Thus a component of type `DoubleEndedQueue` cannot be assigned to a `Queue` variable, but a component of type `Bag` can be assigned to a variable of type `Collection`.

If a method of a supertype is changed in a derived type (for extensions in a behaviourally conforming manner) this must appear in a `redefines` clause. In principle this requires a new formal specification of the method(s) involved, but in the first version of Timor this is strictly speaking only relevant for single inheritance[2] in cases where some exceptions defined in a method of the supertype cannot be thrown in the derived type, as is illustrated in the type `Bag`. Since similar changes can be defined for methods of included types, these may also appear in a `redefines` clause.

As the intended behaviour of a method can in principle change in the first version of Timor even if the signature does not, programmers must list such members in a `redefines` clause even where the signature does not change. In practice the `redefines` clause can be viewed as a list of methods which in an implementation can be overridden in the object oriented sense. Such changes must conform behaviourally with an `extends` supertype but not with an `includes` base type. We anticipate that some (though not exhaustive) checking of behavioural conformity for methods appearing in a `redefines` clause may be possible when a specification technique is added.

5 Implementing Types

An implementation of a type is considered to be an implementation of *all* the members of the type. This can take several forms:

(a) A type (including a derived type) can have a completely new implementation (Figure 3).

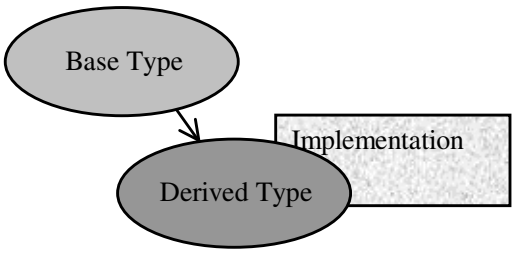

Figure 3: An Independent Implementation of a Derived Type according to the Information Hiding Principle

[2] Timor supports multiple type inheritance, i.e. more than one type can appear in the `extends` and/or `includes` clauses. In that case the `redefines` clause is also used for the clarification of name collisions.

This is well suited to the information hiding principle (Parnas, 1972). The new implementation of the methods of supertypes must conform with the specifications of the supertypes (where relevant as redefined in the derived type). The implementation of new and redefined members must conform with the specification of the derived type.

(b) A type (including a derived type) can re-use implementations of other types (indicated by the keyword `reuses`). In contrast with standard OO practice a subtype relation between the type of the new implementation and the types of its re-used implementations need not (but can) exist. Thus code re-use can be completely decoupled from subtyping and from the inclusion of interfaces.

A `reuses` clause can designate a specific *implementation* to be re-used (Figure 4). This typically reflects the conventional object oriented style of code inheritance, as is illustrated in section 6.

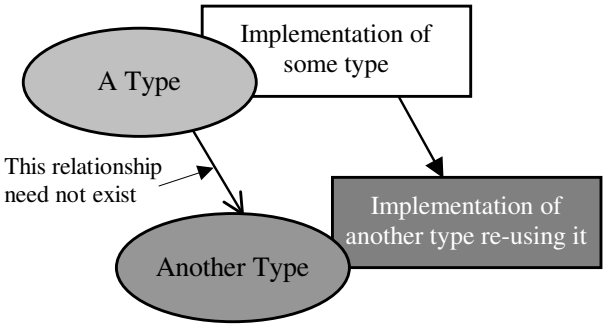

Figure 4: Reuse of a Specific Implementation

Alternatively, it can designate a *type*, any of whose implementations can be re-used (at the level of the public members) (Figure 5). This leads to a quite different style of code re-use, illustrated in section 7.

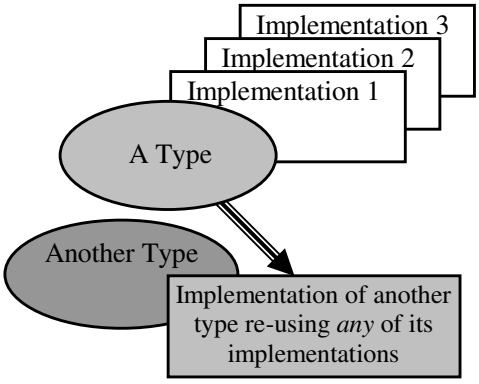

Figure 5: Reuse of any Implementation of a Type

(c) A type can be mapped to another type, and in this way re-use its implementations, again without implying a type relationship (Figure 6). This is illustrated in section 8.

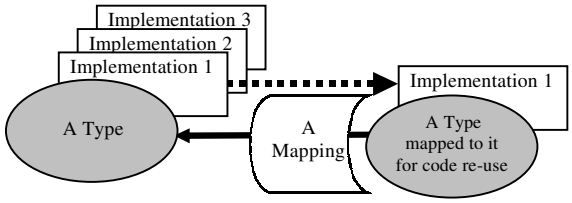

Figure 6: Mapping a Type to another Type

Finally, Timor also allows the possibility of *typeless implementations*. These are free-standing implementations which have no effect on the type system, but which can be freely re-used in the implementations of types.

6 Simulating Subclassing via Code Re-use

In section 2 we illustrated an implementation `ArrayQueue1` of a type `Queue` and in section 4 we showed how the type `Queue` might have a (non-behavioural) derived type `DoubleEndedQueue`. Here an implementation of `DoubleEndedQueue` reuses the *implementation* `ArrayQueue1` in a way which resembles the conventional code inheritance technique (i.e. subclassing):

```
impl ArrayDEQ1 of DoubleEndedQueue
 reuses ArrayQueue1 {
 op void insertAtFront(ELEMENT e) throws FullEx{
  if (size < maxSize)
   {front--; if (front < 0) front = maxSize - 1;
    theArray[front] = e; size++;}
  else throw new FullEx();
 }
 op ELEMENT removeAtBack() throws EmptyEx{
  if (size > 0)
   {back--; if (back < 0) back = maxSize - 1;
    size--; return theArray[back];}
  else throw new EmptyEx();
 }
 enq ELEMENT back() throws EmptyEx {
  if (size > 0)
   {int i = back - 1;
    if (i < 0) i = maxSize - 1;
    return theArray[i];}
  else throw new EmptyEx();
 }
}
```

In this example, the implementation of the derived type nominates an *implementation* of the type which is to be re-used (Figure 7). All the methods of a re-used implementation whose headers match the methods of the type being implemented are "inherited" (along with any data structures and methods which they need); any other methods are ignored.

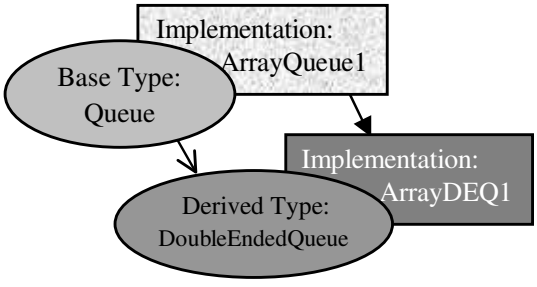

Figure 7: Simulating Conventional Subclassing

Here we see one of the advantages of explicitly naming constructors (rather than using the type name as a

constructor name, as in Java). In appropriate cases – as here with respect to the constructor from `ArrayQueue1` defined in section 2 – a constructor can be re-used in an implementation of a different type without having to be separately coded (and where appropriate explicitly call a constructor of the supertype).

The nomination of a specific implementation of some other type for re-use is typical in cases where code re-use follows the conventional OO inheritance paradigm of incrementally inheriting from base types (whether or not the inheritance is behavioural), where access is needed to data structures. While this style of code re-use is possible in Timor, it is not the preferred style, as it violates the information hiding principle and it does not result in modular components. We now illustrate the alternative style, which is possible because code re-use can be decoupled from type relationships.

7 Code Re-Use without Type Relationships

Because for subtyping reasons `DoubleEndedQueue` was defined by including `Queue` rather than by extending it, the two types are not related from the polymorphic viewpoint. In practice `DoubleEndedQueue` could therefore have been defined as a separate type, as follows:

```
type DoubleEndedQueue{
 maker init(int maxSize);
 op void insertAtFront(ELEMENT e) throws FullEx;
 op void insertAtBack(ELEMENT e) throws FullEx;
 op ELEMENT removeAtFront() throws EmptyEx;
 op ELEMENT removeAtBack() throws EmptyEx;
 enq ELEMENT front() throws EmptyEx;
 enq ELEMENT back() throws EmptyEx;
 enq int length();
}
```

Regardless whether `DoubleEndedQueue` was defined as a separate type or as a derived type, it has the above interface, and it can be treated as a separate type for implementation purposes. (This holds also for a behaviourally conforming subtype: *any* type can be implemented using any of the techniques described in section 5, regardless of its type relationships.)

A significant advantage of implementing any type without reusing a specific implementation of another type (i.e. without using the technique illustrated in section 6) is that this can be done in conformance with the information hiding principle. The type `DoubleEndedQueue` could have several such implementations, e.g. `ArrayDEQ`, `LinkedListDEQ` which do not re-use other code. Given such independent implementations, *all of them* could potentially be re-used to provide implementations of `Queue`. The following illustrates how this is achieved in Timor:

```
impl Queue1 of Queue
 reuses DoubleEndedQueue
{/* no re-implemented methods */}
```

The `reuses` clause in this example nominates a *type* rather than an implementation, indicating that any implementation of the type can be re-used (Figure 8).

This has the advantage that for any new implementation of `DoubleEndedQueue` there is automatically a new implementation of `Queue`. The basic rule is that any method which appears in the re-used type with an identical signature is re-used, unless it is overridden in the new implementation[3].

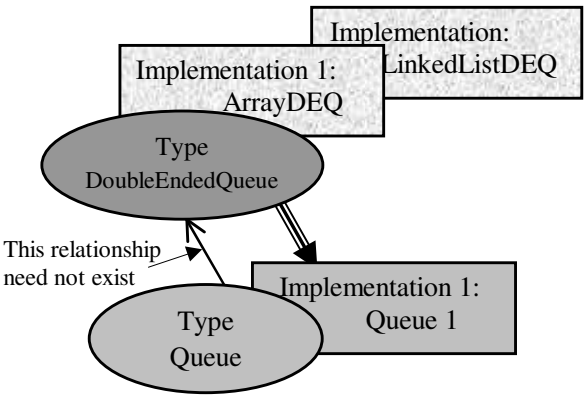

Figure 8: Reusing any Implementation of a Type

Some methods of `DoubleEndedQueue` (e.g. `insertAtFront`) cannot be explicitly invoked by clients of `Queue`. Whether such redundant methods (and fields) are removed from the implementation `Queue1` depends on the compiler (or possibly the component developer). They can of course only be removed if an analysis by the compiler shows that they genuinely are redundant (e.g. if methods are not invoked from the methods which are required).

8 Mapping Types onto Other Types

Implementations of the type `DoubleEndedQueue` could be easily re-used to implement unrelated types which do not have matching member definitions in their types. For example a type `Stack` might be defined as follows:

```
type Stack {
 maker init(int maxSize);
 op void push(ELEMENT e) throws FullEx;
 op ELEMENT pop() throws EmptyEx;
 enq ELEMENT top() throws EmptyEx;
 enq int length();
}
```

To allow the re-use of implementations of `DoubleEndedQueue` for `Stack` the following *map* can be provided:

```
map StackMap1 from Stack to DoubleEndedQueue {
 op void push(ELEMENT e) => insertAtFront;
 op ELEMENT pop() => removeAtFront;
 enq ELEMENT top() => front;
}
```

Members of the mapped type which already match (e.g. in this example the `length` method and the constructor) can be omitted from the map. Unmapped members must be implemented in some other way. When such a map exists, any implementation of the map's destination type (here `DoubleEndedQueue`) can be used to implement the mapped type, e.g.

```
impl Stack1 of Stack
       reuses DoubleEndedQueue via StackMap1 {}
```

This relationship is illustrated in Figure 9.

---

[3] Overriding is not illustrated in our examples, but it does not differ significantly from other overriding techniques which allow the code of an overridden method to be invoked using a `super` construct.

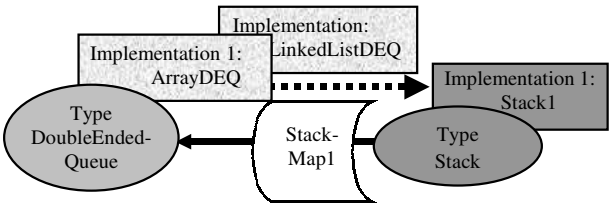

Figure 9: Mapping a Stack onto a Double Ended Queue

In the first version of Timor maps are kept simple, i.e. there must be an exact signature match; only the names of members and of parameters may differ between a mapped member and the member onto which it is mapped.

9 Larger Scale Application

We now briefly describe a realistic library of types and implementations, developed using the concepts of behavioural derived types and code re-use. This is the Timor Collection Library, the design of which is based on one co-author's doctoral thesis (Menger, 2000).

The aim was to provide a library of collection types and implementations which orthogonally support the following properties:

(a) handling the duplication of elements in three forms:
- collections which allow duplicate elements to be inserted,
- collections which ignore attempts to insert duplicates,
- collections which signal as exceptions attempts to insert duplicates.

(b) handling the ordering of elements in three forms:
- unordered,
- user-ordered,
- automatically sorted on the basis of user-defined criteria.

This leads to nine (three times three) concrete types, which are organised in a behaviourally conforming hierarchy that also includes four abstract types (`Collection`, `DuplicateFreeCollection`, `UserOrderedCollection`, `SortedCollection`). Multiple type inheritance (not described in this paper) was used to achieve a maximum of behavioural subtyping among related types.

Implementing this library using the re-use technique primarily involves coding only two of the concrete types:

- `List`: duplicates allowed, user-ordered; and
- `SortedList`: duplicates allowed, sorted.

Furthermore implementations of the type `List` can be almost completely re-used in implementations of the type `SortedList`. In addition two typeless implementations are needed to provide trivially redefined routines which bracket the `insert` routines of `List` and `SortedList`. These check for attempts to insert duplicate elements into appropriate types, in one case throwing an exception.

To provide alternative implementations of the entire library (as arrays, linked lists, doubly linked lists) requires only a new implementation of `List` and (a few methods of) `SortedList`. The implementations which carry out duplicate checking can be re-used with any implementations of the `List` and `SortedList` types. Customised implementations of some types can of course be provided separately, e.g. an implementation of `Set` (duplicates ignored, unordered) and of `Table` (duplicates signalled, unordered) as bit lists, for cases where the element types can be enumerated.

The result is a fully orthogonal, behaviourally conforming collection library with a maximum of code re-use as well as conformance with the information hiding principle. It can easily be extended by users to support their own specialised types, and it can be quickly re-implemented using new data structures.

10 Related Work

Code reuse via inheritance is often considered to be the most important benefit of object oriented programming. But earlier OO languages, where classes unify types and their implementations (e.g. Eiffel, C++), suffer from the problem that subtyping and subclassing do not always fit well together, cf. e.g. LaLonde and Pugh (1991). Nevertheless, even without language support for a clear separation between types and implementations, it is sometimes possible to achieve a surprisingly high degree of separation between types and subtyping on the one hand and implementations and code reuse on the other hand.

Taking C++ (Stroustrup, 1997) as an example, "pure" abstract classes (i.e. classes containing only pure virtual functions) can be used to model a type hierarchy, while normal classes can be used to build an independent implementation hierarchy. Private or protected inheritance, which prevents a subtype relationship between base and derived classes, can be used to represent the implements and reuses relationships of Timor. In combination with *access* or *using* declarations, even the includes relationship can be simulated, without creating a subtype relationship. But as C++ has not been designed to support such a clean separation of types and implementations, its enforcement by the programmer is somewhat artificial and more or less tedious. Especially in cases where an implementation of a completely unrelated type is to be reused to implement a type, it is necessary to explicitly merge the abstract methods of the type being implemented with the corresponding methods of the reused implementation by providing definitions of the methods which forward their calls to this implementation. Furthermore, simply reusing *all* implementations of a type to implement another type, requires similar method definitions which delegate their work to the methods of the reused type.

For more than a decade the relationship between subtyping and subclassing has been discussed in the academic area and languages (e.g. Sather, Signatures and Brew, Java, Theta, Tau) have been put forward which attempt to separate types and implementations with the aim of improving both.

Sather (Stoutamire and Omohundro, 1996) is a descendant of Eiffel (Meyer, 1992) with a fundamentally changed type system. Different kinds of relationships may coexist in a single hierarchy, which has an abstract class as its root. Abstract classes (without code) model type relationships, while concrete classes (which may only appear in leaf nodes) are subtypes of the abstract class(es) which they implement and are themselves types. Furthermore, abstract classes can be added retrospectively to the hierarchy as supertypes of existing classes. Code reuse is achieved by means of an inclusion mechanism, which copies code at the source level. *Partial classes* are classes which can contain code but which are not associated with a type; they can neither be instantiated nor used for variable declarations. Including code from other classes does not imply a type relationship. Static type safety is maintained by means of contravariant type rules. This approach suffers from the fact that types and implementations are not fully decoupled from each other, i.e. concrete classes are also types.

Signatures (Baumgartner and Russo, 1995) have been suggested (and implemented in the GNU C++ compiler) as an extension of C++ which supports the separation of types and implementations more directly and naturally. The basic idea is that signatures, like abstract classes in Sather, represent types which can be implemented in classes, also with the possibility that they can be retrospectively specified. However, signatures cannot serve as complete type definitions, as they may not contain constructors. Unlike Sather they are not part of the class hierarchy. But as signatures have been designed as a conservative extension of C++, they are unable to overcome inherent deficiencies of the base language, especially the fact that classes are also types. With Brew (Baumgartner et al., 1996), the signature concept has been realized in a Java-based language.

Java interface types (Arnold et al., 2000) have similarities with both Sather's abstract classes and with signatures. However, the separation of interfaces from their implementations is more explicit and easier to understand than the Sather approach, because, as with signatures, interfaces are not part of the class hierarchy. On the other hand not only are classes also types, but subclassing also implies a subtype relation. Principles such as information hiding and programming to interfaces can easily be circumvented by using classes without giving up subtype polymorphism (as in Sather). Nevertheless the Java type system is much simpler than that of Sather. Like signatures, Java interfaces may not contain constructors. In contrast with both Sather and the signatures approach, interfaces may not be defined retrospectively.

In Theta (Liskov et al., 1994) types are cleanly distinguished from classes, and the idea of multiple implementations is supported. Subtyping occurs between types, while classes are not types but only implementations of types, so that code reuse is as flexible as in Sather. Constructors may not appear on the type interface, thereby hindering strict programming to interfaces and also induction based type verification (because there is no starting point). Nevertheless, the clear distinction between types and implementations is a significant step forward.

Tau (Schmolitzky, 1999) is a radical adaptation of Java which realises the notion of *abstract typing*. The approach is similar to that found in Theta, in that subtyping occurs between types (called *interfaces* in Tau). Classes are not types, and subclassing does not imply any type relationship. Tau also supports typeless classes (similar to partial classes in Sather), which are intended only for code reuse. A decisive further step is taken in Tau, namely that, in contrast with previous languages, type interfaces can include *abstract constructors*.

Timor is based on the concept of abstract typing in Tau and adds some new features which are intended to be useful for developing software components. It emphasises the significance of behavioural subtyping by permitting a programmer to define derived types in a subtype relationship (where behavioural conformity is intended), while also allowing type interfaces to be included in derived types without implying such a relationship. Both forms of derived types allow inherited methods to be redefined, but they must be listed in a `redefines` clause, thus making it easy to identify the "hot spots" of an implementation. Code reuse can be achieved in a similar manner to Tau, in a way which resembles subclassing. However, Timor goes further than Tau by allowing a *type* to be named in a `reuses` clause, thereby indicating that any implementation of that type can be re-used, with the advantage that the information hiding principle can also be preserved between implementations. This applies also to the mapping technique, which allows one type interface to be mapped onto another for code reuse purposes.

11 Summary and Final Remarks

The paper has briefly described how the new programming language Timor[4] handles a number of issues relating to types and subtyping as well as implementations and code re-use. Aspects relating to multiple type inheritance and multiple code re-use have, however, not been discussed, as these raise a different set of issues which will be discussed in a future paper.

Motivated by the idea of regarding object oriented units potentially as general purpose components for use in a wide range of application systems, the language distinguishes between *types* and their *implementations* (which are not also types). This division of the traditional class construct into separate units has a number of advantages. From the viewpoint of component development and use it allows a single type to be implemented in different ways (e.g. reflecting different time and space tradeoffs), and these separate implementations can be sold/bought as separate components. Thus the existence of multiple implementations for a type does not necessarily imply

[4] The current state of Timor is that it is nearing the end of the design stage. No compiler is yet available.

that more than one implementation will be used in a particular application program or system[5]. It does, however, help to simplify some of the problems associated with subtyping and subclassing.

A user of components representing types and their implementations needs to understand how these are likely to behave, especially with respect to the issue of subtyping. In derived types, which represent a variation of the traditional form of subtyping, a distinction is drawn between *extending* a supertype in a behaviourally conforming manner and *including* base types to which the derived type need not conform behaviourally. In the absence of a specification technique in the first version of Timor, the behavioural requirement of extension subtypes must be regarded as a statement of intent by the designer rather than a feature which can be checked by the compiler. Nevertheless there is provision for a `redefines` clause in derived types which allows the designer to indicate where behavioural changes occur.

At the implementation level Timor supports the conventional code inheritance technique (using a `reuses` clause which typically nominates an *implementation* to be re-used). However, as the example in section 6 shows, this technique effectively encourages programmers to circumvent the information hiding principle by exporting major data structures. This can be avoided by using the `reuses` clause to designate a *type*, which need not be a supertype of the type being implemented. As described in section 7, the information hiding principle is thereby preserved and any implementation of the designated type is allowed to be re-used.

Finally a mapping technique is introduced (section 8) which enhances the possibilities of code re-use by allowing methods of one type to be mapped onto methods of another type such that the mapped type can be implemented by re-using implementations of the type onto which it is mapped. At present this technique requires an exact match of signatures (except the names of methods and their parameters), but we anticipate that the technique will be made more flexible in future versions of Timor.

The example of the Timor Collection Library, briefly outlined in section 9, provided us with confirmation of our view that it is worthwhile in the design of a new language to keep orthogonal ideas (such as types and implementations, subtyping and subclassing, behavioural conformity and behavioural divergence) separate, even if the language gets marginally larger. It is not a significant problem later to allow short-cuts for simple cases, such as allowing "classes" in the form of implementations which implicitly define their own types.

In summary Timor supports ideas which have already been proven in other languages and adds new ones which are considered to be useful for designing and implementing software components.

Acknowledgement: Special thanks are due to Dr. Mark Evered and Dr. Axel Schmolitzky for their invaluable contributions to the development of Timor ideas.

12 References

ARNOLD, K., GOSLING, J., and HOLMES, D. (2000): *The Java Programming Language, Third Edition*, Addison-Wesley.

BAUMGARTNER, G., LÄUFER, K., and RUSSO, V.F. (1996): On the Interaction of Object-Oriented Design Patterns and Programming Languages, Department of Computer Sciences, Purdue University, West Lafayette, IN.

BAUMGARTNER, G., and RUSSO, V.F. (1995): Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++. *Software - Practice and Experience* **25**(8): 863-889.

CARDELLI, L., and WEGNER, P. (1985): On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys* **17**(4): 471-522.

COOK, W., HILL, W., and CANNING, P. (1990): Inheritance is Not Subtyping. *17th ACM Symposium on Principles of Programming Languages*: 125-135.

EVERED, M. (1997): Unconstraining Genericity. *24th International Conf. on Technology of Object-Oriented Languages and Systems*, Beijing: 423-431.

EVERED, M., KEEDY, J.L., MENGER, G., and SCHMOLITZKY, A. (1997): Genja - A New Proposal for Genericity in Java. *25th International Conf. on Technology of Object-Oriented Languages and Systems*, Melbourne.

KEEDY, J.L., ESPENLAUB, K., MENGER, G., SCHMOLITZKY, A., and EVERED, M. (2000): Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes. *6th International Conference on Software Reuse*, Vienna.

KEEDY, J.L., EVERED, M., SCHMOLITZKY, A., and MENGER, G. (1997): Attribute Types and Bracket Implementations. *25th International Conf. on Technology of Object-Oriented Languages and Systems*, Melbourne.

LALONDE, W.R., and PUGH, J.R. (1991): Subclassing ≠ Subtyping ≠ Is-a. *Journal of Object-Oriented Programming* January 1991: 57-62.

LISKOV, B., CURTIS, D., DAY, M., GHEMAWAT, S., GRUBER, R., JOHNSON, P., and MYERS, A.C. (1994): Theta Reference Manual, MIT

[5] Techniques used in Timor for handling the coexistence of different implementations in a single program and the selection of particular implementations are not discussed in this paper, as these themes are not relevant to the issues of subtyping which the paper addresses.

Laboratory for Computer Science, Cambridge, MA.

LISKOV, B., and WING, J.M. (1994): A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* **16**(6): 1811-1841.

MCILROY, M.D. (1968): Mass Produced Software Components. *NATO Conference on Software Engineering, NATO Science Committee*, Garmisch, Germany: 88-98, Petrocelli-Charter.

MENGER, G. (2000): Unterstützung für Objektsammlungen in statisch getypten objektorientierten Programmiersprachen (Support for Object Collections in Statically Typed Object Oriented Languages): *Dept. of Computer Structures*, University of Ulm, Germany.

MEYER, B. (1992): *Eiffel: the Language*. New York, Prentice-Hall.

PARNAS, D.L. (1972): On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**(12): 1053-1058.

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

STOUTAMIRE, D., and OMOHUNDRO, S. (1996): The Sather 1.1 Specification, International Computer Science Institute, Berkley, CA.

STROUSTRUP, B. (1997): *The C++ Programming Language (third edition)*, Addison Wesley.