Qualifying Types
Illustrated by Synchronisation Examples

J. Leslie Keedy[1], Gisela Menger[1], Christian Heinlein[1] and Frans Henskens[2]

[1]Department of Computer Structures
University of Ulm
D-89069 Ulm
Federal Republic of Germany

[2]School of Electrical Engineering
and Computer Science,
University of Newcastle, N.S.W. 2308
Australia

{keedy, menger, heinlein}@informatik.uni-ulm.de henskens@cs.newcastle.edu.au

Abstract. Qualifying types represent a new approach to modifying the behaviour of instances of other types in a general way, in the form of components which can be designed and implemented without a prior knowledge of the types to be modified or their implementations. This paper illustrates the idea by showing how they can be used to program various standard synchronisation problems, including mutual exclusion, reader-writer synchronisation and several variants of the bounded buffer problem.

1 Introduction

Advocates of aspect oriented programming (AOP) [11] rightly emphasise that much can be gained by separating different aspects of programmed applications. The aim in AOP is to separate the descriptions of various aspects of a software system such as class hierarchies, functionality and synchronisation [7] in order to increase readability and facilitate changes. These separate descriptions are then combined by an "aspect weaver" into a standard programming language representation.
The language AspectJ [12] can be considered as an aspect weaver for Java. This supports various constructs (e.g. "pointcuts" for before- and after-"advice", "introduction" for adding methods, wild cards which allow method names to be grouped, etc.) which allow programmers to define aspects as separate textual units that can then be applied to a Java class to produce a new class which contains the required aspect.
An approach which modifies Java programs at the source level inevitably has some disadvantages, especially with respect to aspects which have a wide general application, such as synchronisation, protection or monitoring. Here the ideal would be to define one or more aspects as completely independent modules without them needing to have knowledge of the classes which they might qualify, and then simply apply them to these classes (or even individual objects) as required. To take a simple example, an aspect such as "reader-writer synchronisation" might have an implementation which uses the following pattern (using the reader priority algorithm from [4]):

```
// the data structures
Semaphore mutex = new Semaphore(1);
Semaphore readerExclusion = new Semaphore(1);
int readcount = 0;
```

```
 // the writer synchronisation pattern
  mutex.p();
  --- call the writer method to be synchronised ---
  mutex.v();
 // the reader synchronisation pattern
  readerExclusion.p(); readcount++; if (readcount == 1) mutex.p();
  readerExclusion.v();
  --- call the reader method to be synchronised ---
  readerExclusion.p(); readcount--; if (readcount == 0) mutex.v();
  readerExclusion.v();
```

Java and other standard OOP languages have no mechanism allowing such a general pattern to be defined and implemented once as an independent unit and then applied to any class or object which needs to be synchronised. Consequently any aspect weaver which compiles to such target languages must face some substantial problems. Regardless of the technique used (e.g. simply modifying the source code of individual classes, defining subclasses which implement the aspect) each existing class must be considered separately. It must be clear which methods are readers, which are writers, how to handle public fields, how to handle static members, etc. An aspect of this kind can be written only if the target classes have been designed according to some special rules (e.g. writer methods begin with "set", reader methods with "get", there must be no public fields and no static members). And since using Java ultimately involves modifying the methods of the target class, the modification has a static character (e.g. *all* instances of the target class are reader-writer synchronised, etc.).

Thus it is not possible simply to define a general component and apply it (together with other similar components) in a straightforward manner to objects as they are dynamically created. Yet the requirement is both simple and relevant. The example of synchronisation is *not* an exceptional case. Another significant example is the need to control access to objects, e.g. by dynamically checking whether the client is listed on an access control list (ACL) or whether he can supply a password, whether the expiry date for a trial use of an object has expired, etc. Then there is the issue of monitoring, i.e. maintaining relevant information about access to objects, e.g. for debugging, for detecting hackers, etc. A more advanced application for such general purpose modules is a transaction mechanism.

The aim of *qualifying* types is to provide just such a mechanism, which allows programmers to define and implement general purpose components that can qualify the behaviour of objects without having a special knowledge of their interfaces nor their implementations. However, for reasons hinted at above, such a mechanism cannot be simply added to conventional OOP languages. To make it work, the language must include some special features. In the next section we briefly outline how these features are provided in the language Timor which is currently being designed at the University of Ulm [9, 10].

2 The Timor Language

Timor can be viewed as an object oriented programming language which, although based syntactically on Java and C++, breaks with some fundamental concepts of OOP in order to provide a better support for the idea of developing separate components (including, but not only, qualifying types) in such a way that these can be easily mixed and matched with each other to produce new application systems.

The first major difference from standard OOP is that the class concept is abandoned in favour of a separation of types and their implementations (which are not types), thus allowing a type to have multiple implementations. This feature of Timor is described in more detail in [9].
Types are defined according to the information hiding principle [18]. A key requirement for supporting qualifying types is that the programmer must designate the instance methods of any type either as `op` methods, i.e. operations which can modify the state of the instance, or as `enq` methods, i.e. enquiries which can access but not modify the state of an instance (cf. e.g. methods declared `const` in C++). Thus for synchronisation and other purposes (e.g. protection, transaction management) the instance methods of a type are automatically classified as readers or writers.
For programming convenience, a type definition can also include members resembling fields or references[1]. However, such public members are *abstract variables*, which formally correspond to a pair of instance methods, i.e. an `op` for setting a value or reference, and an `enq` for getting a value or reference. Hence client accesses to abstract variables can be treated from the viewpoint of synchronisation, etc. like other instance methods.
A type can have zero or more (named) constructors, introduced by the keyword `maker`[2]. It can also have methods introduced by the keyword `binary`. These are intended to allow multiple instances of the type, passed as parameters, to be manipulated (e.g. compared). Under normal circumstances binary methods can only access these instances via their instance methods. Other forms of static methods (and fields) are not supported in Timor, but the effects of these can be achieved in other ways. Here is an example of a type which will be used in later examples:

```
type Thing {
 Atype anAbstractVariable;
 Thing* anAbstractReference;
 op void doSomething(int x);
 op int doSomethingElse(int y);
 enq int getSomething();
 enq int getSomethingElse();
 binary boolean equal(Thing t1, t2);
}
```

Separating types and implementations leads to a separation of subtyping and code re-use (which includes, but is no longer limited to, subclassing), described in [9]. Timor supports both multiple type inheritance and multiple code re-use, as is partly illustrated in [10].

3 Qualifying Types and Bracket Routines

A qualifying type is a normal Timor type which has the additional property that its instances can be used to qualify the instances of other types. Qualification is a mechanism not found in the conventional OOP paradigm, although it has similarities to

1 A reference defines a *logical* relationship between objects. It is not a physical pointer. It cannot be directly manipulated, and indirection (i.e. references to references) is not supported.

2 Where appropriate the compiler adds a parameterless maker with the name `init`.

some techniques discussed in section 8. The basic idea is that when a client invokes an instance method of some target object, a special method of a qualifying type (known as a *bracket routine*) can be scheduled in its place. This can, but need not, use a special method name `body` to invoke the intended instance method of the target object. This notion is illustrated in Figure 1.

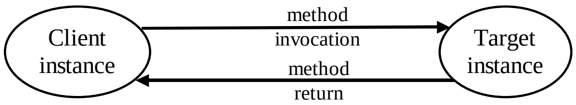

Fig. 1 (a): A Client invokes an Unqualified Target

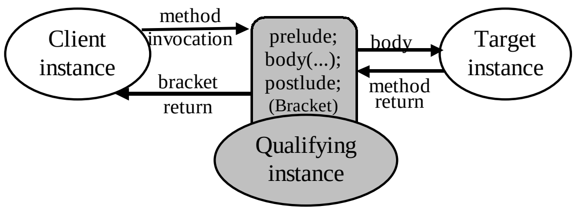

Fig. 1 (b): A Client invokes a Qualified Target

This property is reflected in the type definition by a `qualifies` clause, e.g.

```
type Mutex qualifies any
 augmenting {
  op bracket op(...); // a bracket routine for synchronising ops
  op bracket enq(...);// a bracket routine for synchronising enqs
 }
{/* in this example there are no "normal" instance methods, etc. */}
```

The keyword `any` indicates that instances of this type can qualify instances of *any* other type. The `augmenting` clause lists the bracket routines of the qualifying type[3]. In this case the bracket routines can qualify `op` and/or `enq` methods, as in the example. Bracket routines are themselves classified as `op` or `enq`, depending whether they modify or simply read the instance data of the *qualifying* instance. (This allows qualifying instances themselves to be qualified.)

Here is an implementation of the type `Mutex`:

```
impl Mutex1 of Mutex {
  Semaphore mutex = Semaphore.init(1);//a semphore is initialised to 1
  op bracket op(...) { // the code for handling ops (writers)
    mutex.p(); try {return body(...);} finally {mutex.v();}
  }
  op bracket enq(...) { // the code for handling enqs (readers)
    mutex.p(); try {return body(...);} finally{mutex.v();}
  }
}
```

The invocation `body(...)`indicates at what point in the code the target method invoked by the client (or in cases of multiple qualification, possibly a further bracket

3 The keyword `augmenting` indicates that the bracket routines unconditionally add to the behaviour of qualified methods, i.e. they add a prelude and/or a postlude and unconditionally invoke the target method. Alternative keywords `replacing` and `testing` indicate that the target method is not invoked at all, or that it is invoked conditionally (e.g. after testing some protection condition).

routine) is called. In general bracket routines (i.e. those which qualify `any`) the actual parameter list is not known and therefore cannot be modified in the bracket routine; the notation `(...)` indicates in this context that the parameters supplied by the client are passed on unchanged.

The `try`/`finally` construct is used in this example to ensure that if the target routine terminates abnormally the semaphore will nevertheless be released.

Creating a mutually exclusive `Thing` requires an instance of the type `Thing` and an instance of the type `Mutex`. The relationship between them can be set up as follows:

```
Mutex exclusive = Mutex.init();
Thing t = exclusive Thing.init();
```

In the second line a qualifying expression (here the variable `exclusive`) is associated with the creation of an instance of a (qualified) type. Hence in the above example the new `Thing t` is associated with an *existing* instance of a qualifying type.

There is an alternative way of associating a `Mutex` instance with a `Thing` instance:

```
Thing t = Mutex.init() Thing.init();
```

In this case the qualifying expression returns a new (anonymous) instance of `Mutex`. In either case, the client of `t` accesses it as if it were *not* qualified, e.g.

```
t.doSomething();
```

Indeed he might not know that it is qualified (e.g. if it receives this as a parameter of type `Thing`). In the case of synchronisation the anonymous form is often useful. However, separating the declarations of instances of `Mutex` and `Thing` allows several objects to be qualified by a single instance of a qualifying type, e.g.

```
Thing t1 = exclusive Thing.init();
Thing t2 = exclusive Thing.init();
```

In this case both `t1` and `t2` are synchronised using the same semaphore instance, and in fact even objects of different types could be synchronised together, e.g.

```
Thing t1 = exclusive Thing.init();
AnotherThing t2 = exclusive AnotherThing.init();
```

One effect of creating an *anonymous* instance is that its own instance methods cannot be explicitly invoked. This makes sense for synchronisation, which needs no explicit methods. However, most qualifying types need such explicit methods. For example, the bracket routines of a qualifying type might check that clients are listed in an ACL. Such types need explicit methods for maintaining entries in the ACL. These normal methods must be invocable via an explicit variable. The absence of explicit methods in synchronising types is the exception rather than the rule.

When a client invokes an operation of `t` (i.e. by calling `doSomething` or `doSomethingElse`, or by modifying the abstract variable `anAbstractVariable` or `anAbstractReference`) the `op` bracket of the instance `exclusive` is scheduled. Similarly the invocation of `getSomething` or `getSomethingElse` or the reading of `anAbstractVariable` or `anAbstractReference` cause the `enq` bracket to be scheduled.

If the `equal` method of `Thing` is invoked, no bracket routine intervenes directly. However, when its implementation accesses the instance methods of its parameters, these are bracketed as appropriate.

4 Reader-Writer Synchronisation

Distinguishing between `op` and `enq` methods facilitates the development of components which provide reader-writer synchronisation in a general way. Here is a type definition and an implementation based on the reader priority algorithm first published by Courtois, Heymans and Parnas [4], cf. section 1:

```
type RWsync qualifies any
augmenting {
  op bracket op(...); // brackets ops
  op bracket enq(...); // brackets enqs
} {/* no "normal" methods */}
impl Curtois of RWsync
reuses Mutex1 {
  Semaphore readerExclusion = Semaphore.init(1);
  int readcount = 0;
  op bracket enq(...) { //reader synchronisation
    readerExclusion.p(); readcount++;
    if (readcount == 1) ^Mutex1.mutex.p();
    readerExclusion.v();
    try {return body(...);}
    finally {
      readerExclusion.p(); readcount--;
      if (readcount == 0) ^Mutex1.mutex.v();
      readerExclusion.v();
    }
  }
}
```

In this implementation the `op` bracket routine and the mutual exclusion semaphore `mutex` are re-used from `Mutex1`[4]. (Code re-use in Timor is described in [9, 10].)

5 Bounded Buffer Synchronisation

We now consider how a bounded buffer might be synchronised using bracket routines. First we define a basic unsynchronised type:

```
type BoundedBuffer {
  maker init(int maxSize);
  op void produce(ELEM e);
  op ELEM consume();
}
```

The type `ELEM` can be thought of as any relevant type, here representing the type of the elements in the buffer. Timor supports a generic mechanism along the lines described in [6], but this is not directly relevant to our discussion and is not described here. The following is a simple array implementation:

```
impl BB of BoundedBuffer {
  ELEM[] buffer;
  int nextFull = 0, nextEmpty = 0;
  int bufferSize;
  op void produce(ELEM e) {
    buffer[nextEmpty] = e; nextEmpty++; nextEmpty %= bufferSize;
  }
  op ELEM consume() {
    ELEM temp = buffer[nextFull]; nextFull++; nextFull %= bufferSize;
    return temp;
  }
```

[4] As multiple implementations can be re-used the hat symbol indicates which implementation is actually being re-used in a `super`-like context.

```
   maker init(int maxSize) {
     bufferSize = maxSize; buffer = ELEM[].init(maxSize);
   }
 }
```
There are several possibilities for synchronising a bounded buffer, depending on the number of producer and the number of consumer processes. In the simplest case the basic type can be used in a sequential program which does not require synchronisation (though the program logic must then be designed to ensure that overflow and underflow of the buffer do not occur).

Now consider the case of a single producer and a single consumer process. Here the two instance methods require different synchronisation protocols. But since both are `op` methods, the technique described so far is inadequate. Instead we can use a *specialised* qualifying type, i.e. a type designed to qualify some specifically named type(s) rather than *any* type. Here is an example:
```
 type SyncBB qualifies BoundedBuffer
 augmenting {
   op void produce(ELEM e);
   op ELEM consume();
 }
 { maker init(int maxSize);
 }
```
Here the `qualifies` clause nominates a specific type and specifies which of its methods are to be qualified. In this example a maker of the qualifying type needs to be explicitly parameterised, as we see from the following implementation code:
```
 impl SyncBB1 of SyncBB {
   Semaphore full = Semaphore.init(0);
   Semaphore empty;
   maker init(int maxSize) {
     empty = Semaphore.init(maxSize);
   }
   op void produce(ELEM e) {
     empty.p(); try {body(...);} finally {full.v();}
   }
   op ELEM consume() {
     full.p(); try {return body(...);} finally {empty.v();}
   }
 }
```
In this example the `body` statement in the two bracket routines uses the parameter form `(...)` to indicate that the actual parameters are not modified, although this is possible in the case of specialised bracket routines. (It would be correct for example to formulate the `body` statement in the `produce` bracket as `body(e)`.)

Given an initialised integer `maxSize`, which defines the maximum number of entries in the buffer, an instance of `SyncBB` designed to qualify a buffer can be instantiated as:
```
 SyncBB synchronised = SyncBB.init(maxSize);
```
An instance of `BoundedBuffer` can be qualified as follows by `synchronised`:
```
 BoundedBuffer bb = synchronised BoundedBuffer.init(maxSize);
```
The instance `bb` is now adequately synchronised provided that it is accessed only by a single producer process and a single consumer process.

However, if a buffer can be accessed in parallel by multiple producers, these must exclude each other (though not a consumer process or processes). An inefficient way of achieving this is to associate an instance of `Mutex` with it, e.g.

```
Mutex exclusive = Mutex.init();
BoundedBuffer bb = synchronised, exclusive
                                   BoundedBuffer.init(maxSize);
```

If an instance is qualified by more than one qualifying instance, Timor defines that the order of applying the bracket routines is left to right. Thus in this example when the `produce` or the `consume` method is invoked, the relevant bracket routine of `synchronised` is executed first, and when it executes the `body` statement this results in the `op` bracket routine of `exclusive` being invoked. Then when the latter executes the `body` statement the relevant method of `bb` is invoked. In this example the order of the qualifying types is significant. Reversing this order leads to a deadlock if a producer attempts to access a full buffer or a consumer attempts to access an empty buffer.

Using mutual exclusion with a bounded buffer is inefficient, because producers need only exclude other producers and consumers other consumers, as the `SyncBB` type takes care of mutual interference between the two groups as groups.

To handle the case of multiple producers another specialised type can be defined:

```
type MultiProducer qualifies BoundedBuffer
augmenting {
  op void produce(ELEM e);
}
{ /* no normal methods */ }
impl ProdMutex of MultiProducer {
  Semaphore mutex = Semaphore.init(1);
  op void produce(ELEM e) {
    mutex.p(); try{body(...);} finally {mutex.v();}
  }
 }
```

This is a simple variant of `Mutex` which is defined as a specialised type, thus allowing the bracket code to be applied specifically to the `produce` method without affecting the `consume` method. (`Mutex` cannot discriminate between these two `op` methods.) An instance can be initialised as follows:

```
MultiProducer multiProducer = MultiProducer.init();
```

Similar considerations apply to multiple parallel consumers, leading to the definitions:

```
type MultiConsumer qualifies BoundedBuffer
augmenting {
  op ELEM consume();
}
{ /* no normal methods */ }
impl ConsMutex of MultiConsumer{
  Semaphore mutex = Semaphore.init(1);
  op ELEM consume() {
    mutex.p(); try{return body(...);} finally {mutex.v();}
  }
}
```

and an instantiation

```
MultiConsumer multiConsumer = MultiConsumer.init();
```

Given these additional components actual buffers can be declared to suit any synchronising case, as follows:

(a) One consumer process, several producers:

```
BoundedBuffer bb = synchronised, multiProducer
                                   BoundedBuffer.init(maxSize);
```

(b) One producer process, several consumers:

```
BoundedBuffer bb = synchronised, multiConsumer
                                   BoundedBuffer.init(maxSize);
```

(c) Several producers, several consumers:

```
 BoundedBuffer bb = synchronised, multiProducer, multiConsumer
                                   BoundedBuffer.init(maxSize);
```

6 Specialised Qualifying Types vs. Subtyping

Inheritance is a hallmark of traditional OO languages. This is not a reasonable alternative to general qualifying types for achieving such aims as mutual exclusion or reader-writer synchronisation, because the use of subtyping to achieve such aims implies that special code has to be added to each type to be qualified. However, it is interesting to compare subtyping with specialised qualifying types.

In any OO programming language a class corresponding to the type `BoundedBuffer` and its implementation `BB` can have a subclass which extends it to add the functionality of `SyncBB` and its implementation `SyncBB1`, by overriding the methods `produce` and `consume` and from within the overriding methods using `super` to invoke the original methods at the point where the bracket routines of `SyncBB1` invoke the `body` statement. We call this subclass `SyncBBderived`. It can correctly implement the case of a single producer and a single consumer.

We can now apply the same technique to mimic the effect of `MultiProducer`, i.e. by extending `SyncBBderived` with a subclass `MultiProducerDerived`, which overwrites the method `produce` to add mutual exclusion.

Since multiple consumers are handled orthogonally to multiple producers they might be provided for in a further subclass of `SyncBBderived`, i.e. `MultiConsumerDerived`, which overwrites the method `consume` to add mutual exclusion.

The case of both multiple producers and multiple consumers can be handled in a new class `MultiProducerConsumerDerived` which inherits from both `MultiProducerDerived` and `MultiConsumerDerived` without adding new methods. This is illustrated in Figure 2:

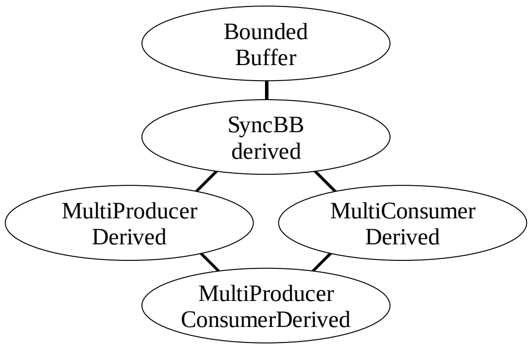

Fig. 2: A Subclass Hierarchy

Since this ideally requires multiple implementation inheritance, languages such as Java cannot handle the situation cleanly, so that a language which supports qualifying types might be regarded as superior to these. However, there are of course OO languages which do handle multiple implementation inheritance, including Timor.

This hierarchy is in fact typical of the kind of multiple inheritance which arises when orthogonal properties (here multiple producers and multiple consumers) are com-

bined, as is discussed in more detail in [10], and a Timor solution can follow the same pattern as is outlined there for the example of the Timor Collection Library.
Although a solution based on qualifying types can in this case be handled via subtyping, not all solutions can easily achieve this. In fact one might consider it almost a matter of luck that the above solution is correct, because it actually results in the producer and consumer mutual exclusions being applied *before* the standard buffer synchronisation (from `SyncBBderived`) is applied. Fortunately this does not lead to a deadlock. However, using subtyping in the same way to mimic the first solution presented (based on the use of a single `mutex` semaphore to synchronise both producers and consumers, see Figure 3) would result in an incorrect solution which contains the risk of deadlocks.

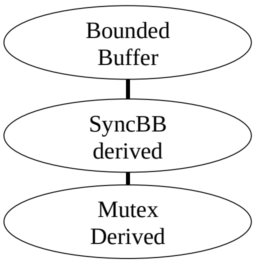

Fig. 3: A Subclass Hierarchy with Full Mutual Exclusion

Changing the order in the hierarchy (i.e. placing `MutexDerived` above `SyncBB-derived,` see Figure 4) can solve that problem, but then it leads to a further problem: instances of `MutexDerived` would not synchronise correctly. That can be avoided by defining it as an abstract class. But there remains a further problem. We now no longer have a class which simply synchronises a single producer and a single consumer without the overkill of mutual exclusion.

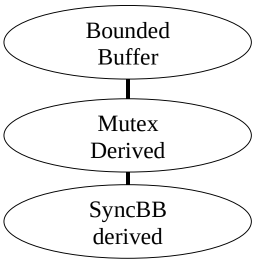

Fig. 4: A Rearranged Subclass Hierarchy

We see from this example that subtyping is considerably less flexible in some circumstances than qualifying types, because using it to mimic bracket routines with calls to overridden methods determines the order of the "bracket routines", and this order may not correspond to the logic required by the problem being solved. Furthermore, in contrast with qualifying types in Timor (which do not result in new subtype relationships), the use of subtyping to mimic bracket routines also creates new type relationships which may not be appropriate [14].
Two further points are also relevant. Synchronisation, as was noted above, is exceptional from the viewpoint of qualifying types in that it does not require its own explicit methods. If qualifying types are mimicked by subtyping they must add any additional methods to those which appear in the supertype. While this is possible, it

may not be desirable. For example if a qualifying type implements protection by means of an access control list (and therefore has methods for adding entries to and removing them from this list) it would be inappropriate to add these methods to instances being protected! Finally the flexibility which allows an instance of a qualifying type to qualify more than one qualified instance cannot be simulated straightforwardly by means of subtyping.

7 Synchronising Views

The use of specialised qualifying types has so far been illustrated by defining types which qualify instances of individual *types*. However, the real power of specialised bracket routines comes into play when they are used to qualify `view` interfaces[5] which might be included in *many* types. For example one might define a view `Openable` along the following lines:

```
view Openable{
 const int CLOSED = 0;
 const int READ = 1;
 const int WRITE = 2;
 op void open(int openMode) throws OpenError;
 op void close();
 enq int currentOpenMode();
}
```

which could appropriately be included in many type definitions. On the basis of such a view it is then possible to define a specialised qualifying type, e.g.:

```
type OpenSynchronised qualifies Openable
augmenting {
  op void open(int openMode);
  op void close();
}
{/* as usual with synchronisation, there are no explicit methods */}
```

with an implementation which uses the parameter `openMode` to determine when to apply the reader and when the writer synchronisation protocol. (In the close bracket it can access the enquiry `currentOpenMode` for the same purpose.)

Objects of types extending `Openable` can be instantiated with `OpenSynchronised`, e.g.

```
type OpenableThing extends Thing, Openable;
OpenSynchronised openSync = OpenSynchronised.init();
OpenableThing ot = openSync OpenableThing.init();
```

To attempt to use conventional subtyping to achieve the bracketing in such a case would require that each type to be synchronised (e.g. type `OpenableThing`) would have to be individually implemented with the synchronisation code. Hence it makes sense to support specialised qualifying types in addition to conventional inheritance.

5 A view interface defines methods which can be included in different types. It can have an implementation but no maker. It cannot be independently instantiated.

8 Related Work

This paper has illustrated by example how one aspect commonly encountered in programming situations, viz. synchronisation, can be handled using qualifying types with bracket routines. The basic idea is based on earlier work in our group, beginning with the concept of attribute types and bracket routines, cf. [8].

The idea that code can be added to existing procedures is by no means new, and dates back at least to Pascal-Plus [21]. A form of bracketing is possible in almost all object oriented languages by redefining the methods in a subclass and calling the original methods from within the redefined methods via a `super` construct. So, for example, a class `RWsyncThing` can be defined as a subclass of `Thing`. But in languages which support only single inheritance, a subtype `RWsyncBook` of `Book` must include all the same additional code as `RWsyncThing`.

In languages such as Eiffel [15] with multiple inheritance, a class `RWsync` can be defined and inherited by both `RWsyncThing` and `RWsyncBook`. This means that the type `RWsync` is only declared in a single place. The bracketing must, however, still be achieved via redefinition in both `RWsyncThing` and `RWsyncBook`.

When the `inner` construct of Beta [13] (cf. `body`) appears in a superclass method, the same method in a subclass is bracketed by the code of the superclass method. But a Beta superclass `RWsync` would need to know exactly which methods occur in its subclass `RWsyncThing` in order to bracket them and would therefore be of no use in bracketing `RWsyncBook`.

Mixins are a generalization of both the `super` and the `inner` constructs. The language CLOS [5] allows mixins as a programming technique without supporting them as a special language construct, but a modification of Modula-3 to support mixins explicitly has also been proposed [3]. A mixin is a class-like modifier which can operate on a class to produce a subclass in a manner similar to that of qualifying types. So, for example, a mixin `RWsync` can be combined with a class `Thing` to create a new class `RWsyncThing`. Bracketing can be achieved by using the 'call-next-method' statement (or `super` in the Modula-3 proposal) in the code of the mixin methods. As with Beta, however, the names of the methods to be bracketed must be known in the mixin. This again prevents it from being used as a general component.

In [19] encapsulators are described as a novel paradigm for Smalltalk-80 programming. The aim is to define general encapsulating objects (such as a monitor) which can provide pre- and post-actions when a method of the encapsulated object is invoked. This is similar to bracket routines but is based on the assumption that the encapsulator can trap any message it receives at run-time and pass this on to the encapsulated object. This is feasible only for a dynamically typed system. The mechanism illustrated in this paper can be seen as a way of achieving the same result in a statically type-safe way via a limited form of multiple inheritance. The applications of encapsulators are also more limited than bracket routines since there is no way for them to distinguish between reader and writer methods.

Specialised qualifying types can be simulated using Java proxies, but the programming is considerably more cumbersome, and methods to be bracketed cannot be isolated from those not requiring brackets. Thus all method calls to a target object must be redirected to the proxy. In a case such as `Openable`, where the `open` and `close` methods need be called only once, between which many other method invocations can occur, this can be very inefficient. Even when methods require bracketing the ap-

proach is inefficient: the proxy object and an associated handler must both be invoked, and reflection used to establish which target methods have been invoked. Multiple qualification of a target method is particularly complicated and inefficient.
Composition filters [2] allow methods of a class to be explicitly dispatched to internal and external objects. In addition the message associated with a method call can be made available via a *meta* filter to an internal or external object, thus allowing the equivalent of a bracket routine to be called. However, because filters are defined in the "target" class, a dynamic association of filters with classes is not possible, and all the objects of a class are qualified in the same way.
MetaCombiners support the dynamic addition/removal of mixin-like *adjustments* for individual objects [16]. The effect of specialised qualifying types can be achieved with *specialisation adjustments* (which can invoke `super`) on an individual object basis. Similarly field acquisition and field overriding [17] can be used to simulate inheritance of field methods and therefore in conjunction with the keyword `field` (cf. `super`) can simulate the use of `body` in bracket routines. In both cases there appears to be no equivalent to general bracket routines.
The experimental language Piccola [1] is a component composition language which allows abstractions not well supported by the OO paradigm (such as synchronisation) to be integrated into applications. While it has similar aims, it differs from the Timor approach, where qualifying types are integrated into the base language and therefore need no special composition language.
The AOP language AspectJ [12] and similar languages (cf. [20]) can achieve many of the aims of qualifying types, but with a number of limitations:
- Because Java has no way of distinguishing between `op` and `enq` methods, some convention for method names must be used (e.g. methods beginning with `set` are writers, those with `get` are readers). For target classes not developed according to the convention each class must be examined individually and a separate aspect developed for it.
- Because they operate at the source level an aspect affects the target class, so that different objects of the same class cannot be qualified in different ways.
- Because aspects are not separately instantiated an aspect "instance" cannot be flexibly associated with a group of objects rather than a single object.
- New methods explicitly defined with an aspect ("introduction") become methods of the qualified objects. Thus methods defined, for example, to manipulate an ACL in a protection aspect, become methods of the objects being protected, so that a protected object includes the methods which control its protection!
- Because the order of the execution of AspectJ advice is statically defined in aspects, these must be defined with a knowledge of each other, except in cases where precedence is considered to be irrelevant. In contrast the execution order of Timor bracket routines is easily defined at the time a target object is created.
- In contrast with AspectJ aspects, general qualifying types and specialised types based on view interfaces (e.g. `Openable`) do not depend on a knowledge of (or the presence at compile time of) each other's source code or that of types which they might qualify.

9 Conclusion

The paper has illustrated the use of Timor qualifying types, using synchronisation as an example. Inevitably not all features of this new concept have been described in

full. Future papers will discuss how qualifying types are defined, how they relate to the type system and how they behave (for example when nested) at run-time.
The comparison with other work indicates that qualifying types provide a powerful new mechanism for supporting general aspects of programming, such as synchronisation, protection and monitoring. It is particularly advantageous that they can be separately implemented as components which can be applied in many cases to any type, or in more specialised cases to any type which supports a particular view interface.

Acknowledgements

Special thanks are due to Dr. Mark Evered and Dr. Axel Schmolitzky for their invaluable contributions to discussions of Timor and its predecessor projects. Without their ideas and comments Timor would not have been possible.

References

[1] F. Achermann and O. Nierstrasz, "Applications = Components + Scripts - A Tour of Piccola," in *Software Architectures and Component Technology*, M. Aksit, Ed.: Kluwer, 2001, pp. 261-292.

[2] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns Using Composition Filters," *Communications of the ACM*, vol. 44, no. 10, pp. 51-57, 2001.

[3] G. Bracha and W. R. Cook, "Mixin-based Inheritance," ECOOP/OOPSLA '90, Ottawa, Canada, 1990, ACM SIGPLAN Notices, vol. 25, no. 10, pp. 303-311.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667-668, 1971.

[5] L. G. DeMichiel and R. P. Gabriel, "The Common Lisp Object System: An Overview," ECOOP '87, Paris, 1987, Springer-Verlag, LNCS, vol. 276, pp. 151-170.

[6] M. Evered, J. L. Keedy, G. Menger, and A. Schmolitzky, "Genja - A New Proposal for Genericity in Java," 25th International Conf. on Technology of Object-Oriented Languages and Systems, Melbourne, 1997, pp. 169-178.

[7] D. Holmes, J. Noble, and J. Potter, "Aspects of Synchronisation," 25th International Conference on Technology of Object-Oriented Languages and Systems, Melbourne, 1997, pp. 7-18.

[8] J. L. Keedy, K. Espenlaub, G. Menger, A. Schmolitzky, and M. Evered, "Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes," 6th International Conference on Software Reuse, Vienna, 2000, pp. 420-435.

[9] 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.

[10] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," *Journal of Object Technology (www.jot.fm)*, vol. 1, no. 1, pp. 81-106, 2002.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," ECOOP '97, 1997, pp. 220-242.

[12] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP 2001 - Object-Oriented Programming, 2001, Springer Verlag, LNCS, vol. 2072, pp. 327-353.

[13] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard, "The Beta Programming Language," in *Research Directions in Object-Oriented Programming*: MIT Press, 1987, pp. 7-48.

[14] 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.

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

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

[17] K. Ostermann and M. Mezini, "Object-Oriented Composition Untangled," OOPSLA '01, Tampa, Florida, 2001, ACM SIGPLAN Notices, vol. 36, no. 11, pp. 283-299.

[18] 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.

[19] G. A. Pascoe, "Encapsulators: A New Software Paradigm in Smalltalk-80," OOPSLA '86, 1986, pp. 341-346.

[20] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," 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. 53 - 60.

[21] J. Welsh and D. W. Bustard, "Pascal-Plus - Another Language for Modular Multiprogramming," *Software-Practice and Experience*9, pp. 947-957, 1979.