JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 1, January-February 2004

Qualifying Types with Bracket Methods in Timor

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

Abstract

A new kind of type is described whose objects ("qualifiers") have bracket methods which can modify the run-time behaviour of other objects ("targets"). Bracket methods can qualify either specific methods of a target or can separately qualify their reader and writer methods, thus allowing general qualifiers to be developed for standard activities such as synchronisation, monitoring and protection. Qualifiers are associated with a target when it is created, in the form of a qualifier list. Individual qualifiers can be dynamically added to and removed from the list even while the target object is active.

1 INTRODUCTION

Software often involves the programming of "aspects" which are not only orthogonal to each other, but which sometimes have a quite general character and a wide area of application (e.g. synchronisation, monitoring). As proponents of aspect oriented programming have emphasised [13], it is not only important to be able separately to program such aspects, but also to have mechanisms ("aspect weavers") which allow the aspects to be woven together into the compound units actually needed for application systems.

The present authors consider that such a weaving mechanism can best be provided as a construct within (new) programming languages. In order to handle at least some of the aspects of software design which can arise, we present in this paper a novel programming language feature, which we call *qualifying types*. These allow software units to be defined which can qualify the methods of other software units in a general way, using *bracket methods* in their implementations. They are useful for programming a wide variety of general purpose properties, e.g. synchronisation, monitoring, logging, protection and transaction control. Preliminary versions of this idea have been presented in [7, 8] and the technique has been illustrated with respect to synchronisation in [12].

Cite this article as follows:J. Leslie Keedy et al: “Qualifying Types with Bracket Methods in Timor”, in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 101-121.
http://www.jot.fm/issues/issue_2004_01/article1

In the present paper we describe how qualifying types are integrated into Timor[1], a programming language which is currently being developed at the University of Ulm in Germany. Section 2 briefly outlines the basic concepts of Timor which are relevant for an understanding of the paper. Section 3 explains the basic idea of qualifying types and section 4 describes how they are defined and implemented using bracket methods. Section 5 describes how qualifying objects (qualifiers) are associated with objects of the types which they qualify (targets) and section 6 briefly addresses some flow of control issues. Section 7 describes when bracket methods are applied. Section 8 briefly considers issues relating to type compatibility and section 9 compares the technique with subtyping/subclassing. The paper concludes with a discussion of related work in section 10 and some concluding remarks in section 11.

2 A QUICK TOUR OF RELEVANT TIMOR CONCEPTS

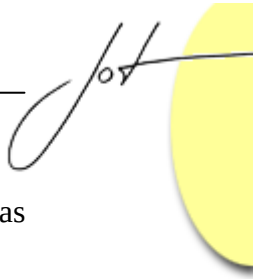
Timor has been designed primarily as a language for supporting the development of software components. Wherever feasible, Java and C++ have been used as its basic models, but it is structurally a quite different language. One of its aims is to support a components industry which develops general purpose software for re-use in many different application systems. This aim has fundamentally influenced the main features of the language.

Timor has abandoned the traditional OO class construct by decoupling interfaces and their implementations. Components designed at an abstract level can often have quite different implementations (e.g. a type `Queue` can be implemented as a linked list, as an array, etc.) and a component designer may well wish to produce and distribute several different implementations for the same type. Consequently there is a rigorous distinction between an *interface* and its potentially multiple *implementations*. Interfaces and implementations are formulated according to the information hiding principle [21]. An interface is either a *type* (which may be concrete or abstract) or a *view*.

Concrete types are units from which multiple instances can be constructed. Constructors, known in Timor as *makers*, have individual names and are listed in a section introduced by the keyword `maker`. A maker returns an instance of its own type. It may have parameters of its own type (e.g. to construct a new instance by concatenating or merging values from existing instances). If a maker is not explicitly defined in a concrete type the compiler automatically adds a parameterless maker with the name `init`.

The instances of a type are manipulated by methods defined in an `instance` section. Instance methods must be designated by the keyword `op` (i.e. operations which can change state variables of the instance) or the keyword `enq` (enquiries which can read but not change state variables). *Abstract variables* can be defined in the instance section.

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



These have the appearance of value or reference[2] variables but are formally defined as pairs of methods (an `op` for setting and an `enq` for getting the value of the "field") [11].

It is well known that binary methods are problematic [4]. Timor type definitions can have a method category `binary`, which defines binary *type* methods, i.e. methods that access multiple instances of the type (e.g. to compare them). A compile time error occurs if a programmer attempts to define an instance method which has a parameter (value or reference) or local variable of its own type or a supertype thereof. As we shall see later, such public binary instance methods would create substantial difficulties for a clean design of qualifying types.

Abstract types are intended to be abstractions of "complete" types (e.g. a collection as an abstraction for a set, a bag, a list, etc.) and although they do not have real makers, they can predefine makers and binary methods which are inherited as real methods in concrete types derived from them [10].

A *view* is an interface which defines a related set of instance methods and/or abstract variables that can usefully be incorporated into different types. It is intended to encourage the idea of "programming to interfaces". A view can have implementations, but it cannot define makers or binary methods.

In Timor a type can be derived from other types by *extension* and/or *inclusion*. The resulting units are known as *derived types* and those from which they are derived are their *base types*. The keyword `extends` defines a subtyping relationship and is intended to be used to signal *behavioural subtyping* (in the sense of Liskov and Wing [17]), though this cannot be checked by a compiler. Instances of a type derived by extension can be assigned to variables of the supertype. The keyword `includes` allows an interface to be inherited without implying a subtyping relationship. In this case the derived unit cannot be used polymorphically as if it were an instance of the base unit [9]. Timor provides several techniques for supporting multiple type inheritance, addressing different kinds of problems to be modelled. From the viewpoint of the present paper these need not be described further, since the end effect of each technique is that a derived type is defined in terms of instance methods, binary methods and makers, as above. Instance methods are what determines how a type can be qualified, as we shall shortly see.

Subtyping is decoupled from code inheritance. An interface (type or view) can have multiple implementations, which must be *behaviourally equivalent*, in the sense that each fulfils the interface's specification. An implementation, whether for a base or a derived type, can be a completely new implementation, i.e. code is not automatically inherited. But it can optionally re-use the code of implementations of other related and/or unrelated interfaces, by defining re-use variables in the `state` section of an implementation. These are variables defined either in terms of interfaces (without specifying a particular implementation) or of individual implementations, and are demarcated by a hat symbol. The compiler compares those public methods of the type being implemented which do not appear in the `instance` section of the implementation with the public methods of the re-use variables and when a match occurs this is treated as the implementation for that

[2] References are not directly related to physical addresses. They are logical references to objects.

method. When a re-use variable is defined in terms of an interface *any* of its implementations can be re-used, but the re-using implementation can access this only via its public members. When individual implementations are nominated the re-using implementation can gain access to the internal state of the re-used implementations (and therefore can – but need not – simulate conventional subclassing). An earlier version of this mechanism is described in [9]. Again, the important point here is that an implementation consists of instance methods, binary methods and makers, and these determine how a type can be qualified.

Types can be instantiated either as separate objects or as values which can be regarded as components of an object. Objects are always accessed via references. An object can contain references to other objects of the same or different types.

3 QUALIFYING TYPES: AN OVERVIEW

A qualifying type is a type whose objects can qualify the behaviour of objects of other types. In the normal OO paradigm an object of one type (the client) can invoke methods of an object of any other type (the target), as is trivially illustrated in Figure 1.

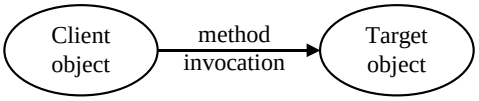


Figure 1: A Normal Method Invocation

An object of a qualifying type "interferes with" this in that it "catches" the invocation and executes the code of the appropriate bracket method (see Figure 2).

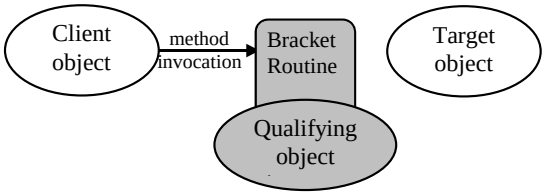


Figure 2: A Qualifying Type with a Bracket Method

At this point the bracket method can do one of three things.

First, the bracket method can augment the code of the target method by adding a prelude and/or a postlude. On completing the prelude it executes a `body` statement to indicate the point where the target method is to be invoked (cf. Figure 3). This technique can be used e.g. to add synchronising or logging operations in the prelude and postlude.

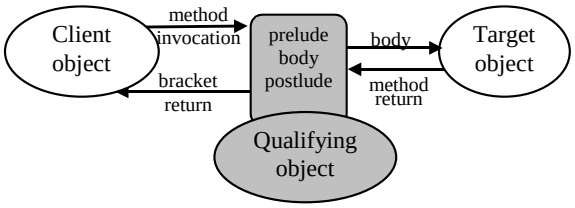


Figure 3: An Augmenting Bracket Method

Because `body` is a normal statement, it can be executed conditionally. Hence a bracket method might allow the target object to be invoked only if some test is passed (cf. Figure 4). This allows qualifying types to implement protection mechanisms. Thus the state of the qualifier might include an access control list used to carry out the test, or the test may involve demanding a password from the user before allowing access to the target object.

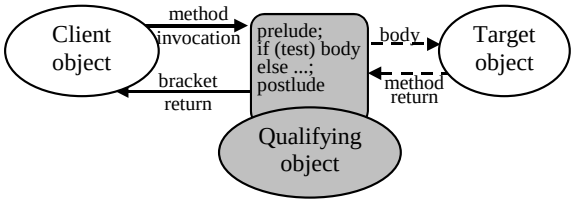


Figure 4: A Testing Bracket Method

In the simplest case the target method need not be invoked at all. Control is returned to the client on completion of the execution of the bracket method, without any warning as such that this has occurred (see Figure 5). This might for example be used as a disinformation technique, e.g. to fool a hacker into thinking that he can access some object when in fact its owner has substituted a decoy. Or it might be used to test client software before a working target is available.

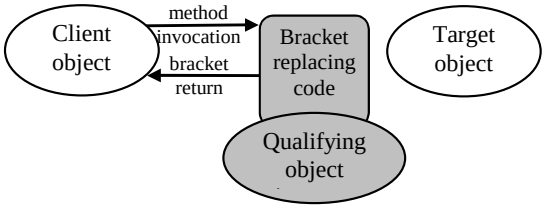


Figure 5: A Replacing Bracket Method

As the above examples make clear, the qualifying object is a normal object of its own type and it has its own state and methods, which are quite separate from the states and methods of the client and target objects. Its only special property is that its implementation includes bracket methods, which can execute `body` statements. These, like the other methods of the qualifying object, have access to the state of the qualifying object. But they do not have access to the state of the client object or target object (unless they possess a reference which provides such access).

4 DEFINING AND IMPLEMENTING QUALIFYING TYPES

A qualifying type is characterised by the appearance of one or more *qualifies clauses* in its definition. Each of these nominates an interface which can be qualified and provides a list of bracket methods which can perform this qualification. Like other interfaces, these can have multiple implementations.

Qualifying Objects of Any Type

Bracket methods which can qualify objects of *any* type are introduced by the qualifying clause `qualifies any`. In this case the type can be defined either to have a single bracket method which qualifies `all` the methods of target objects, or up to two bracket methods can be listed, one which can qualify `op` methods and one which can qualify `enq` methods (see section 2). Here is a simple example of a type definition defined to provide mutual exclusion synchronisation for target objects:

```
type Mutex {
qualifies any:
 op bracket all(...); // this provides mutual exclusion
 // and brackets all the instance methods of a target object
}
```

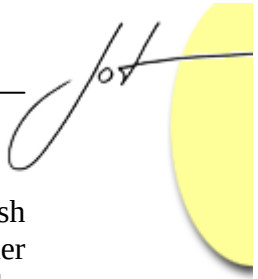
Although the bracket method declaration in this examples looks special, it is like a "normal" instance method declaration except that it uses special syntax. The modifier in the first position is the normal `op`/`enq` modifier, indicating whether the bracket method itself is an operation or an enquiry with respect to its own state. In this example the bracket method is designated as an `op`, because it changes its state (e.g. by modifying a semaphore). The requirement to designate bracket methods as `op` or `enq` methods is important, because it allows qualifiers themselves to be qualified.

The keyword `bracket` replaces the usual return type, indicating that the return value has the type of whatever method is being bracketed. The method "name" `all` signifies that the method is applied to all instance (and bracket) methods of the target object. Finally a parameter list containing the ellipsis character is used to indicate that any parameters defined for a target method can be accepted, and that these are not accessed in the bracket method's implementation.

The type `Mutex` can be implemented using semaphores as follows:

```
impl MutexSemImpl of Mutex {
state:                          // introduces state variables
 Semaphore mutex = Semaphore.init(1);
qualifies any:                  // implements the bracket method
 op bracket all(...) {
  mutex.p();
  try {return body(...);}
  finally {mutex.v();}
 }
}
```

The `body` statement indicates the point in the code at which the method of the target object invoked by the client is actually called. A `body` statement is a "normal" statement, except that it can only appear in an implementation of a bracket method. The use of the ellipsis in the argument list indicates that the parameters which were originally provided are passed on without change. A `return` statement is used, indicating that the value which `body` returns (which may be `void`) is in turn passed back.



A qualifying type for providing reader-writer synchronisation can distinguish between `op` instance methods (i.e. writer methods) and `enq` instance methods (i.e. reader methods), without requiring the designer of the qualifying type to know details of the individual methods of the objects to be qualified. Here is a definition:

```
type RWsync {       // provides reader writer synchronisation
qualifies any:      // for any type
 op bracket op(...);  // brackets op methods (writers)
 op bracket enq(...); // brackets enq methods (readers)
}
```

and the reader priority implementation of Courtois, Heymans and Parnas [5]:

```
impl CourtoisEtAl of RWsync {
state:
 Semaphore mutex = Semaphore.init(1);
 Semaphore readers = Semaphore.init(1);
 int readcount = 0;
qualifies any:   // provides reader-writer synchronisation
 op bracket op(...) {  // the writer protocol
  mutex.p();
  try {return body(...);}
  finally {mutex.v();)
 }
 op bracket enq(...) { // the reader protocol
  readers.p(); readcount++;
  if (readcount == 1) mutex.p();
  readers.v();
  try {return body(...);}
  finally {
   readers.p(); readcount--;
   if (readcount == 0) mutex.v();
   readers.v();
  }
 }
}
```

In this example the method "names" `op` and `enq` signify which bracket method is applied to operations and which to enquiries of the target object.

Both `Mutex` and `RWsync` are unusual in that they have no instance methods of their own. Most qualifying types need instance methods independently of their bracket methods. These typically provide information which governs the decisions taken in bracket methods and/or allow information to be recovered which has been acquired by bracket methods. Sometimes the instance methods form a consistent type without the bracket methods, in which case it can be more modular to define a separate type and then extend this to a type with bracket methods as in the following example of an access control list (ACL) [16]. The ACL is held in the state variables and is maintained via its instance methods. Subjects are threads which have unique integer identifiers.

```
seq AccessMode {NOACCESS, READ, WRITE}
// a seq is an enumeration with values in ascending order
type ACL {
```

```
maker:
 ThisType init(int maxId); // highest thread id in the ACL
 // the keyword ThisType indicates that derived types also
 // automatically have a maker with the same definition
instance:  // these are normal instance methods
 op void addThread(int threadId; AccessMode access)
  throws InvalidThreadId;
  // adds access information for a thread to the ACL
 op void removeThread(int threadId) throws InvalidThreadId;
  // logically removes a thread from the ACL
 enq AccessMode currentAccess(int threadId)
  throws InvalidThreadId;
  // returns the current access of the thread with this id
 enq int maxId(); // returns the maximum thread identifier
}
```

Using the normal subtyping technique we can extend the type ACL to become a qualifying type which checks an invoking thread's right to modify or read the target object:

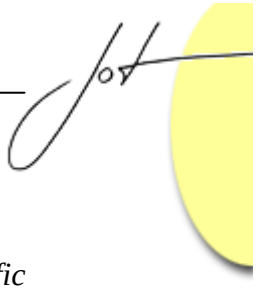
```
type ACLprotecting {
extends: ACL;
qualifies any:
 enq bracket op(...) throws InvalidAccess;
 enq bracket enq(...) throws InvalidAccess;
}
```

This could be implemented, re-using any implementation of ACL, as follows:

```
impl ACLprotectingImpl of ACLprotecting {
state: ^ACL acl;
qualifies any:
 enq bracket op(...) throws InvalidAccess {
  // the writer bracket
  if (acl.currentAccess(thisThread) == WRITE)
   return body(...);
  else throw new InvalidAccess.init();
 }
 enq bracket enq(...) throws InvalidAccess {
  // the reader bracket
  if (acl.currentAccess(thisThread) >= READ)
   return body(...);
  else throw new InvalidAccess.init();
 }
}
```

The keyword `thisThread` is a built in expression which identifies the current thread.

Bracket methods for qualifying `all`, `op` or `enq` are known as *standard* bracket methods, because they qualify non-specific methods of the target interface. Standard bracket methods cannot access parameters nor the values returned by the target object. Exceptions thrown by a target method are (implicitly) passed on unchanged. Standard bracket methods can throw exceptions, as the above example illustrates.



Qualifying Specific Views and Types

Bracket methods designed to qualify particular instance methods are known as *specific* bracket methods. These can be defined to qualify the instance methods of particular types, and they are often appropriate for qualifying objects of more than one type which contain the same view interface. Consider, for example, a view `Openable`, which might be incorporated into many "file" types:

```
view Openable {
 op void open(AccessMode mode) throws OpenError;
 op void close();
 enq AccessMode currentOpenMode();
}
```

A type designed to synchronise access to individual objects containing this view might be defined as follows:

```
type OpenSynchroniser {
qualifies Openable:   // qualifying a view
 op void open(AccessMode mode) throws OpenError;
  // this bracket method allows
  // multiple readers or a single writer.
  // OpenError is thrown if mode is set to NOACCESS
 op void close() throws InvalidAccess;
  // throws InvalidAccess if currentOpenMode is
  // NOACCESS, otherwise releases the synchronisation
 enq bracket op(...) throws InvalidAccess;
 // throws InvalidAccess if not open for writing
 enq bracket enq(...) throws InvalidAccess;
 // throws InvalidAccess if not open for reading or writing
}
```

The advantage of qualifying a view is that the same qualifying type can be used to bracket objects of different types (here different kinds of "files") which contain that view.

A specific bracket method declaration must match the signature of the method which it is designed to qualify, except that, as with standard bracket methods, the `op`/`enq` qualifier reflects the behaviour of the bracket method with respect to its own state data. It can be defined to throw some or all of the exceptions of the target method, and it can add new exceptions. An implementation of a specific bracket method can access the parameters intended for the target object and return a different value from that returned by the target object.

In an implementation of a specific bracket method the `body` statement can be used either with an ellipsis as its parameters (indicating that the parameters are passed unchanged to the target method) or parameter values may be explicitly supplied. Similarly, a different return value can be supplied from that which `body` returns. If the method is defined as void, the `return` statement need not be used.

The example illustrates that standard bracket methods can be defined in a `qualifies` clause for a specific interface. These are not to be confused with the standard bracket methods defined to qualify `any`. When appearing in a clause for qualifying a

specific view (or type), they are applied to all matching instance methods of the target object (including those not defined in the view) except those for which specific bracket methods have been defined.

5 USING QUALIFYING TYPES

Timor types can be instantiated either as dynamic objects (using the operator `new`) or as variables within objects (without `new`). Only dynamic objects can be qualified using the technique described in this section, which shows how qualifying objects can be associated with target objects. A later paper will describe how qualifiers can be statically associated with instances of other types.

Instantiating Qualifying and Qualified Objects

A qualifier is instantiated like any other object, as follows:

```
ACLprotecting* protected = new ACLprotecting.init(100);
```

In this example `protected` is a reference to a new `ACLprotecting` object. At this or any later point in the program's execution the instance methods of this object can be invoked to add and remove ACL entries, etc., e.g.

```
protected.addThread(20, READ);
```

This qualifier might be used to protect access to an object of type `Thing` (which we need not explicitly define, because objects of any type can be qualified by `ACLprotecting` objects). An association with the qualifier can be established when the qualified object is instantiated, e.g.

```
Thing* t1 = new {protected} Thing.init();
```

The same qualifier can be used to qualify more than one object, e.g.

```
Thing* t2 = new {protected} Thing.init();
```

Furthermore an object can have multiple qualifiers. Thus after instantiating a further qualifier using a statement such as

```
RWsync* synchronised = new RWsync.init();
```

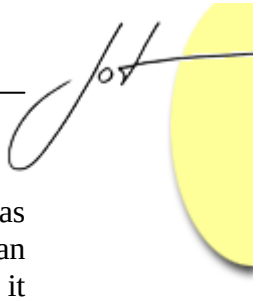
it would be possible to define another `Thing` qualified as

```
Thing* t3 = new {protected, synchronised} Thing.init();
```

The Qualifier List

The expression `{protected, synchronised}` is a normal Timor list literal, here with two entries. In the object creation context it refers to a qualifier list, which has the type `List<:Qualifier*:>` (specialising the generic type `List` as a list of `Qualifier` references). The type `Qualifier` is automatically a supertype of all qualifying types.

The effect is that all method invocations on this `Thing` object (whether via `t3` or via some other reference) are qualified via the `ACLprotecting` and then the `RWsync` object. When an instance method of the target is invoked, the qualifiers in its qualifier list are



searched for appropriate bracket methods, and these are scheduled from left to right as described in section 6. For example the `ACLprotecting` qualifier would permit an attempt by thread 20 to invoke an `enq` method on this `Thing` object to proceed, and it would then be synchronised as a reader, while an attempt by the same thread to invoke an `op` method would result in the exception `InvalidAccess` being thrown.

The qualifier list for an object contains *copies* of references to the qualifiers which apply to a target. Hence if a new reference value is subsequently assigned to a reference variable which has been used to add an entry to a qualifier list this does not affect the qualifier list.

Because any `List<:Qualifier*:>` expression can appear in an object creation clause, a qualifier list can be explicitly created as an object and then be associated with a target object or objects. For example the creator of the `Thing` in the above example might have used the following alternative code:

```
List<:Qualifier*:>* qualified = new List<:Qualifier*:>.init();
ACLprotecting* protected = new ACLprotecting.init(100);
RWsync* synchronised = new RWsync.init();
qualified.insert(protected); // inserts at end of list
qualified.insert(synchronised);
Thing* t4 = new qualified Thing.init();
```

Making the qualifier list a first class object has a number of advantages. For example, the qualifier list can be directly accessed as a separate object, with the effect that qualifiers can be dynamically added to and removed from it, even after the list has been associated with target objects. (In this case the qualification of targets is affected.)

A user can prevent other users from changing the qualifier lists of target objects which he creates simply by choosing to pass references for the targets into their scopes while not passing them references for their qualifier lists.

Alternatively a user might choose to protect the qualifier lists by qualifying these with qualifiers which carry out security checks. This is another advantage of making qualifier lists into first class objects, e.g.

```
ACLprotecting* secured = new ACLprotecting.init(100);
List<:Qualifier*:>* qualified =
            new {secured} List<:Qualifier*:>.init();
```

Synchronising Targets with their Qualifier Lists

Care must be taken when modifying qualifier lists to ensure that such activities are properly synchronised. This is the responsibility of the programmer, but it can easily be achieved using techniques which have already largely been described. For this purpose we define a qualifying type `ListSync`. This can be viewed as a special form of reader-writer qualifier which regards all critical accesses to a qualifier list as writer actions, and all accesses to its target object(s) as reader actions. In this way a change to a qualifier list can only occur when no other changes to the list are in progress and no methods of the target objects are active. On the other hand methods of the target object(s) can be invoked

in an unrestricted manner (e.g. in parallel) provided that the qualifying list is not being changed[3]. To achieve this the type `ListSync` can be defined as follows:

```
type ListSync {
qualifies List<:Qualifier*:>: // qualifies a qualifier list
 op bracket op(...);
  // operations on a qualifier list are treated as writers
 op bracket enq(...);
  // enquiries on a qualifier list are treated readers
qualifies any:          // qualifies any other target object(s)
 op bracket all(...);  // all methods are treated as readers
}
```

This example illustrates that even if a specific type (here `List<:Qualifier*:>`) is being qualified, the qualifier need not include specific bracket methods but can simply define general bracket methods.

The example also illustrates that a qualifying type can contain multiple `qualifies` clauses. The rule determining their applicability is that the most specific qualification applies. In the present case it means that if an object of type `List<:Qualifier*:>` is qualified the first set of bracket methods applies, whereas any other type is qualified by bracket methods of the second qualifies clause.

This qualifier might be used as follows to ensure properly synchronised access to a qualifier list and its target:

```
ListSync* ls = new ListSync.init(); // create the qualifier
List<:Qualifier*:>* qualified =
                        new {ls} List<:Qualifier*:>.init();
// create a qualifier list qualified by the qualifier
qualified.insert(ls);
// insert the same qualifier into the qualifier list
Thing* t5 = new qualified Thing.init();
// create the target as qualified by the qualifier list
```

This code takes advantage of the fact that the same qualifier can qualify multiple targets and that a qualifier list can itself be qualified (here even by a qualifier which is also in its own list)[4].

6 FLOW OF CONTROL

The flow of control of bracket methods is complicated by at least the following issues:

a) a `body` statement is a "normal" statement which can, for example, be used in a conditional statement, and it can be invoked more than once in a bracket method;

[3] Further qualifiers can be included in a target's qualifier list in order to provide appropriate synchronisation for the target itself.

[4] If the target is itself a qualifier list this code will not work quite as intended, as the target (qualifier list) will have its operations synchronised exclusively, although a higher degree of parallelism might in principle be achievable. However, this somewhat pathological case (which does not cause an error) does not merit a modification of the definition of qualifying types.

b) exceptions can be thrown by the target method and/or by bracket methods;
c) an object can be qualified by more than one qualifying object;
d) bracket methods can themselves be bracketed by other bracket methods; and
e) access to an object may proceed (using the dot notation) via other objects which are themselves qualified.

The combined effects of these possibilities leads to a non-trivial flow of control algorithm, the details of which must be left to a later paper.

Suffice here to say that the basic algorithm involves dynamically traversing a tree of objects with the target object at its root and its child nodes being those objects in its qualifier list which have a matching bracket method for the target method being invoked. Lower levels of the tree are based on the same principle, except that the bracket method selected at the higher level is considered to be the target method and child nodes are those objects in the qualifying object's qualifier list with a bracket method which matches this, recursively. The selected bracket methods are executed in endorder sequence (activated by the invocation of a target method or a `body` statement).

7 APPLICABILITY OF QUALIFIERS

Bracket methods are applied only to invocations of the public instance methods of a target object, including the set and get methods of its abstract variables. They are not applied to invocations of its private methods, nor to the invocations of the methods of its internal variables.

When the object dereferencing operator is used in a right side expression (i.e. when it returns the complete state of an object) this is considered to be the equivalent of an `enq` method. An object dereferencing operator used on the left side of an assignment statement is similarly considered to be equivalent to an `op` method. Hence `enq` or `op` (or `all`) bracket methods associated with qualifiers are applied as appropriate to dereferencing operations on targets.

If instance or bracket methods of an object invoke public instance methods of the object itself (whether or not the keyword `this` is explicitly used) these are not subject to qualification, because that could lead to difficulties such as creating deadlocks when a synchronisation qualifier is used.

If it were possible to pass a reference for an object to the object itself which could then be used by the object to invoke its own public methods, this could also create situations which could easily lead to bracketing problems such as deadlocks. However, the rule designed to prevent binary instance methods (cf. section 2) also prevents this situation from arising under normal circumstances.

When an abstract reference of an object is read (e.g. using the dot notation, to reach another object), this is of course an invocation of the get method associated with the object which contains the reference, and its qualifiers must be applied accordingly. For example if the object has a `qualifies any` qualifier then the standard `enq` bracket

method must be applied (unless a more specific bracket method applies). However none of the qualifiers associated with the referenced object is applied at this stage.

8 TYPE COMPATIBILITY AND POLYMORPHISM

The type of a reference to which a target object can be assigned does not reflect that qualifiers may be associated with the object. Hence given the existence of a type `Thing`, the following statements are all valid:

```
Thing* t1 = new Thing.init();
Thing* t2 = new {protected} Thing.init();
t1 = t2; // t1 refers to the protected Thing created in line 2
```

This reflects the decision that the qualifiers associated with an object do not formally affect the type of the object, i.e. a qualified object is always compatible with an unqualified object which has the same basic type.

On the other hand the behaviour of an object can be radically influenced by its qualifiers. For example it might be synchronised or monitored or protected. In the last case the effect is that it may not be called at all, and if a qualifier is used as a decoy object the client may not be able to observe that the behaviour has changed. In some cases qualifiers may throw exceptions unexpectedly (e.g. to indicate access violations), or a qualifier might demand a password from the user at the terminal, etc. These points clearly indicate that behavioural conformity in the sense of [17] can by no means be guaranteed, though often the behaviour could be described as "behaviourally conform if successful" (e.g. if an `ACLprotecting` object allows an access to proceed).

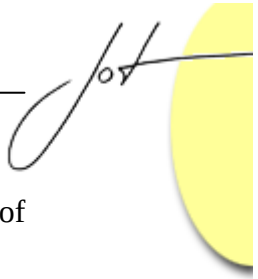
If a qualified object is assigned to a supertype reference the normal subtyping rules apply. The methods invocable from the supertype reference are dynamically scheduled in the usual way, and the bracket methods are still applied, of course.

9 BRACKET METHODS VS. SUBCLASSES/SUBTYPES

It is sometimes assumed that qualifying types can be viewed as an alternative to subclassing/subtyping. In [12] we discussed the relationship between these two concepts in more detail, reaching the conclusion that the latter are considerably more powerful than normal subtyping/subclassing, for at least the following reasons.

a) There is no general way of simulating `qualifies any` via subclassing in the standard OO paradigm. Each class qualified in this way must be separately subclassed.
b) There is no general way of imitating standard bracket methods for `all`/`op`/`enq` methods. Each superclass method must be separately programmed in a subclass with the code which appears in these brackets, depending whether it is a reader or a writer.

Points a) and b) taken together help to explain why it is impossible, for example, to have user-supplied standard synchronisation modules in the standard OO paradigm. (There are



additional reasons, e.g. concerned with static variables and methods and with the use of public fields in the OO paradigm, but these are not relevant to the present discussion.)

c) Although in many cases the use of specialised qualifying types can be simulated by subclassing, this is not always so. The creator of a new target object in Timor can combine different qualifiers in any order which he chooses. Using subclassing is less modular in that an order has to be statically determined for successive subclasses.
d) As we have shown in [12] in an example involving various orthogonal combinations of bounded buffer synchronisation, subclassing cannot always be reasonably used to simulate qualifying types.
e) Qualifying types are more powerful in that they can provide code which in reasonable circumstances permits methods declared as *final* to be "overridden" e.g. to protect an object.
f) The instance methods of a qualifier do not automatically become methods of the target. This is important in a case such as `ACLprotecting`, where a protected object should not also include the methods which allow the ACL to be modified! (Passing the object via a supertype variable does not help if the language allows downcasts.)
g) Subclassing cannot be used to simulate the fact that a single qualifying object can be used to qualify more than one object.
h) Subclassing cannot be used to define a subclass which modifies its own behaviour as a superclass, whereas one `ACLprotecting` object can be used for example to protect the access to another object which is also an `ACLprotecting` object.

This list shows that subclassing is not a real alternative to qualifying types. On the other hand we do not regard it as a replacement for subtyping, which is also supported in Timor. Subtyping in particular is important when types are being modelled which can be used polymorphically, e.g. a set or a bag can be modelled as more specific examples of a general type collection. In such cases we are concerned with variants of the "same" kind of thing. On the other hand the idea of qualifying types becomes more significant when we are concerned with programming quite different aspects of a problem.

10 COMPARISON WITH OTHER WORK

This paper has described the basic concepts of qualifying types with bracket methods. The idea is based on earlier work in our group [7, 8].

The idea that code can be bracketed is by no means new, and dates back at least to Pascal-Plus [24]. 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 [18] 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 [15] (cf. `body`) appears in a superclass method, the same method in a subclass is bracketed by the superclass method's code. But a Beta superclass `RWsync` needs to know which methods occur in its subclass `RWsyncThing` in order to bracket them and is therefore of no use in bracketing `RWsyncBook`.

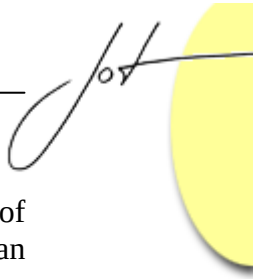
Mixins are a generalization of both the `super` and the `inner` constructs. The language CLOS [6] allows mixins as a programming technique without supporting them as a special language construct, but a modification of Modula-3 to support mixins explicitly has also been proposed [3]. A mixin is a class-like 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 [22] 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 methods 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 methods as they cannot 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. But even for methods which require bracketing the approach is inefficient: the proxy object and an associated handler must both be invoked, and reflection must be 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 method 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 [19]. 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 [20] 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 methods. In both cases there appears to be no equivalent to general bracket methods.

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 [14] and similar languages (cf. [23]) can achieve many of the aims of qualifying types, but with a number of limitations:

a) 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). Target classes not developed according to the convention must each be examined individually and separate aspects developed.
b) 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.
c) Because aspects are not separately instantiated an aspect "instance" cannot be flexibly associated with a group of objects rather than a single object.
d) 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!
e) 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 methods is easily defined at the time a target object is created.
f) 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 (nor the presence at compile time of) each other's source or bytecode or that of types which they might qualify.

11 CONCLUDING REMARKS

Qualifying types and their implementations, as defined in Timor, strongly support the idea that a system can be built from separately developed, re-usable components which need not have an advance knowledge of each other. It is easy for example to develop qualifying modules which can synchronise, protect or monitor target modules in a general way, although the targets may have been developed in complete isolation from them or may not even exist at the time the qualifiers are produced. And because qualifiers are themselves first class objects they can themselves easily be qualified by other qualifiers.

Mixing and matching such modules is easily and flexibly achieved at run-time using object creation expressions which allow multiple qualifiers to be associated with a single target and/or multiple targets to be qualified by a single qualifier. This flexibility contrasts with other approaches to weaving aspects into a single program, where the qualifying code is in effect cut and pasted into existing types at the source or bytecode level and thus affects the static types of objects.

Achieving this level of flexibility in a general way was only possible by defining a new language with particularly clean structures for type definitions. These include

a) replacing public fields by abstract variables that correspond to instance methods,
b) a clear distinction between instance methods and binary methods (which can only be defined as binary type methods),
c) designating each instance method as either an `op` (writer) or an `enq` (reader),
d) abandoning Java's idea that all type instantiations are referenced objects in favour of the concept that a type can be instantiated either as a separate object (which can be referenced from other objects) or as a value component of some other object,
e) abandoning Java's concept of type related static variables and methods.

It has been impossible in the space available for this paper to provide a fuller discussion of Timor's structure, and how it handles features such as static variables, which cannot be cleanly integrated with the concept of qualifying types.

It has similarly also been impossible to describe other features of qualifying types, e.g. how they can be integrated more statically with the type definitions of their targets (e.g. how to produce a type which is always synchronised) and how calls out of a target can be qualified. Lack of space also prevents us from providing examples of qualifying types which program aspects that are conceptually strongly integrated with particular types. These are all topics which will be addressed in future papers.

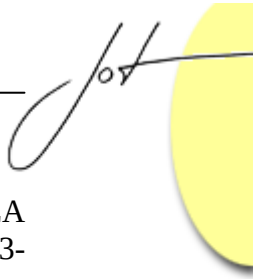
ACKNOWLEDGEMENTS

Special thanks are due to Dr. Mark Evered and Dr. Axel Schmolitzky for their invaluable contributions to discussions of Timor and to the ideas which have been taken over from earlier 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] K. B. Bruce, L. Cardelli, G. Castagna, et al., "On Binary Methods," *Theory and Practice of Object Systems*, vol. 1, no. 3, pp. 221-242, 1995.

[5] 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.

[6] 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.

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

[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] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] B. W. Lampson, "Protection," Proc. 5th Princeton Symposium on Information Sciences and Systems, 1971

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

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

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/

Klaus Espenlaub is working towards his Ph.D. in Computer Science at the University of Ulm. Currently he works as a research assistant in the Department of Computer Structures at the University of Ulm. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is espenlaub@informatik.uni-ulm.de.

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.