Encapsulation, delegation and inheritance in object-oriented languages

by Mario Wolczko

For the reuse of software to become routine, it is essential that all modules have well defined interfaces and that all users of these modules respect these interfaces. This paper examines the interfaces in object-oriented programs. It uses the notion of *delegation*, which can be thought of as underlying all inheritance mechanisms, to provide a framework for the examination of encapsulation mechanisms in object-oriented languages. Using delegation, the encapsulation mechanisms in class-based languages are reviewed, and suggestions are made as to how they might be improved.

1 Introduction

When building large systems, it is important that the designer of each component is able to specify the behaviour of that component independently of its implementation. This has several benefits.

- Once a component has been specified, it can be implemented independently of other components.
- Implementation at one design level can proceed independently of design and implementation at lower levels.
- The implementation of a component can be changed at any time, so long as it meets its specification.
- A readily available component can be used when it is clear that it meets the required specification.

In an object-oriented language, the components are object definitions. One of the main advantages claimed for object-oriented programming is that a designer can reuse existing object definitions readily. Furthermore, it is claimed that, since object definitions are modular, you can switch between alternative implementations of an object with minimal fuss. For this to be true, it is important that there is a clear understanding between the provider of an object definition and a potential user as to exactly what is part of the specification, and therefore invariant between implementations, and what is part of the implementation and therefore subject to change.

Clearly, there are many ways of communicating this distinction to a user. The most common technique is that of informal comment, either embedded in the program text or as a separate user guide. However, comments suffer from the usual problems of being inaccurate, unmaintained, unchecked and unenforced. The other extreme is represented by a formal specification in a language that can describe the semantics of a program at the required level of detail, while still retaining precision. To check that a module met its specification, and that user's code did not violate the assumptions of the module, would require the programmer and user to conduct formal proofs. This approach is difficult and time-consuming, and is not widely accepted in the industry. An intermediate approach is to provide language constructs, in which the programmer can make some assertions about the behaviour of the module, and which can be checked and enforced by the language system.

To this end, various language constructs have been proposed that attempt to distinguish the 'public' features of an object, which are necessarily part of the specification, from the 'private' features. Such constructs are used to *encapsulate* an object, hiding its private features so that they are externally invisible. This acts to document, in a formal way, which features are visible and to enforce the invisibility of private features. The use of such mechanisms ensures than a programmer can readily change the implementation of an object and be certain that users of that object are not relying on a private feature. It cannot, of course, guarantee that the new implementation of a feature is entirely compatible with a previous version, or that the user is not relying on an artefact of the previous implementation; only formal specifications and proofs can do this.

The practical benefit of a good encapsulation mechanism is that certain changes can be made in the knowledge that no existing programs will break. The sorts of changes that we would wish to include in this category are

- ☐ factoring common code into a shared procedure.
- ☐ adding, removing or renaming an instance variable (stored attribute) or a private method.
- ☐ changing the inheritance structure.
- ☐ dividing a definition of object behaviour into two or more parts related by inheritance.

To our knowledge, no existing object-oriented language enables a programmer to make all of these kinds of changes without potentially compromising existing code.

In this paper, we examine the issues of encapsulation in object-oriented languages and attempt to provide a framework in which these issues can be discussed. We suggest language constructs and structuring principles that can be used to provide a complete set of encapsulation facilities.

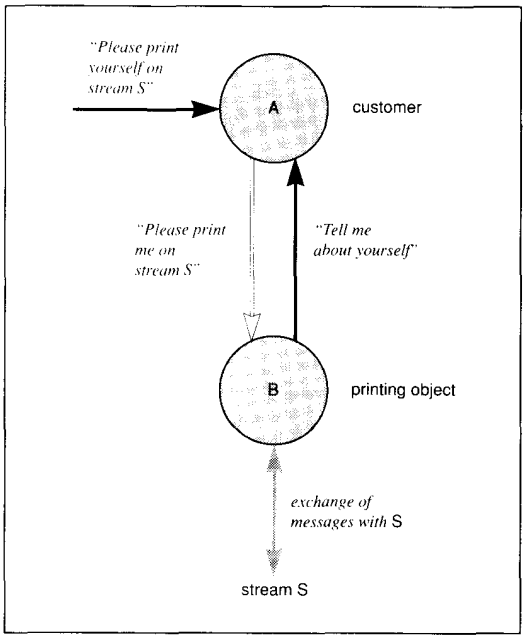

Fig. 1 Example of delegation

2 Encapsulation boundaries

By definition, every object-oriented system has at least one encapsulation boundary; that between the insides and outsides of objects. It is now widely accepted that, for a system to be called object-oriented, its objects must maintain control over access to their private state. The instance variables of an object are only accessible within the methods associated with that object and can be manipulated only in response to messages to that object.

We can also identify another aspect of encapsulation at the object boundary. Some of an object's methods may be present purely as a structuring mechanism to provide a separate implementation of a facility used by other methods. This is a technique common to conventional programming languages, and is used to improve the modularity and readability of code. For example, in a conventional language, an algorithm may be broken into separate procedures with each procedure performing a single logical task. The procedures need not be useful in themselves but only as part of the implementation of the whole algorithm, and therefore should not be part of the external interface. Another motivation is that of code-sharing; rather than having two identical or near-identical pieces of code in a program, it is useful to create a procedure, possibly parameterised, and invoke it where necessary. In an object-oriented language, methods play the role of procedures, and therefore an object may possess methods simply to make the implementations of other methods cleaner. These ancillary methods, however, should never be invoked from external objects. For example, an object representing a collection may have a basic sorting method, which should only be invoked by interface methods that perform essential initialisation (e.g. setting up the direction of the sort or the message to be used to compare two keys).

A third encapsulation boundary is provided by the structures that define shared object behaviour, usually known as classes. Most object-oriented languages organise objects into classes. A class defines the structure and behaviour of all its instances. It lists the instance variables each instance is to possess and defines the responses to messages each instance may receive, usually by associating a method with each kind of message. You might expect some methods to be only visible within the same class, hidden from other classes. Encapsulation of classes is complicated by inheritance, another common feature of object-oriented languages. Each class may have one or more parent classes, from which it inherits both structure and behaviour. An instance of the child class responds to all the messages for which an ancestor class defines a response, together with any others mentioned locally. Should more than one class define a response to a message, the 'nearest' candidate definition in the class hierarchy is chosen. Different languages, especially those with multiple inheritance, have different algorithms for choosing the nearest method.

Inheritance means that we must distinguish the boundaries between a class and its parent(s), and between a class and its children. Clearly, when a class is created, its parents must be known, and the specifications of its parents can be checked against a specification of what the class requires. However, in general, the children of a class are not known when it is designed and implemented. The designer must choose which aspects of the class are to be made part of the interface visible to children and which are to be private. This decision is almost independent of the other decisions; features may be visible to child classes that are not 'public' features of the object, or indeed accessible to other objects of the same class or the class object itself.

Some attempts have been made at identifying the encapsulation boundaries in object-oriented languages and at providing language constructs to support them. For example, Snyder [1] provides an excellent discussion of encapsulation in the presence of inheritance. However, no attempt has been made to place these discussions in a more general framework, and some issues have not been covered at all. This paper describes the issues and suggests possible solutions, using delegation as a framework.

The concepts of class and inheritance are familiar to most object-oriented programmers. However, there is much less awareness of an alternative organisational scheme that is conceptually simpler and more flexible; *delegation* [2]. The principal difference between class-based inheritance and delegation is that the former relates whole classes by inheritance, whereas the latter relates individual objects. This enables each object to make its own decision as to when and where it delegates, allowing the pattern of inheritance to vary dynamically. In contrast, the inheritance pattern in a class-based system is fixed when the classes are created. This makes delegation a more flexible and powerful way of organising objects, in that it can be used to model class-based inheritance, whereas the converse is not true [2]*.

* Although Stein [3] claimed that inheritance and delegation are in some sense equivalent, she assumed that classes were first-class objects and could respond to messages. This is not the case in many object-oriented languages.

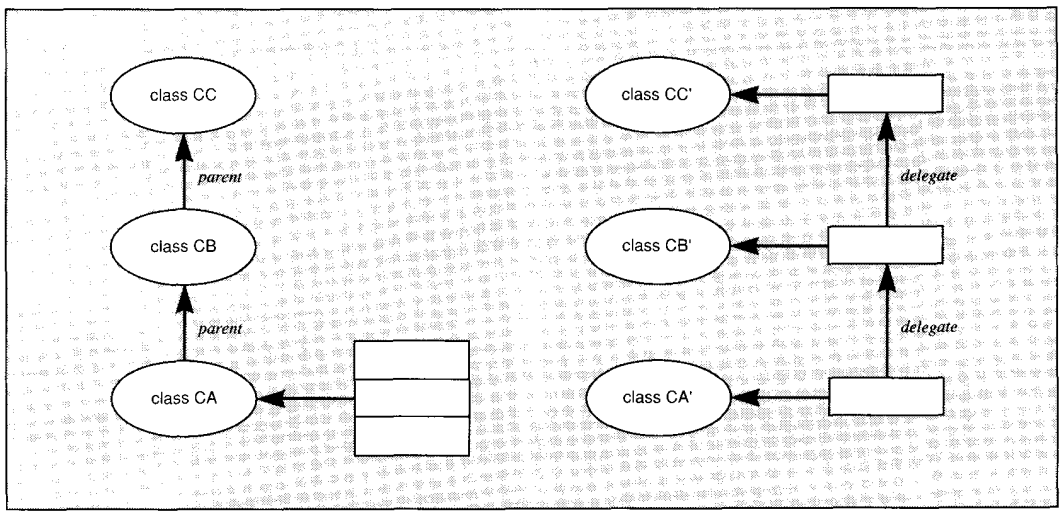

Fig. 2 Class hierarchy and instance

On the left is a class hierarchy and an instance of one class. The instance comprises parts that are defined by each class in the hierarchy. On the right is a similar structure, but using explicit sub-objects and delegation. The classes on the right are not related by inheritance.

3 Delegation

In a delegation-based system, there are two kinds of messages. The first kind is present in all object-oriented systems and simply involves sending a message from one object to another, possibly passing some parameters. In the second kind, one object *delegates* a task to another. As in real life, delegation implies shared responsibility for the completion of a task. When a manager delegates a task to a subordinate, the subordinate is expected to consult the manager should help be required. Similarly, when one object delegates a message to another, it expects to be consulted should further information be required.

In practice, the effect is simple; when object *A* delegates a message to object *B*, a reference to *A* is implicitly passed to *B*. Object *B* knows object *A* as its *customer* or *client*. For example, consider an object that wishes to print a representation of itself on a text stream (Fig. 1). It may not know how to manipulate the stream directly but could delegate the task to a 'printing' object that interfaced to the stream. However, the printing object might need extra information to complete its task, such as a string representing the internal state of the object to be printed. To do this, it sends appropriate messages back to the customer object. The task of assembling these strings and sending them to the stream has been effectively delegated to the printing object, and the manager simply has to provide the extra information specific to the task at hand, without having to be aware of the finer details of printing. Using delegation has advantages over passsing the information as extra arguments; it loosens the coupling between objects, making the interface more flexible, and it allows the extra information to be generated on demand.

Of course, a delegation-like style of programming can be employed in a class-based language, by passing the reference to the customer explicitly with each message. However, there is a deeper connection, which is explored below.

3.1 The relationship between delegation and class-based inheritance

In a class-based system, the methods of a parent class can be invoked in one of two ways; either by being the 'nearest' definition of a method in the class hierarchy to the actual class of an object, or by being invoked explicitly by 'nearer' methods. In Smalltalk, for example, the latter occurs when a method in a parent is invoked using the super keyword. However, in the method invoked by the send to super, self still refers to the same object and messages to self are looked up in the class of self. There is a direct analogy here with delegation; we can consider each ancestor class as defining a subordinate object and invocations of methods via super delegate tasks to these subordinates. Self plays the role of the customer. This is illustrated in Fig. 2; on the left, an instance of class *CA* possesses the instance variables defined by *CA* and its ancestors *CB* and *CC*. On the right, an equivalent set-up is shown using three objects, which are instances of classes *CA'*, *CB'* and *CC'*; these classes are not related by inheritance.

In this analogy, a method that is inherited by one class from another can be considered as a default delegation, for which most languages do not require an explicit definition. This is shown in Fig. 3, where a fragment of Smalltalk is compared with an equivalent fragment in a delegation-like language.

Using this analogy, we can think of class-based inheritance as a technique for building systems where the patterns of inheritance can be determined statically. A class-based system uses this static information to build a more efficient implementation; it can statically bind messages to super, and need not construct individual subordinate objects because it is known that they can never be referenced directly. In the rest of this paper, we use delegation to examine encapsulation issues and then transfer the results into a class-based environment.

The relationship between delegation and class-based

```
class          X
printOn: aStream
  aStream nextPutAll:
  (self class name, self summary)

class          Y
superclass     X
summary
  ↑'string representing internal state ...'
```

```
class          X
printOn: aStream
  aStream nextPutAll:
  (customer class name, customary summary)

class          Y
instance variables    xComponent
printOn: aStream
  'Delegate this task to my x component'
  xComponent.delegate printOn: aStream
summary
  ↑'string representing internal state ...'
```

Fig. 3 Pseudo-code

The pseudo-code on the right is a delegation-based version of the Smalltalk on the left. On the left, class Y inherits the printOn: method from class X; a printOn: message to an instance of Y will invoke the inherited method, which, by sending summary to self, will invoke the summary method in Y. On the right, the use of self (e.g. in printOn:) has been replaced by customer, and inherited messages (such as printOn:) are delegated explicitly.

inheritance also extends to multiple inheritance. The various types of multiple inheritance [1] can all be modelled using delegation between component objects. This enables us to discuss the properties of various inheritance mechanisms using a common base of delegation.

It should be emphasised that classes and delegation can co-exist. In the simplest case, a system can use classes and delegation but have no inheritance mechanism to relate classes. Thus, a class defines the behaviour of its objects completely, and the programmer must use delegation to relate objects to their components, as in Fig. 3. In a more general system, there may be both delegation and inheritance between classes. As we have seen, the class inheritance mechanism can be used as a shorthand for delegation, when the pattern of inheritance can be fixed in advance. We can even envisage tools that analyse a program to discover where delegation can be replaced with conventional inheritance. Such tools would be invaluable in transforming a prototype system into a production system, as they would allow the programmer to discover and classify in a formal way the relationships between objects.

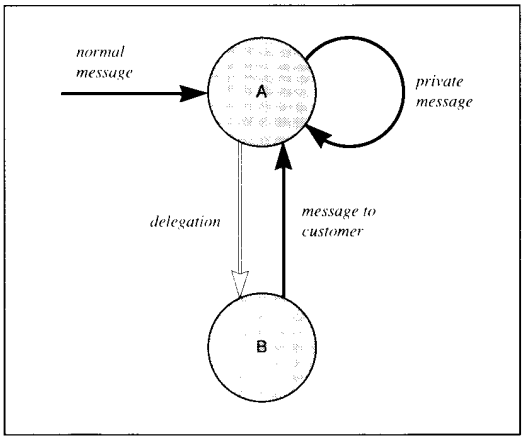

Fig. 4 Object interfaces

Every object has four interfaces: to normal messages, to private messages, as a customer (e.g. object *A*) and when being delegated to (as *B*)

4 Encapsulation and delegation

Given the connection between delegation and class-based inheritance, we can now examine the encapsulation issues using a general and powerful base. In this Section, we look at the problems of encapsulation in a delegation framework and relate these to class-based systems.

4.1 Encapsulation of instance variables

To retain the advantages of object-orientation, instance variables must be hidden from other objects by having access to them mediated via messages. If we accept this idea in a delegation-based environment, the carry-over to the class-based systems is clear; a class should not have direct access to instance variables defined in its ancestors. If inherited classes represent component objects, direct external access to the internal state of these objects should be impossible, as with all other objects. Snyder reached this conclusion by an alternative argument, that of encapsulation for maximal reusability [1].

All access to inherited instance variables should be via messages. If the implementor of a class makes clear which instance variables can be accessed or set in subclasses, a traditional variable accessing syntax can be used as a 'sugar coating' for the usual message-sending form. This does not violate encapsulation principles; a replacement class could provide methods that mimic the behaviour of the inherited instance variable.

4.2 Encapsulation of methods

It was mentioned earlier that some methods should be entirely private to an object, callable only by direct invocation from that object. Although this seems a relatively obvious encapsulation mechanism, surprisingly few languages support it.

Once delegation enters the scene, a new set of possible controls emerges (Fig. 4). A first step is to have an object only respond to some messages when those messages have been sent from objects to which it has delegated a task, i.e. when the object is playing the role of the customer. The motivation for this is simple; delegation takes

place when one object wishes to have another co-operate to accomplish a task. To achieve this, the object being delegated to may need privileged access; just as in real life a subordinate may have a 'hot line' to their manager. When an object delegates, it can choose to which objects it delegates and when, and thereby control which objects have extra access rights.

Similarly, an object may have methods that should only be invoked by delegation. This is complementary to the previous type of encapsulation; if object *A* wishes to send a message *m* to object *B*, *B* may need privileged access to *A* in order to fulfill the request implied by *m*. It can guarantee that it has this by marking the method for *m* as executable only by delegation.

Carrying these ideas over to class-based systems, we see that encapsulation between an object and its delegate takes place when messages are passed between a class and its parent. A message to super is a delegation to a subordinate object, whereas a message to self is equivalent to a message to the customer. The programmer should be able to control visibility at both of these interfaces.

The ability to control messages to super already exists in languages such as Trellis/Owl [4] and C++ [5], in the form of *subclass-visible* methods (known as *protected members* in C++). The other category of methods, i.e. those that should only be visible to superclasses (to be invoked by sending a message to self), is not available. Note that there is a subtle distinction between these *superclass-visible* methods and private methods. A private method is only visible within the same class in which it is defined (or, in delegation terms, to its associated object), whereas a superclass-visible method can be invoked by sends to self in superclasses.

4.3 *Hiding messages*

In existing languages, the external interface of an object is the set of messages to which it responds (and, in some languages, the set of instance variables accessible in subclasses). For the most part, these languages ignore another essential aspect of an object's interface, i.e. the messages it *sends*. For example (Fig. 3), if object *y* sends a printOn: message to object *x*, clearly printOn: is part of the interface to *x*, *but so is the fact that x sends* summary *back to y in response*. We can think of every object as being an implementation of an abstract object type, which describes the behaviour of that object and all compatible objects [6]. The type describes not only the set of messages that an object will respond to, but also what the *externally observable* response is. For proper control of encapsulation, a programmer should be able to control the visibility of messages, as well as methods. Ideally, we would also be able to describe the circumstances under which each message emanates from an object, perhaps using the notion of *contracts* [7].

The closest that most languages come to providing a description mechanism for an object's outgoing message interface is the provision of *deferred* or *pure virtual* methods (described using subclassResponsibility in Smalltalk). The presence of a virtual method *m* in a class *C* is a declaration that a message of that name might emanate from *C*, sent to self, but that no response is defined by *C*; a descendant class must provide a response. This is similar to saying that a customer delegating a message to an instance of *C* should expect to receive a message *m* in response.

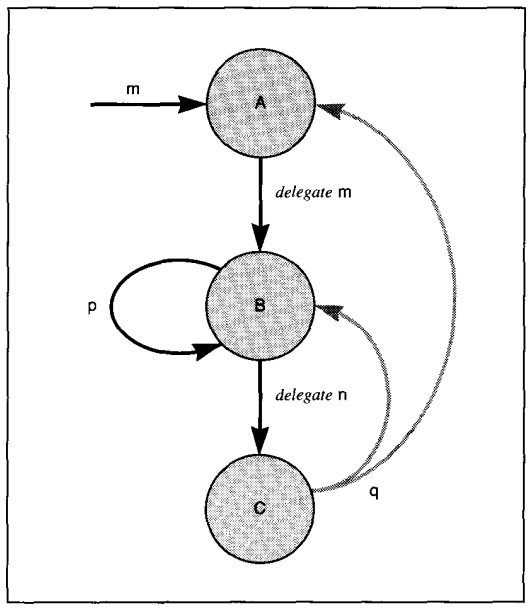

Fig. 5 Use of different object interfaces
Object *A* receives a message *m*, which it delegates to *B*. *B* performs part of the task locally by sending *p* to itself, but also delegates another part, *n*, to *C*

4.3.1 Private messages: Fig. 5 shows one aspect in which encapsulation should be provided. When object *A* receives a message *m*, it delegates part of the task to object *B*, which performs some of the task locally but, in turn, delegates a part of its task to *C*. Clearly, *B* should have the opportunity to complete some part of its task without communicating with *A* if it so desires, by sending a message to itself. If *A*, *B* and *C* were all component parts of an object defined by classes *CA*, *CB* and *CC* (as in Fig. 2), this would mean that a method in *CB* would be able to invoke another method in *CB* without inadvertently picking up a redefinition in *CA*. In many languages, this is not possible; Smalltalk, for example, insists that sends to self are always rebound in the class of the receiver. Other languages use conventional, statically bound procedure calls to achieve this (such as private non-virtual functions in C++).

This can also be illustrated using a Cunningham-Beck diagram [8]. In such a diagram (Fig. 6), an object is represented by a layered box, each layer corresponding to a class in the object's class hierarchy. The class of the object is at the top, with superclasses below. A message send is represented by an arrow, whose tail is in the class from which the message is sent and whose head is in the class that provides the method for the message. Arbitrary sends enter an object from the top, i.e. in the object's class, and proceed downwards ('up' the superclass hierarchy) until they encounter a corresponding method. Sends to self rise up out of the object, re-enter in the top (the object's class) and then drop down ('up' through the class

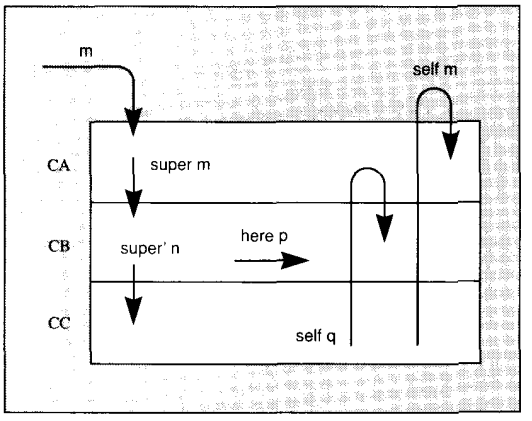

Fig. 6 A Cunningham-Beck diagram [8] representing the message interfaces between the different classes that define an object

An instance of class *CA*; cf. Fig. 5

hierarchy) until a method is found. Sends to super proceed directly downwards. To allow a programmer to encapsulate behaviour fully, it should be possible for an arrow to go sideways within an object, i.e. be confined to a single class.

This enables us to send a message within a class without it constituting part of an object's external interface. Additionally, by declaring the method executed in response to the message as private to the object, we can ensure that the use of the method is totally invisible outside the object. For maximum flexibility in control of the interface, the privacy of the method should be independent of the privacy of the message.

4.3.2 Localising communication: referring back to Fig. 5, consider also the response of *C* to the message *n* delegated to it by *B* as part of *B*'s response to the message *m* from *A*. If *C* needs to communicate with the customer by sending it message *q*, which object does it use, *A* or *B*? Is delegation transitive? In our opinion, it should be *B*'s decision as to whether *C* sees *B* as the customer or *B*'s customer (in this case *A*) as the customer. The interface to *C* should make clear that, when *C* is delegated message *n*, it may contact the customer with one or more messages. In turn, *B* can decide whether it should handle these messages (thereby making the use of *C* totally invisible to *A*) or pass them to its customer (thereby making them part of the public interface). Existing class-based languages only deal with one of these cases; if a class *CB* uses super to invoke a method in its parent *CC*, and *CC* uses self expecting a subclass to provide a method, there is no way for *CB* to ensure that it will intercept the message, making its use invisible to its descendents. This is one aspect of 'the problem of self' described by Lieberman [2].

The inability to 'close' the interface to a class, by capturing such messages and making them invisible to subclasses, means that, in general, it is impossible to split a class into two or more classes related by inheritance, without compromising encapsulation. This makes it hard to divide a class into smaller parts, such that each part is a useful class in its own right, and hinders the evolution of reusable classes.

5 Encapsulation in MUST

The encapsulation issues described thus far are being investigated in a research language called MUST [9]. MUST is based on Smalltalk-80, but includes the following new features:

- **Multiple inheritance** based on 'tree' semantics [1]. Instance variables are private to a class.
- **Method encapsulation:** each method in a class can be categorised as either
  - ☐ private (only visible within the class in which it is defined).
  - ☐ subclass-visible or superclass-visible, or both (visible to super and self, respectively).
  - ☐ public.
- **Message encapsulation:** messages can be of
  - ☐ the 'normal' kind (i.e. sent to an object which is the result of an expression), which contacts an object via its public interface.
  - ☐ sends to super (as in Smalltalk-80, but qualified with a parent class name in the presence of multiple parents), which accesses the subclass-visible interface of a parent class
  - ☐ sends to self, which uses the superclass-visible interface.
  - ☐ sends to here, which uses the private interface to invoke a method in the same class, without the opportunity for interception in subclasses.

As in Smalltalk, inherited components do not actually exist as independent objects. This is possible because both super and here can only be used to indicate special forms of message sends; they cannot be used as identifiers for component objects.

- **Delegation:** in MUST, class inheritance is an abbreviation for delegation. In addition, an explicit delegation mechanism is provided. A message can be delegated to an arbitrary object, provided that the message is part of the object's subclass-visible interface; this rebinds self accordingly. There are two types of delegation; one passes on the customer (self) unmodified, and the other sets the customer to the object doing the delegation.

The provision of a delegation mechanism integrated with class-based inheritance enables the programmer to express patterns of inheritance in convenient ways that are not otherwise possible. For example, it is possible to build totally secure *proxies* (also known as *encapsulators* [10]), which can intercept a message stream to an object entirely transparently. This is made possible by the rebinding of self during delegation and the ability to categorise methods as private. Proxies can be used for serialisation, the construction of *futures* [11] and in distributed systems to forward messages to remote objects.

6 Outstanding problems

There are still some unresolved encapsulation issues in this framework. They can broadly be grouped into two categories: primitive methods and initialisation.

In Smalltalk-80, as in some other languages, primitive methods that are applicable to all objects are defined in the most general class, Object. This means that definitions of methods such as class (which returns the class of an

object) or == (which compares object identities) need only be in one class. However, if we view class inheritance as an abbreviation for delegation between sub-objects, it is clear that these methods belong in each class; it does not make sense to delegate a test for object identity to another object. There seem to be two solutions to the problem, neither of which is entirely satisfactory.

☐ Have a system of such primitive methods in every class, perhaps automatically provided by the system.
☐ Adopt rather contorted definitions of class, == etc., which operate on the customer.

Another problem arises if we wish to have classes represented by objects, and give each class the capability to create instances of itself. After creation, we would usually want to initialise an object to a state that satisfies the invariants for that object. However, this usually requires privileged access to the state of the newly created object; we would not want this access to be available to any other object after initialisation has taken place. One possibility is to have a special initialisation interface to each object. Another is to have specially designated constructor methods in the class, which are allowed to perform initialisation (as in C++ and Eiffel).

Finally, it should be mentioned that it has been assumed throughout this paper that the unity of modularity and encapsulation is that which defines the behaviour of a single object or component object (usually a class). Sometimes this may not be the case, in that a group of objects, unrelated by inheritance, will co-operate to perform a task. The collection of objects may be implemented as a unit, and encapsulation may only be an issue between the collection and external objects. In such circumstances, the encapsulation mechanisms proposed above are inappropriate (in that an object in the group may wish to allow others in the group privileged access, while maintaining privacy from objects outside the group). The friend construct in C++ and the selective export list of Eiffel attempt to address this problem, but place the access controls between classes and not between individual objects. The resolution of this problem is the subject of current research within the MUST language, using a more general capability-like mechanism based on *agents* [9].

7 Summary

We have explored encapsulation issues in object-oriented languages, using delegation as a framework for inheritance. It has been argued that existing class-based object-oriented languages do not provide sufficient support for encapsulation, especially in the area of message interfaces. Specifically, to enforce strict encapsulation, the following additional mechanisms are suggested.

- A class should be able to distinguish which of its methods are public, visible only to subclasses, and private.
- There should be a type of message send that can invoke a local method without the chance of being intercepted by a subclass.
- In addition to the conventional super send, which does not alter self, a version should be provided that rebinds self to the class of the sending method.

A new object-oriented language, MUST based on Smalltalk-80, has been designed to explore these issues and to gain experience with the suggested encapsulation mechanisms.

8 Acknowledgments

The author would like to thank Andrew Barnard, William Cook, Neil Dyer, Michael Fisher, Trevor Hopkins, Borek Vokach-Brodsky, Ifor Williams and the referees for their comments; and Andrew Barnard and Marion Godfrey for help in the preparation of the diagrams.

This work was supported by a Research Fellowship from the UK Science and Engineering Research Council and by Research Grant GR/E/65050.

9 References

[1] SNYDER, A.: 'Inheritance and the development of encapsulated software components' *in* WEGNER, P., and SHRIVER, B. (Eds.): 'Research directions in object-oriented programming' (MIT Press, Cambridge, Massachusetts, 1987) pp. 165–188
[2] LIEBERMAN, H.: 'Using prototypical objects to implement shared behavior in object oriented systems'. Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1986, pp. 214–223
[3] STEIN, L.A.: 'Delegation is inheritance'. Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, 1987, pp. 138–146
[4] SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAM, M., and WILPOLT, C.: 'An introduction to Trellis/Owl'. Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1986, pp. 9–16
[5] STROUSTRUP, B.: 'The C++ programming language' (Addison-Wesley, Reading, Massachusetts, 1991) 2nd edn.
[6] CANNING, P.S., COOK, W.R., HILL, W.L., and OLTHOFF, W.G.: 'Interfaces for strongly-typed object-oriented programming'. Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, New Orleans, Louisiana, 1989, pp. 457–467
[7] HELM, R., HOLLAND, I.M., and GANGOPDHYAY, D.: 'Contracts: specifying behavioral compositions in object-oriented systems'. Proc. Joint Conf. OOPSLA/ECOOP, Ottawa, Canada, 1990, pp. 169–180
[8] CUNNINGHAM, W., and BECK, K.: 'A diagram for object-oriented programs'. Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1986, pp. 361–367
[9] WOLCZKO, M.I.: 'Introducing MUST — the MUSHROOM programming language'. Technical Report of the MUSHROOM project, Department of Computer Science, University of Manchester, 1988
[10] PASCOE, G.A.: 'Encapsulators: a new software paradigm in Smalltalk-80'. Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1986, pp. 341–346
[11] HOPKINS, T.P., and WOLCZKO, M.I.: 'Writing concurrent object-oriented programs using Smalltalk-80', *Computer J.*, 1989, **32**, (4), pp. 341–350

The paper was first received on 6 March and in revised form on 29 November 1991.

The author is with the Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL.