Extending Ordinary Inheritance Schemes to Include Generalization

Claus H. Pedersen
Hewlett Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS12 6QZ
England
Phone: +44 272 799910
email: chp@hp.co.uk or chp@hplb.hpl.hp.com

Abstract

The arrangement of classes in a specialization hierarchy has proved to be a useful abstraction mechanism in class-based object oriented programming languages. The success of the mechanism is based on the high degree of code reuse that is offered, along with simple type conformance rules.

The opposite of specialization is generalization. We will argue that support of generalization in addition to specialization will improve class reusability. A language that only supports specialization requires the class hierarchy to be constructed in a top down fashion. Support for generalization will make it possible to create super-classes for already existing classes, hereby enabling exclusion of methods and creation of classes that describe commonalities among already existing ones.

We will show how generalization can coexist with specialization in class-based object oriented programming languages. Furthermore, we will verify that this can be achieved without changing the simple conformance rules or introducing new problems with name conflicts.

1 Introduction

Traditional class-based object oriented programming languages [3, 5, 6, 14] allow the programmer to create new classes either by describing them from scratch, or by specializing existing ones. This has proven to be a successful approach. Two of the main reasons for this are:

- Specialization allows existing class descriptions to be reused in the creation of new classes.
- In spite of the reuse offered, the conformance rules are normally very simple.

Conformance rules are the rules in the language that are used to check whether a type T conforms to another type S. A type T conforms to a type S if instances of type T can be used as if they were instances of type S. We also refer to instances of type T conforming to S.

In class-based object oriented programming languages, the classes are the types. If the classes are arranged in a specialization hierarchy the conformance rules become very simple. The rules are usually that a class A conforms to a class B if and only if A and B are the same class or B is an ancestor of A in the hierarchy.

More sophisticated type systems have been proposed to further improve reusability [7, 11], but

in these systems the programmer has to deal with more elaborate conformance rules. For a more thorough discussion of types and conformance refer to [1, 2, 4].

The reason that the conformance rules in class-based object oriented programming languages can be so simple is that the languages only support creation of sub-classes through specialization. Instances of any descendants of a class A are guaranteed to support the methods defined by A.

The organization of classes in a pure specialization hierarchy has been criticised for two reasons [8, 12].

- It is not possible to exclude methods that are defined in a super-class. This would destroy the basis for the simple conformance rules.
- It is not possible to create a new class that describes commonalities among existing classes. Specialization requires that super-classes are created before sub-classes. If we realize that there are similarities (a potential common super-class) among a set of classes, there is no support to create a more general class, using the specialized ones that already have been implemented. As more programs are developed in an incremental fashion, this problem will grow.

Both these restrictions can be eliminated by introducing a notion for generalization. This makes it possible to exclude methods, and it also makes it possible to create new classes that describe similarities among already existing classes. Both of these properties enhance class reusability in class-based object oriented programming languages. Generalization is an extension of the primitives found in traditional class-based object oriented programming languages. The explicit classification hierarchies, specialization and the simple conformance rules are all maintained.

The rest of this paper is organized as follows. Section 2 clarifies the notions of specialization and generalization by examples leading to more precise definitions. Section 3 gives examples to illustrate the use of generalization in programming. These are related to the current literature to show how generalization solves some of the established problems. Section 4 is the main section of the document. It describes how generalization can be introduced into class-based object oriented programming languages. Solutions to the major problems are presented. We also show that problems related to name conflicts are equivalent to those found in languages supporting multiple specialization. Throughout the paper we distinguish between the interface (method declarations) and the implementation (instance variables, code) of classes.

2 Specialization and generalization

Specialization and generalization are relationships between concepts. We will use a simplification of the Aristotelian view of concepts [10]. We will say that any concept is strictly defined by a set of properties called the *intension* of the concept. The *extension* of a concept is all phenomena fulfilling the intension. We will denote the intension of a concept C by $C^{intension}$ and its extension by $C^{extension}$.

An example of a concept is *mammal*. Its intension is "animals possessing mammae in which milk is secreted for the nourishment of their young"[1]. An example of a phenomenon belonging to the extension of mammal is my neighbour's dog.

In object oriented programming languages, concepts and relationships between them are often modeled by classes and class hierarchies. A class models the intension of a concept by defining the properties that must be fulfilled by all instances of the class (for example the methods they have). In programming terminology we would say that the neighbour's dog is an instance of class mammal.

[1] According to "The Shorter Oxford English Dictionary, third edition"

2.1 Specialization

A concept $C_{special}$ is a *single specialization* of a concept C if all phenomena belonging to $C^{extension}_{special}$ also belongs to $C^{extension}$. The concept car is a single specialization of the concept vehicle. This is because cars have all the properties of vehicles plus some more. This implies that a car will always do whenever one needs a vehicle, but not necessarily the other way around. This can be expressed formally by using the notation introduced above. A concept $C_{special}$ is a single specialization of a concept C iff: $x \in C^{extension}_{special} \Rightarrow x \in C^{extension}$. This can also be expressed by saying that any phenomenon x belonging to $C_{special}$ will at least fulfil the properties defined by $C^{intension}$.

A concept is a *multiple specialization* of a set of concepts if it is a single specialization of each of the concepts in the set.[2] The concept escalator is a multiple specialization of the concepts staircase and conveyer belt. This is because any escalator fulfils the defining properties of a staircase as well as those of a conveyer belt.

Formally, $C_{special}$ is a multiple specialization of $C_1,\ldots,C_n$ iff: $x \in C^{extension}_{special} \Rightarrow \forall\, i \in 1\ldots n : x \in C^{extension}_i$. We will use the term specialization to describe single as well as multiple specialization.

In this paper we shall view the methods described by a class as the defining properties determining membership of the class. Creating a sub-class corresponds to specialization, as new methods can be added or the behaviour of existing ones refined (virtual methods). Specialization has been modeled in this way by class-based object oriented programming languages for a number of years, starting with the class concept in Simula [5]. In programming terms we would say that class car is a specialization (sub-class) of class vehicle and class escalator is a multiple specialization of class staircase and class conveyer belt.

Some languages have other ways of defining sub-classes apart from adding and refining methods, e.g. restricting the possible values of an object. Such languages will only be partly covered in the discussion to follow.

2.2 Generalization

A concept $C_{general}$ is a *single generalization* of a concept C if membership of $C^{extension}$ implies membership of $C^{extension}_{general}$. The concept vehicle is a single generalization of the concept car because all cars are vehicles. Formally, $C_{general}$ is a generalization of C iff: $x \in C^{extension} \Rightarrow x \in C^{extension}_{general}$

Just as there is multiple specialization, so there is *multiple generalization.* A concept is a multiple generalization of a set of concepts if it is a single generalization of each of the concepts in the set,[3] i.e. membership of any of the concepts in the set implies membership of the generalization. Mammal is a multiple generalization of dog, elephant and human, because any dog, elephant or human is a mammal. Formally, $C_{general}$ is a multiple generalization of $C_1,\ldots,C_n$ iff: $\forall\, i \in 1\ldots n,\ x \in C^{extension}_i \Rightarrow x \in C^{extension}_{general}$

We will use the term generalization to describe single as well as multiple generalization.

Generalization corresponds to being able to create a super-class for an existing class or a set of classes. So if class A is a generalization of class B then class B will conform to A even if A was created after B. We do not know of any programming languages that explicitly support generalization.

We note that A is a generalization of B if and only if B is a specialization of A.

The relationship between single/multiple specialization and generalization is illustrated in figure 1 below. A_1 is a (multiple) generalization of $B_1,\ldots,B_4$. A_2 is a (single) generalization of B_4. $B_1,\ldots,B_3$ are all (single) specializations of A_1 and B_4 is a (multiple) specialization of A_1 and A_2.

[2] Note that multiple specialization is single specialization if the set of concepts consists of only one element.

[3] Note that multiple generalization is single generalization if the set of concepts consists of only one element.

$B_1,\ldots,B_4$ all conform to A_1 while only B_4 conforms to A_2

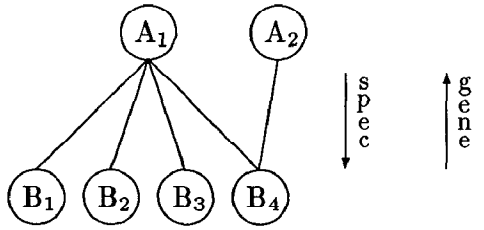

Figure 1: Generalization and specialization.

3 Use of generalization in programming

We will discuss the usefulness of a notion for generalization in class-based object oriented programming languages. In section 4 we will show that it can actually be supported and that multiple generalization causes no new problems compared to multiple specialization.

3.1 Exclusion of methods

One obvious use of (single) generalization is to reuse selected parts of an already existing class. An example of this which has been discussed in the literature is deque [12]. A deque is a stack which permits elements to be added and removed from either end. Deque is illustrated in figure 2 below.

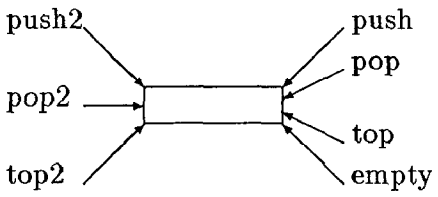

Figure 2: Deque

Assume that class deque is already implemented. Clearly, deque implements the functionality of an ordinary stack (push, pop, top, empty), but in order to convert deque so that it behaves strictly as a stack, we must be able to exclude the methods push2, pop2 and top2 from its interface. As pointed out in [12], this cannot be described in an inheritance scheme that only supports specialization. In such a scheme we will have to create class stack, and then make deque a specialization of it.

Introducing a notion of generalization solves the problem. Class stack can be obtained by generalizing class deque. This makes stack a super-class for deque. In this way stack has been created by reusing selected parts of deque and without violating the conformance rules. Stack as a generalization of deque is illustrated in figure 3.

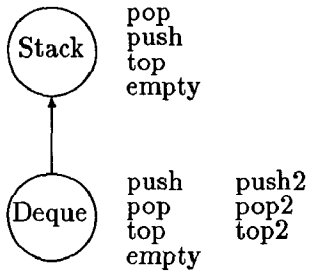

Figure 3: Stack is a generalization of deque and deque therefore a subclass of stack.

3.2 Late classification

People recognize commonalities among concepts by seeing a number of them. We have to understand a number of "specializations" before we realize that there is a more general concept that defines the similarities in their own right. Normally we have to understand concepts such as car, bus and lorry before we understand the concept vehicle. This learning pattern is not supported in an ordinary inheritance scheme, where super-classes have to exist before sub-classes.

This was one of the arguments used for the introduction of delegation [8]. Delegation allows programmers to use objects as "prototypes" for other objects without having to create a classification hierarchy. Delegation is an elegant way of modeling how we initially describe phenomena by relating them to other phenomena, but it does not help us in creating a class hierarchy when we have obtained enough knowledge to do that. Generalization will support this.

As an example of this, consider a programmer who is implementing classes that enable programs to control different types of terminal devices. He has implemented classes $Terminal_1$ and $Terminal_2$. A new type of terminal, $Terminal_3$, arrives, and he realises that it is an ANSI terminal – it supports the ANSI-defined control sequences. As well as these, it has other means of control that enable optimized screen control. The programmer realizes that this was also the case for $Terminal_1$ and $Terminal_2$. He can create $Terminal_{ANSI}$ by generalizing the classes $Terminal_1$ and $Terminal_2$. Then he can create class $Terminal_3$ as a specialization of $Terminal_{ANSI}$. He will only have to implement methods that allow screen control in the ways not defined by the ANSI standard.

Not only did the programmer save some work implementing class $Terminal_3$ but he also obtained a more general class that describes a common behaviour which all the terminal types have. Thereby he explicitly stated relationships among the different kind of terminals. These relationships ensure that programs can do basic screen control assuming that the terminal is of class $Terminal_{ANSI}$ (more specific screen control will still require knowledge of the exact type of the terminal). This would not have been possible using specialization. $Terminal_{ANSI}$ would have had to be implemented from scratch and $Terminal_1 \ldots Terminal_3$ re-implemented as specializations of $Terminal_{ANSI}$. The relationships between the different types of terminals are illustrated in figure 4 below.

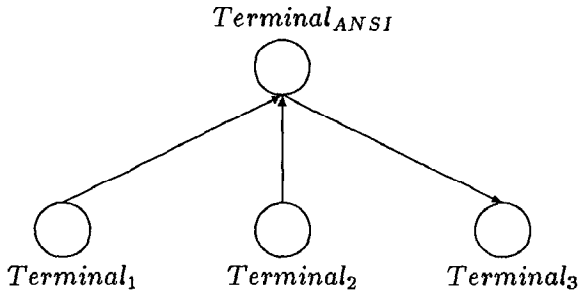

Figure 4: $Terminal_{ANSI}$ is a generalization of $Terminal_1$ and $Terminal_2$. $Terminal_3$ is a specialization of $Terminal_{ANSI}$

4 A generalization scheme in a simple language

We will demonstrate that generalization can coexist with specialization in class-based object oriented programming languages. This is done in three stages. In section 4.1 we discuss interface issues, that is, what methods an instance of a class will have if the class has been created via generalization. From this we verify that generalization can coexist with specialization without introducing conflicts in the class-hierarchy. In section 4.2 we discuss how a class created by generalization can obtain its implementation, that is, its instance variables and code.

In section 4.3 we discuss the problems of name conflicts. We show that the name conflicts in multiple generalization are the same as those encountered in multiple specialization (inheritance) and that they can be solved in the same way.

As a vehicle for the discussions in section 4.1 and 4.2 we will use a language called $\mathcal{SL}$. $\mathcal{SL}$ has a notion of (multiple) generalization as well as specialization. $\mathcal{SL}$ is only a vehicle for a theoretical discussion and is **not** proposed as a new programming language.

4.1 Class interfaces

We will present simple rules that decides which methods an object will have, in a language offering both specialization and generalization. We will use $\mathcal{SL}$ to present these rules. Following this we revisit some of the examples from section 3.

4.1.1 Class interfaces in $\mathcal{SL}$

The subset of $\mathcal{SL}$ relevant to the following discussion is presented in figure 5. It is the subset used to define class interfaces. The subset of $\mathcal{SL}$ relevant to the definition of class implementations is discussed in section 4.2.

<ClassDef>	::=	**class**<Name> **is** {<MethodDefList>|<AdvClassDesc>};
<MethodDefList>	::=	<Name><ParameterDesc> {ϵ |, <MethodDefList>}
<AdvClassDesc>	::=	{<Generalization>|<Specialization>}
<Generalization>	::=	**generalizing** <NameList> **removing** <NameList>
<Specialization>	::=	**specializing** <NameList> **adding** <MethodDefList>
<NameList>	::=	<Name>{ϵ |, <NameList>}

ϵ denotes the empty string.

Figure 5: $\mathcal{SL}$ – syntax for interfaces

The simplest way to define a class is to list all its methods. In $\mathcal{SL}$ this can be written as:

class A **is** $m_1, \ldots, m_n$; [4]

We will denote the set of methods of a class A by $A^{methods}$. So in this case:

$$A^{methods} = \{m_1, \ldots, m_n\}$$

Specialization is described by listing all the super-classes of the new class and by defining new methods that the class will have in addition to those obtained from the super-classes:

class B **is specializing** $A_1, A_2, \ldots, A_k$
adding $m_1, \ldots, m_n$;

This means that class B has $A_1, A_2, \ldots, A_k$ as super-classes and defines the new methods $m_1, \ldots, m_n$. For now, we will assume there are no name conflicts, so if the same name is present in more than one of the classes we can assume it is the same method. $B^{methods}$ will therefore be the union of all the methods inherited from the super-classes plus the newly defined ones.

$$B^{methods} = \bigcup_{i=1}^{k} A_i^{methods} \cup \{m_1, \ldots, m_n\}$$

A class conforms to all its immediate super-classes (for B this is $A_1, A_2, \ldots, A_k$) and all classes to which they conform. So instances of B will conform to B and all B's ancestors in the class hierarchy.

Generalization is described as easily as specialization in $\mathcal{SL}$, however, the classes listed will be those that the new class will have as sub-classes, and the methods listed will be methods to be removed from the interface of the new class:

class A **is generalizing** $B_1, B_2, \ldots, B_k$
removing $m_1, \ldots, m_n$;

Class A is a generalization of the classes $B_1, B_2, \ldots, B_k$. This means that B_i $(i \in 1 \ldots k)$ must conform to A. This again implies that the methods of instances of class A will have to be a subset of those of any instance of class B_i $(i \in 1 \ldots k)$. The way to obtain this is by defining $A^{methods}$ to be the intersection of all the $B_i^{methods}$ minus those explicitly removed. This is expressed formally below.

$$A^{methods} = \bigcap_{i=1}^{k} B_i^{methods} \setminus \{m_1, \ldots, m_n\}$$

In this way we have ensured that B_i $(i \in 1 \ldots k)$ conforms to A. The conformance rules remain unchanged as A is a super-class of B_i $(i \in 1 \ldots k)$ in the class hierarchy.

The symmetry between generalization and specialization in $\mathcal{SL}$ should now be clear. Going "upwards" (towards the super-classes) in the classification hierarchy corresponds to generalization and therefore removes methods compared to the sub-classes. Going down corresponds to specialization

[4] Specification of formal parameters are ignored for simplicity.

and therefore extend the set of methods compared to any super-class.

Any class will conform to all its super-classes. It does not matter whether the sub/super-class relationship has been introduced through specialization or generalization. The conformance rules are still that a class A conforms to a class B if and only if A and B are the same class or B is an ancestor (generalization) of A in the class hierarchy.

4.1.2 Examples revisited - interfaces

In $\mathcal{SL}$ the generalization of class deque to class stack as discussed in section 3.1 would be described as:

class *stack* **is generalizing** *deque*
removing $push2, pop2, top2$;

which (according to the rules above) defines class stack to have the desired interface, namely methods $push, pop, top$ and *empty*.

The terminal example from section 3.2 will be expressed as:

class $Terminal_{ANSI}$ **is generalizing**
$Terminal_1, Terminal_2$
removing $non_1, \ldots, non_n$;

where $non_1, \ldots, non_m$ are the names of the methods that can be used for controlling both $Terminal_1$ and $Terminal_2$ but are not supported by the ANSI standard.

$Terminal_3$ also conforms to the ANSI standard. So it can be described as a specialization of class $Terminal_{ANSI}$:

class $Terminal_3$ **is specializing** $Terminal_{ANSI}$
adding $t_1, \ldots, t_n$

$t_1, \ldots, t_n$ are the control operations supported by $Terminal_3$ that are not supported by the ANSI standard.

4.2 Class implementation

We will describe one way that a class created as a generalization may obtain its implementation. The proposal should only be seen as demonstrating that a solution exists, as the problem can be solved in different ways. We will do this using another subset of $\mathcal{SL}$. We will only briefly discuss implementation of classes defined through specialization (section 4.2.3), as we assume it is already well understood. The relevant subset of $\mathcal{SL}$'s syntax is presented in figure 6

The discussion is divided into two parts. In section 4.2.1 we assume that none of the methods in the class are virtual. A virtual method is a method whose behaviour can be refined in a sub-class. Virtual methods are supported in a number of different languages, for example [3, 5, 6, 14]. In the second half of the discussion (4.2.2) this assumption is removed and we propose a way to solve the problems unique to virtual methods. Finally, we revisit the examples from section 3.

4.2.1 No virtual methods

Assume that class A has been obtained by generalizing classes $B_1, B_2, \ldots, B_n$ and none of the methods of A are virtual methods.

Any instance of B_i $i \in 1 \ldots n$ conforms to A. So an implementation of A can be obtained simply by using any one of the implementations of $B_1, \ldots, B_n$. The compiler/interpreter just has to ensure that only methods belonging to $A^{methods}$ can be executed on instances of A. Implementation of other methods may still be used internally.

In the case of n = 1 the choice of implementation is implicit. If $n > 1$ the system or the programmer can chose the B_j that will provide A with its implementation. We recommend that the choice is left to the programmer for two reasons:

- The programmer has knowledge about the implementation of $B_1, \ldots, B_n$ and is therefore

```
<ClassImpl>   ::=  {<GenImpl>| <SpecImpl>}
<GenImpl>     ::=  implementation <Name> implements <Name> {ϵ;| <ImplPart>}
<SpecImpl>    ::=  implementing <Name><ImplPart>
<ImplPart>    ::=  by <Variables><MethodImplList>
```

Figure 6: $\mathcal{SL}$ – syntax for implementation

more likely to make an optimal choice than the system is.

- In many languages, interface inheritance implies implementation inheritance as well. If the programmer wants to create a specialization of class A in such a language, he will have to know which implementation was chosen for A.

In $\mathcal{SL}$ this is simply expressed as:
implementation B_j **implements** A;

We also recommend that it should be possible to provide a complete new implementation for a class. This can be useful if a more efficient implementation is needed or if the programmer wants to avoid dependencies on implementations of other classes.

4.2.2 How to deal with virtual methods

Assume again that class A has been obtained by generalizing classes $B_1, \ldots, B_n$ but that $a_v \in A^{methods}$ is virtual.

Virtual methods are particularly interesting as they may be refined differently in $B_1, \ldots, B_n$. We want a_v's implementation to express the common behaviour among the implementations of a_v in $B_1, \ldots, B_n$.[5]

We will distinguish three different cases:

1. No common behaviour. In this case the implementation of a_v should be empty. The presence of a_v in A's interface only specifies that specializations of A will all have this method. An example of this is a graphical system where all graphical objects possess a method print. But printing a rectangle is not the same as printing a circle. Class GraphicalObject will therefore have an empty virtual method print, that is differently refined in every specialization of the class.

2. All implementations exhibit the same behaviour. In this case the implementation of a_v can be obtained from the class from which A has obtained its implementation of non-virtual methods. This solution does not work if the programmer wants a less refined (more general) implementation of a_v. See 3 for this case.

3. Some common behaviour. The different implementations of a_v implement different levels of refinement. The programmer will have to provide information for the system to ensure that the implementation of a_v in A describes the common behaviour only.

We suggest that 2 should be the default choice. The programmer can override this by providing an implementation for a_v. This covers cases 1 and 3, as he can provide an empty implementation in case 1.

In $\mathcal{SL}$ we express the case where a different implementation is needed for a_v as follows:
implementation B_j **implements** A **by method** a_v **begin** "statements" **end**;

The problems discussed above are closely related to discussions concerning multiple specialization of virtual methods and how their implementations are combined to express the complete behaviour of all the implementations in the super-classes. A

[5] By behaviour we mean the externally visible effects or specified state changes obtained by invoking the method.

thorough discussion of this problem can be found in [13].

It is worth noticing how the problem is reduced in the case of single generalization. In this case we do not have to consider different behaviour in different implementations. The problem is reduced to case 2.

4.2.3 Examples revisited - implementation

To illustrate the implementation issues discussed above we will briefly review the examples discussed previously.

Obtaining class stack from deque was an example that emphasized the reuse aspect of generalization. It is therefore important to notice that the implementation of stack can simply reuse the implementation of deque without changes. In $\mathcal{SL}$ this is stated as:

implementation deque **implements** stack;

In the terminal example we assume that there is a method *ScreenSize* in both $Terminal_1$ and $Terminal_2$. It will describe different behaviour for the two types of terminals (assuming they are of different sizes). As the behaviour is different for the two types of terminals the programmer has to provide a new (empty) implementation for this method in $Terminal_{ANSI}$. We will also assume that the programmer wants to use the implementation of $Terminal_2$ to implement $Terminal_{ANSI}$.

implementation $Terminal_2$
implements $Terminal_{ANSI}$ **by**
ScreenSize **begin end**;

$Terminal_3$ is an ordinary specialization of $Terminal_{ANSI}$. The programmer will have to provide implementations of the methods added (see section 4.1.2), and refine the method *ScreenSize*.

implementing $Terminal_3$ **by**
ScreenSize **begin** "statements" **end**;
t_1 **begin** "statements" **end**;
t_2 **begin** "statements" **end**;
⋮
t_n **begin** "statements" **end**;

4.3 Name conflicts in multiple generalization

In 4.1 and 4.2 we assumed that name conflicts did not exist. In this paper we will **not** try to solve the problems concerning name conflicts in multiple generalizations, but we will show that solving them for multiple generalization is equivalent to solving them for multiple specialization.

We will discuss name conflicts at a very low technical level. For a more thorough discussion of name conflicts refer to [9].

In the following discussion *name* denotes what identifies a method, such as qualifications and formal parameters.

Assume class A has been created by generalizing the two classes B_1 and B_2, and that this has introduced a name conflict between the two names b_1 from B_1 and b_2 from B_2. We will consider two types of name conflicts between b_1 and b_2.

1. The names b_1 and b_2 can not be distinguished but denotes different methods. In this case neither b_1 nor b_2 should denote a method on instances of A as they are different properties.
2. b_1 and b_2 are different names but denote the same method. In this case some name $b_{1,2}$ must denote this method on instances of class A.

Assume the system supports multiple specialization. If we created class C as a multiple specialization of B_1 and B_2 we would have the same problems. In case 1 b_1 and b_2 must be distinguished by the system so that instances of C have both methods, and two different names will have to be provided to be able to distinguish the two. In case 2

the system will need enough information to know that b_1 and b_2 denote the same method. One solution is to allow both b_1 and b_2 to be used, which would also be a perfectly valid solution for generalization.

Our conclusion is that the information needed to solve name conflicts in a multiple generalization of classes is the same as is needed in the multiple specialization of the same classes. Resolving name conflicts in a multiple specialization scheme is a research issue in itself and is outside the scope of this paper.

5 Conclusion

It has been shown that introducing generalization in class-based object oriented programming languages has several advantages. It extends the reuse aspects of the languages and allows the programmer to create classification hierarchies when similarities between classes are discovered.

Generalization has been discussed solely as a language construct. It may also be an integrated part of a programming environment such as the one provided by Smalltalk [6]. Like other constructs for modularization and reuse, generalization will increase the need for good programming environments. Good tools for browsing are needed to find classes and to make clear the side effects if a class interface/implementation is changed.

By means of a simple example language, we have shown the symmetry between generalization and specialization and how the two can coexist. In addition we have verified that the simple conformance rules from traditional "specialization-only" languages are sufficient in the combined case.

We discussed how a class that is created by generalization can obtain its implementation. This is simple if there are no virtual methods, as any of the classes that were generalized potentially provides an implementation. Virtual methods introduce problems for which a solution was proposed. The solution may require the programmer to rewrite part of the implementation of the virtual methods.

We then showed that the potential name conflicts introduced by multiple generalization are essentially the same as those created by multiple specialization. No solution was proposed to the problems, but we concluded that if they were solved for multiple specialization, then the same solution could be applied to multiple generalization.

Name conflicts will not occur in the case of single generalization. Furthermore the problems concerning the implementation of virtual methods in multiple generalization are simplified in the case of single generalization.

References

[1] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *OOPSLA'86 Object oriented programming systems, languages and applications.* ACM, 1986.

[2] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transaction on Software Engineering*, SE-13(1), January 1987.

[3] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The Beta Programming Language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, Series in Computer Systems, 1987.

[4] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing surveys*, 17(4), December 1985.

[5] Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language.* Norwegian Computing Center, February 1984.

[6] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Company, Xerox Palo Alto Research Center, 1983.

[7] Chris Horn. Conformance, Genericity, Inheritance and Enhancement. In *ECOOP'87 European Conference on Object-Oriented Programming.* Springer-Verlag, 1987.

[8] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA'86 Object oriented programming systems, languages and applications.* ACM, 1986.

[9] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In *ECOOP '88 European Conference on Object-Oriented Programming.* Springer-Verlag, 1988.

[10] Jørgen Lindskov Knudsen and Kristine Stougård Thomsen. A conceptual framework for programming languages. Publication PB-192, Computer Science Department, Aarhus University, Aarhus, Denmark, April 1985.

[11] David Sandberg. An alternative to subclassing. In *OOPSLA'86 Object oriented programming systems, languages and applications.* ACM, 1986.

[12] Alan Snyder. Inheritance and the Development of Encapsulated Software Components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, Series in Computer Systems, 1987.

[13] Kristine Stougård Thomsen. Multiple inheritance a structuring mechanism for data, process and procedures. Publication PB-209, Computer Science Department, Aarhus University, Aarhus, Denmark, April 1986.

[14] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.