Inheritance and Subtyping

Massimo Ancona

Departmento di Matematica
Università di Genova
v. Alberti 4, 16132 Genova (Italy)

Abstract

A method for implementing subtyping and inheritance, as independent features of object-oriented languages, is presented. There are at least two reasons for decoupling subtyping from inheritance: the negative influence on encapsulation and the different perspective of specification inheritance (subtyping) and implementation inheritance (code sharing). In our approach, subtyping and inheritance are defined in two independent constructs: the class module and the implementation module which are separately compiled. Subtyping is declared in the class module which contains all the public definitions of the class (e.g., types and operation signatures), while inheritance is defined in the implementation module, which contains the implementations of all the operations of the class. The class module includes assertions which are an algorithmic form of operation specification.

1 Introduction

Inheritance and subtyping are two of the most important features of object-oriented programming languages. Inheritance is mainly connected with code sharing, while subtyping relates to late binding and polymorphism.

A related concept is encapsulation. Encapsulation is a technique for minimizing interdependencies between separately-written and separately-compiled modules by defining a strict external interface: objects are accessible only through their external operations [7].

If a client of a class has access to the external interface only, then the class could be reimplemented without affecting its clients, providing that the new version implements the same semantic with the same interface.

In most object-oriented languages inheritance and subtyping are strictly integrated in the sense that classes connected with an inheritance hierarchy are considered to be one subtype of the other.

There are reasons for decoupling subtyping from inheritance: one of them has been discussed in [7] and concerns the negative influence of inheritance on encapsulation. A second argument, presented here and also considered in [7,1,5] shows that subtyping and inheritance should be considered independent concepts.

Inheritance is sometimes applied as an extension and sometimes as a specialization [4], and this characteristic cannot be described in terms of the rigid concept of subtyping induced by inheritance. Extension is a technique for code reuse: class B extends the operations defined in class A by inheriting the operations of A and defining additional operations. Specialization introduces the type concept: for example, car is a more specialized concept than vehicle because it has the same properties of a vehicle (a car is a vehicle) plus other properties specific to a car. In other words, there are two different views of inheritance: inheritance of implementation and inheritance of specification (i.e., subtyping) [5].

In this paper we present a method for decoupling subtyping from inheritance. The method is based on the separation of subtype specification and inheritance of implementation in two different and separately compiled modules: the class module and the implementation module. The class module contains the subtype specification and a form of object behavior definition. The object behavior is expressed through assertions, divided in preconditions, postconditions and invariants [4]. The method is discussed and compared with data abstraction, visibility control and encapsulation. The comparison is considered from both points of view of language design and of efficiency of implementation mechanisms. The purpose of the work is to obtain a reasonable compromise between expressive power and implementation efficiency. The method is not difficult.
to implement, and enables the programmer to define arbitrary subtyping relations between classes, in a way that is completely independent of inheritance and encapsulation concepts.

Object-oriented programming is based on the abstract data type model and is defined in terms of objects, classes and inheritance. Objects, which represent the basic notion, are autonomous entities that respond to messages or operations and store a state defined by the value of their internal data (instance variables).

Objects are organized in classes describing their common behavior: objects of the same class share the same operations and the same instance variables [9]. A class describes the implementation of an abstract data type [4]. Objects represent data abstractions because their internal variables are hidden from their clients and are accessible only through the object operations.

In order to combine abstraction with inheritance, some mechanism controlling the visibility of inheritance should be introduced. For example, C++ classes are distinguished from structures (records) for the encapsulation mechanism present in class definition: a class is divided into a public and a private part, the public part specifies the attributes visible from outside. Data abstraction is the ability to define new types of objects whose behavior is defined abstractly, without reference to implementation details such as the data structure used to represent the objects.

Encapsulation is a technique for minimizing interdependencies among modules: a module is encapsulated if clients are authorized, by the definition of the programming language, to access the module only via its defined external interface.

Encapsulation may violate encapsulation in several ways: by letting a class have direct access to internal data of ancestor classes or by letting a client of a class to implement a code dependent upon the inheritance hierarchy of the class.

Reusability is connected with separately compiled modules. In this sense encapsulation may affect at least two levels of entities: the module and the class, in some cases they are coincident (i.e., Eiffel and Trellis), in others they are nested (C++ and Modula-3).

2 Strong Typed Object-Oriented Languages

In almost all object-oriented languages inheritance among classes induces a corresponding subtyping hierarchy. For example, inheritance is bound to subtyping in Eiffel, C++, Trellis/Owl [4,8,6].

Eiffel [4] implements multiple inheritance: a new class is defined as a combination or specialization of existing classes. In Eiffel encapsulation and inheritance are completely orthogonal. A class is free to export or not features inherited from an ancestor class, even though the features are invisible (secret) in the ancestor class.

A class in C++ [8,10] is a generalization of a structure (i.e., a record type) and it is the logic unit of encapsulation. Inheritance is implemented through derived classes. Derived classes are considered a sub-type of the originating base class. Encapsulation is specified through different visibility control mechanisms. The public part of a class specification is the external interface of the class, while operations and variables defined in the private part are invisible to the derived class unless specifically authorized by the friend and protected mode option [10].

A Trellis/Owl program consists of a set of related type definitions organized into type modules. These definitions include operations, components, and fixed names, but not further type definitions. A definition is the smallest unit of source that can be compiled.

The subtyping hierarchy is defined in Trellis/Owl by means of the subtype and inherit clauses. The subtyping hierarchy is defined by the following rules: if type S lists T in its subtype clause, then, T is an immediate supertype of S. In this case, only public and subtype visible operations of T are visible to S, thus only these operations are inherited. Moreover each (public) specification of an operation on T is automatically added to the definition of S.

Operation implementations are inherited similarly to specifications. If an operation F in S also exists (visible) in one or more immediate supertypes, then the implementation of F can be inherited from these supertypes. To avoid ambiguities, there must be just one source of the implementation.

Trellis/Owl and Eiffel introduce subtype compatibility by defining a set of rules which govern the type system of the language. Let S be a subtype of T. For every public instance operation F defined on T, there must be a corresponding public instance operation F defined on S. These two operations must be compatible in number of arguments, type of arguments and results as required by the subtyping hierarchy. The two languages use a different function subtyping: Trellis/Owl is based on a contravariant rule of function subtyping, while Eiffel follows a covariant approach. For more detail on this point see [2].
Similarly to C++, an object in Oberon [12] and Modula-3 [3] is an extension of the record type. Both languages are derived from Modula-2 [11].

The most important feature of Oberon is the extended record type and the associated controls, i.e., type tests, type guards and partial public projections of record types [12]. The extended record type permits the construction of new types on the basis of existing types and establishes a subtype relation between new and old types. No visibility constraints exist inside an extended record, as in a C++ structure: all identifiers defined in an extended record, or inherited from a base type, are public.

An object in Modula-3 [3] is a reference to a data record paired with a sequence of methods, which is a record of procedures (each of them accepting the object as the first argument). Similarly to Oberon, Modula-3 introduces a subtype relation between objects: the subtype relationship is specified by listing, in the object declaration, the supertype name. Subtyping in Modula-3 is defined by a well-defined set of rules involving predefined types, type constructors and user defined types.

The main problems deriving from the integration of subtyping with implementation inheritance are [7]:

- accessing inherited variables safely,
- visibility of inheritance,
- possibility of excluding operations.

Encapsulation requires that instance variables be protected from direct access by clients of a class. More exactly, instance variables that are not part of the interface of a class, cannot be accessed, in descendant classes, like non-inherited variables, i.e., variables defined in the inheriting class. They should be hidden and accessible only through operations exported by the class. A second problem is represented by the visibility of inheritance: if the use of inheritance is part of the external interface then the code of a client class may become dependent on the inheritance hierarchy in such a way that a change in the inheritance hierarchy may render the client code illegal. Another problem is represented by the possibility of excluding operations from inheritance: this possibility makes the subtyping or is-a relationship associated with inheritance inconsistent. There are several ways to violate encapsulation through inheritance: a class may directly refer to instance variables or to private operations of a parent class or access to features of ancestor classes not exposed in the external interface of the ancestor class.

3 Extension and Specialization

"Inheritance is sometimes viewed as extension and sometimes as specialization" [4].

The two different views of inheritance depend on whether a class is considered a type or a module. In the first case, inheritance represents the is-a relationship and is considered as specialization; if B is heir to A, the objects that may be associated with an entity of type A are instances of A, of B and of its descendants.

From the module perspective, a class is viewed as a provider of services: B implements the operations of A plus its own operations. The operations that can be applied to instances of A are a subset of those applicable to instances of B.

"Inheritance is specialization from the type point of view and extension from the module viewpoint" [4].

From the above considerations the two concepts should not be integrated into a unique construct: the combination of different properties in a single construct is of dubious value, causing ambiguities in the implementation and unexpected side-effects.

Inheritance is connected with the implementation of the classes, while subtyping hierarchy is based on the behavior of the instances [1]. In fact, it is possible to define a class that specializes the behavior of another class, but employing a completely different implementation based on different code (even for methods with the same name), so there can be subtyping without inheritance [1].

The opposite situation (i.e., inheritance without subtyping) is also possible, inheritance without subtyping: for example suppose to have a large class A, that for modularity purposes has been divided in several small classes A1, A2, ..., An, which are independently implemented. Class A could be defined as a class inheriting all the subclasses Ai (if the language allows multiple inheritance). In this case we obtain the unwanted (and often meaningless) side effect that A is a subtype of each Ai.

Consider the following example [7]. Two classes Stack and Dequeue are defined. Stack is a standard lifo queue and Dequeue is a queue implementing both the lifo and fifo strategies (i.e., elements can be inserted and extracted from either end of the queue). The external interface of Stack is a subset of the interface of Dequeue which includes two additional operations for entering and extracting elements from the rear of the queue. Stack could be implemented through inheritance from Dequeue by excluding the extra operations:
in this case Stack inherits the implementation of De-
queue but is not a subtype of Dequeue, as it does not
have all the Dequeue operations.

The following example shows a more complex re-
lation connecting inheritance with subtyping. Sup-
pose we define a class Ext.Int implementing an ex-
tended precision integer arithmetic and two descendant
classes Ext.Real and Ext.Complex implementing multi-
precision floating and complex arithmetics. We can
obviously implement Ext.Real as a client (or heir) of
Ext.Int and Ext.Complex as a client of Ext.Real, be-
cause the arithmetic operations of each class are imple-
mented by using the operation defined in the preceding
class. However the correct (mathematical) subtyping
hierarchy is completely reversed: Ext.Int is a subtype
of Ext.Real which, in turn, is a subtype of Ext.Complex.

In standard object-oriented programming languages
(e.g., Eiffel and C++) this problem can be avoided
by defining Ext.Real (resp. Ext.Complex) to be both
a client and a supertype of Ext.Int (respectively of
Ext.Real). A more direct (and clear) solution, obtained
by separating subtype definition from inheritance, and
defining two separate hierarchies.

Finally, three important concepts to be mentioned
are redefinition, renaming and virtual operations. While redefinition is a semantic mechanism, renaming
is only a syntactic feature [4] allowing to identify the
same entity with different names in different contexts,
which may be combined in various ways [4]. Renaming
is an alternative to name qualification, useful to avoid
name clashing in multiple inheritance.

Virtual [8] operations represent a powerful mecha-
nism for implementing software reuse, late binding and polymorphism. A virtual operation is the specifi-
cation of a routine that can be redefined in descendant
classes[8]. This concept plays a fundamental role in our
approach.

4 The Method

Our solution of the problem is based on the separa-
tion between subtyping specification and inheritance
of the implementation. The subtyping specification is
defined in a class module (also called class declara-
tion), while the implementation (containing the inheri-
tance clause) is introduced in the corresponding imple-
mentation module (to avoid syntactic details we use a
Modula-like syntax paired with an informal semantic
presentation). The subtyping specification is defined
by listing the supertype as a parameter of the class
(we adopt a notation similar to the Oberon syntax for
defining record extensions [12]). For example:

\begin{verbatim}
class B(C);
procedure P(...);
function F(...):..;
function H(...):...;
end B.

class A(B);(* A is a subtype of B*)
(*specification of class A*)
procedure Q(...);
function G(...):...;
end A.
\end{verbatim}

\begin{verbatim}
module A;
(*implementation of class A*)
of B use P;
of D use F;
procedure Q(...);
begin
....
end Q;
function H(...):...;
begin
....
end H;
function G(...):...;
begin
....
end G;
....
end A.
\end{verbatim}

Specifies that class A is a (direct) subtype of class
B. The subtype mechanism is the following: A inherits
from B the external interface, i.e., all attributes (types
definition and operations signatures) specified in class
B (in the example operations P, F and H). Subtype
inheritance is implicit, i.e., all inherited entities may
directly referenced by name and the types and opera-
tions of all ancestor classes (C in the example) become
transitively visible.

Furthermore, only the specification of each opera-
tion listed in the external interface of B (and C), is
inherited, while the corresponding implementation is
excluded. In other words, only the signature of each
operation belonging to the class module B is included in the interface of A: operations are inherited as virtual and deferred operations (because inherited only from the class specification). Their implementation could be independently inherited via the use clause, in the corresponding implementation module, or they could be reimplemented in the implementation of the class A (module A). Operation specifications inherited through the subtyping hierarchy and not inherited nor redefined in the implementation are implicitly inherited from the supertypes. In either case, clients of A are not informed on the implementation inheritance defined in A: they only know that A and B are connected by a subtype relationship, as specified in the external interface of A.

In the above example, operation H is reimplemented in module A, operation P is inherited from B, and F is inherited from class D. In this latter case, the signature of the inherited operation must be defined in a common ancestor type of A and D (for type compatibility). Inheritance here means inheritance of the operation implementation.

Our approach provides three possible layers of visibility: visibility of subtyping specification, visibility of inheritance of implementation, visibility of methods to clients of a class.

Based on the consideration that in statically-typed languages subtyping rules are of critical importance [4] (because they determine the legality of programs), we decided to introduce a single layer of visibility for all clients of a class: only the subtyping specification, i.e., the public external interface of a class, is visible. Inher-
ited attributes, not included in the external interface of the class, are hidden to client modules.

This approach provides a more effective implementation of encapsulation. In fact "A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface. Encapsulation thus assures designers that compatible changes can be made safely, which facilitates program evolution and maintenance" [7].

The separation between subtyping and inheritance of the implementation and the specification of inheritance in the private (non-visible) part of a class avoids the above discussed problems. Clients of a class are excluded from accessing private information of a class and do not know how an operation is implemented: (whether it is inherited or not and from which class). Clients of a class are informed only on the subtyping hierarchy of classes, without having any visibility of the corresponding implementation and of inheritance hierarchy.

Assertions

A type can be considered a collection of objects sharing common properties, represented by operations that can be performed on them [1,2]. "The type T associated with a class B is the type comprising all the instances of B. A type S comprising all the instances of A and B together is a supertype of T" [1]. In other words, a type could be considered the specification of object behavior. Its definition, in practical programming languages, should be as close as possible to a formal specification (ADT).

Conformably to the ADT paradigm, after the definition of sorts and the operation signature some form of operation specification must be provided. In ADT theory, this is expressed in axiomatic form (e.g., equations or Horn clauses). In practical programming languages, operation specification could be expressed via assertions [4], or in some algorithmic form close to the semantic description of operations.

Assertions can be implemented via boolean expressions which make a class declaration as soon as possible conformant to the corresponding abstract definition. Though, only a restricted part of the abstract specification may be expressed in this form, yet this method has the advantage of narrowing the gap existing between abstract specifications and concrete implementations. Assertions [4] can be divided in:

- **preconditions**: i.e., conditions that must be satisfied at the activation of an operation (method);
- **postconditions**: assertions that must hold at the exit from an operation, and
- **invariants**: that must be always verified during a class operation execution.

The following example shows how the class STACK defined in [4] could be described with our approach (for assertions we use the notations used in Eiffel [4]):

```
class STACK;
    procedure PUSH(x: Eltype);
    require not FULL;
    ensure not EMPTY; TOP=x;
    function FRONT: Eltype;
    function EMPTY: Boolean;
    function FULL: Boolean;
    function TOP: Eltype;
    procedure POP;
    require not EMPTY;
    ensure not FULL;
end STACK.
```

```
module STACK;
    procedure PUSH(x: Eltype);
    ensure nb.elems = old nb.elems + 1;
end ...

function FRONT(Q: LIST): Eltype;
begin ...
end;

function EMPTY: Boolean;
begin ...
end;

function FULL: Boolean;
begin ...
end;

function TOP: Eltype;
begin ...
end;
```
end;

procedure POP;
begin
    ....
    ensure nb.elems = old nb.elems - 1;
end;
    ....
end STACK.

The above example shows two kinds of assertion: abstract assertions which depend only on the abstract data type definition, and implementation-dependent assertions. Assertions of the first kind belong to the (abstract) specification of the class. Assertions of the second kind must be inserted in the implementation module (usually, they depend on state variables which are not visible in the class definition). In the above example, the assertions ensure nb.elems =... are implementation dependent, in fact, nb.elems is relative to a specific implementation of STACK.

An exception mechanism could be associated to each assertion in the class implementation. The use of assertions in the class definition is useful to maintain the semantic of reimplemented operations, conforming to its original specification.

5 Conclusions

Inheritance and subtyping are two different concepts that should be independently implemented and not unified in a single construct as it is in most object-oriented languages. An approach to their independent (though integrated) implementation has been proposed. Arguments supporting such a decision have been compared and discussed. The proposed approach combines the advantage of maintaining the simplicity of methods integrating both concepts in a single construct, with the flexibility offered by two independent constructs. The execution cost of late binding is the same: an extra level of indirection in routine calls, while the added complexity of the compilation process remains acceptable. This approach, when compared with the traditional method, has the advantage of introducing more flexibility in the language as well as a stronger encapsulation and abstraction mechanism.

Our present implementation allows a strictly hierarchical subtyping (single subtyping) and multiple inheritance. The redefinition of an operation signature and associated assertions in a class definition is not allowed. This fact introduces useful constraints on the implementation semantics.

References