Do We Need Inheritance?

Wolfgang Weck
ETH Zürich
Weck@inf.ethz.ch

Clemens Szyperski
Queensland University of Technology
c.szyperski@qut.edu.au

Introduction

Object-oriented programming rests on four pillars: polymorphism, late binding, encapsulation, and inheritance. Of these, inheritance is the one most controversially discussed. Various problems have been stated, that occur when inheritance is used for object composition, modification, and extension. These problems are caused by the possibility to override single methods of the inherited object, thereby changing it rather than reusing it as a whole. This possibility contradicts actually the requirements of encapsulation.

Note that in this context delegation (as used in object-based language such as Self [US 87]) is equivalent to inheritance [e.g., Stein 87] and introduces the same problems. Overriding of individual methods then corresponds to implementing individual methods while delegating others. Forwarding is a weaker mechanism and does not have the problems of inheritance or delegation; for a simple statement on how forwarding and delegation differ, c.f. [JZ 91].

Already in 1987 Pierre America discussed that subtyping and subclassing should be treated separately. Furthermore, he stated that both concepts may loose importance when concurrent object-oriented programming is considered [America 87]. In the proposal at hand we go one step further: inheritance may be of little or no importance when object composition is considered in the context of distributed objects and components.

Also, the role of subtyping becomes doubtful in such environments. The ability to build linear hierarchies is not sufficient for object composition on a larger scale: it must be possible to integrate properties defined in separate components. Thus, a way to aggregate types may be of more value than the ability to define subtypes.

Recent development introduced alternative concepts, for instance Java's interfaces [Sun 95] or the older protocols in Objective C [NextStep 93]. Such concepts are offered as complementary or alternative constructs, not as substitutes for the classical, hierarchical constructions. Compared to real multiple inheritance, as for instance supported by C++ or Sather [SOM 94], the combination of interfaces and hierarchical inheritance may appear restrictive. However, these trials to harness the overmighty inheritance construct are really indications in favor of our line of arguments.

At the same time, Microsoft's *Common Object Model* (COM) [Brockschmidt 95] relies entirely on non-hierarchical, non-extensible, immutable interfaces. Neither inheritance nor subtyping are supported. Objects can be composed using either containment or aggregation, similar to forwarding.

COM is a low level object composition standard but not a programming model at the language level. Currently, compilers are expected to bridge the gap between inheritance-based programming languages and COM. An alternative would be to make COM's model of interfaces and implementations available at the language level, thereby substituting inheritance and possibly subtyping. An example of a compiler adding COM-support to the language Oberon (although without abolishing Oberon's subtyping and inheritance constructs) is Oberon microsystem's Direct-To-COM Compiler [Om 96].

Here is what we need to investigate: can object composition replace inheritance and can interface aggregation replace subtyping? What other forms of polymorphism would still be required? To what extend does static typing suffer or gain? What further mechanisms are required?

Inheritance Breaks Encapsulation

In the following we review briefly, why we think that abolishing inheritance may be worthwhile. Actually, nothing is wrong with inheritance as such. If an object inherits state and methods from another one, it may add additional things as long as it leaves all the inherited stuff alone. This means that no methods must be changed and that additional methods must not change the internal state of the base object.

Unfortunately, this is not how inheritance is viewed and how it is being used currently. The ability to replace entire methods and to manipulate the inherited state is viewed as an essential ingredient of inheritance.

The purpose of encapsulation is to create abstractions and to protect them. The behaviour of an object is abstracted by a specification. The latter serves as an interface between two worlds: the object's internals must be defined in a way such that they meet the requirements set up by the specification, and the users of the object can rely on the specification but need not to worry about its concrete realization. Apart from imposing a structure on larger systems this mechanism gives object-oriented programming its power: a program can use any object that meets the required specification!

An object, designed to meet a certain specification will maintain certain invariants. Encapsulation is the tool allowing to protect these invariants from destruction through improper use of the object. This is the reason for hiding the object's internal state from the outside and for making it accessible through methods only. (Another reason is to fully hide representation details, such that an object can represent state in the most appropriate way.)

Obviously, all these guarantees vanish at the very moment in which it becomes possible that any of the methods restricting access to internal state can be changed or that the internal state can be manipulated by additional methods. Not only the object may stop to meet its specification, existing methods may break entirely and abort if invariants they rely on do not hold any more.

To be able to correctly override a method of an object, the complete internals of the object must be understood. At this point the abstraction character of an object is given up. A number of recent proposals aim at solving this problem to some extend (eg, [Lamping 93] or [PMG 95]). However, the problem has not yet been fully solved and the stream of incremental remedies continues.

The problem has an additional dimension: time. One purpose of encapsulating software and abstracting it is that the implementation might be changed later without invalidating the clients, as long as the specification is not being changed. If an inherited object is changed in that way, it is likely that the overriding methods also must be changed, since some internal invariants may change. This problem is known as the semantic fragile base class problem.

Bottom line: the problem with inheritance is that it contradicts the idea of encapsulation of an object's implementation.

Inheritance Considered Harmful

So far, it has been shown that one cannot have both true encapsulation and unrestricted inheritance together with overriding. A decision has to be made which way to follow.

Boris Magnusson claimed that inheritance shall not be used just to reuse code from existing objects [Magnusson 91]. Rather, inheritance should be a modelling aid to express relationships between different objects. This modelling aid can be found in subtyping, separated from subclassing.

Furthermore, it has to be recognized that the specification (or contract, as it is often called) of an object consists not just of the signatures of its methods, but also of assertions, e.g. of pre and post conditions of method calls. (This is the reason for Eiffel to support assertion and pre/postcondition checking [Meyer 92].) Hence, to really support modelling, abstractions and encapsulation are essential.

On the other hand inheritance appears not to be essential. The important structuring method is to offer objects as building blocks for later composition. In contrast to the ad hoc reuse of (parts of) objects, which has been criticized by Magnusson, use of objects as complete entities does not affect their integrity. An object may be embedded in the larger context of another object by forwarding, aggregation, or the like.

Inheritance is needed only to reuse object parts not beeing meant to be reused. If a part of an object can be used independently and if the designer of the object wants to allow for reuse of this part, it can be implemented as a separate object. Truely reusable material requires that the reuse is anticipated [JF 88]. This will be reflected by the separation into individual objects.

Bottom line: inheritance is needed only to break laws that have been set up by the original object's designer.

Which Hierarchies Do We Need?

Taking the separation of specification and implementation literally, lends itself to separation of subtyping and inheritance. Sather [SOM 94], for instance, has gone this way. In Sather, inheritance is by definition equivalent to textually copying the inherited material. Multiple inheritance has the semantics of sequentially copying and possibly renaming inherited material. On the specification side, multiple subtyping is supported.

Building hierarchies makes sense for typing: an object may have all properties specified by a particular type plus some extra ones. Furthermore, individual objects can have properties associated with different types. They would implement all of them. This is expressed through multiple subtyping.

A more radical approach is being introduced by Microsoft's COM. With COM, the specification side allows for no hierarchies either. What is left, is that any object can implement an arbitrary number of interfaces. Interfaces are non-extensible types.

To allow for reuse of existing objects, containments and aggregation are used. Both are related to forwarding. It is their essential goal to keep the encapsulation intact, when reusing an object.

The *is-a* relation between types is gone. One is left with an *implements* relation between an object and an interface (or type). The *is-a* relation can be computed statically, without considering an implementation, i.e., an object. The *implements* relation can only be decided for a given object. Considering late binding and polymorphism, it is obvious that the *implements* relation must be decided dynamically. (In COM it is implemented by a standard method, allowing to ask an object for a particular interface.)

This dynamical approach is feasible for a standard for object interaction. It is assumed, that COM will not be programmed at this abstraction level. A compiler will generate the required code. Static checks and assertions have to be done at the programming language level, therefore COM does not deal with these itself.

We consider it an open question how COM-like technology can be embedded at the programming language level. Obviously, more static checking is necessary than COM offers. Therefore, some relation between variables (not objects!) and interfaces must be introduced. It is not clear yet, whether this should be full subtying in its known form, or whether a simpler possibility exists. The latter can be imagined as a form of multiply typed variables: a variable ensures that the object it holds implements a certain set of interfaces, not just one interface.

In any case, some degree of dynamic checking will remain. This can be covered by interface case analysis, similar to type case statements used in several languages.

A First Indication Of Feasibility: INNER Can Be Substituted

The following shall give a first indication that interfaces and object aggregation may be a sufficient substitute for inheritance. To do so, we present how inheritance as used in Simula [DMN 68] and Beta [MMN 93] can be expressed using interfaces. The resulting pattern is slightly less powerful, but it has to be: it can no longer break encapsulation.

Simula and Beta do not allow for total overriding of methods. Instead, the base method must contain an INNER statement to explicitly allow for inserting code at this point. (A method without an INNER cannot be changed at all.) The base method will be executed in any case, but it can be extended by statements to be executed in place of INNER.

Firstly, this type of inheritance can be implemented using virtual methods, maybe involving little syntactical overhead: the INNER is simply replaced by a call to the virtual method. Still, combining virtual and real methods in one specification is a concept related to inheritance. The implementation of the virtual method has complete access to the internal state of the object. A second transformation step is required.

True encapsulation can be reestablished if two objects are used. These must be bundled together. The first object corresponds to the original base object. It refers to the second object which implements the method to be up called. This second object needs also a reference to the first one to call its methods, but it cannot access its internals.

It is an open question how exactly the object bundling should work. The bundling can be established explicitly through mutual registration via method calls, sometimes called Twin Objects [Mössenböck 93]. This is a bit cumbersome. It would be preferable if the bundling could be expressed statically, like inheritance.

Functional Building Block Composition

Above we claimed that inheritance is not required as an object composition tool. In the following, this shall be supported further for the important special case where object composition is used to actually compose larger building blocks.

Frameworks are collections of classes representing the knowledge how to solve problems of a certain kind (e.g., [Deutsch 89]). Most of the time, objects are not composed individually but in the context of frameworks. Object composition is the main tool to specialize or extend frameworks.

Similarly, components for object environments, like OpenDoc [FM 96] or OLE 2 [Brockschmidt 95], are collections of objects. Also, component composition is based on composition of individual objects.

Frameworks and components present functional building blocks. They implement entire concepts and offer them for reuse. These building blocks are designed with this purpose in mind. They are prepared for reuse of a particular kind. They explicitly provide the necessary extension hooks.

Analysis of the design of extensible frameworks, for instance the comprehensive component framework Oberon/F [Om 94], shows that three patterns of object composition seem to be sufficient. The patterns relate to individual methods in the first place but the classification can be extended to specifications of entire objects.

Abstract Methods: an abstract method is specified to be implemented (always and entirely) by the extender of the framework. The framework itself does not contain any implementation of such a method. In fact, an abstract method is a specification rather than a real method. Abstract methods are used to allow the framework to up call external code, i.e., the implementation to be bound later.

Default Methods: the framework may specify a method and offer a default implementation at the same time. The default can be used as is or it can be replaced entirely. Empty methods are a special case. They are used when an object's activity is optional. (The default is not to perform the activity.)

Base Methods: a base object implements some functionality which may be extended later. The method defined by the base method must be executed in any case, but further activity may be added.

With inheritance, all these patterns can be implemented: abstract methods are being overriden by the subclass presenting the implementation, default methods are overriden only if the default is not used and inherited otherwise, base methods are extended and a super call is embedded.

It is suggested to investigate how all three patterns can be implemented without inheritance but based on interfaces. The first case is obvious: interfaces correspond to classes with abstract methods only. The second case needs to be implemented by two separate parts: firstly, the interface giving the specification to be met by any implementation; secondly the default method, such that it can be bound to the interface if wished. The third case requires to provide an extra extension hook: the base method is implemented completely but calls an yet abstract method. The latter is implemented to add further activity. (The pattern corresponds to the replacement of Beta's INNER discussed above.)

Implementing the discussed patterns in an inheritance-free environment will prevent abuse of objects since the latter can only be used as designed. At the same time, the implicit documentation is improved. It is clear what the extender of a framework needs to implement: the abstract methods.

Questions To Be Addressed At The Workshop

The question whether and how to do away with inheritance and method overriding in practical work should be addressed on different levels.

From the perspective of building block composition: can the three general design patterns that extensible systems are based on (i.e., abstract method specification and implementation, default methods, and method specialization) be expressed without inheritance and method overriding? Which alternative constructs are needed for this purpose?

From the perspective of composition rules and language design: restriction to plain forwarding eliminates implicit recursion across extended and extending objects. If recursive couplings have not been foreseen and therefore allowed for explicitly, then such couplings cannot be constructed. Inheritance and delegation both allow to construct implicit recursive couplings, but it is just this power that breaks encapsulation; at least with the current state of the art in inheritance and delegation techniques. Could we do without this power, or do we need to find a construct that better harnesses recursion?

From the perspective of design patterns: can inheritance-free implementations be found for the design patterns in one of the emerging comprehensive pattern collections, for instance [GHJV 95, Pree 95]?

Conclusions

We suggest to abolish inheritance and possibly subtyping, at least in its present form. Inheritance presents enough disadvantages to justify such an attempt. Instead, the design principle could be separation of interfaces and implementations in a COM-like style. Indications that this is possible are the mere existence of COM and preliminary attempts to express common inheritance patterns that way. The latter have been presented in this paper. The major open question is whether and how the low level COM design can be lifted to the programming language level.

A concrete research plan involves definition of an experimental programming language as the first step. In the second step various design patterns will be expressed using this language. In a third step the language could be implemented and used to construct some real world frameworks.

References

[America 87] P. America, "Synchronizing actions," Proceedings, EOOP'87, June, 1987.
[Brockschmidt 95] K. Brockschmidt, Inside OLE (2nd ed.), Microsoft Press, 1995.
[Deutsch 89] L. P. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," in: Software Reusability, Vol. 2 (T. J. Biggerstaff, A. J. Perlis, eds.), ACM Press, 1989.
[DMN 68] O.-J. Dahl, B. Myrhaug, K. Nygaard, SIMULA 67 Common Base, Norwegian Computer Center, Oslo, Norway, 1968.
[FM 96] J. Feiler, A. Meadow, Essential OpenDoc, Addison-Wesley, 1996.
[GHJV 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns," Addison Wesley, Reading, MA, 1995.
[JF 88] R. E. Johnson, B. Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming, 1:2, June 1988.
[JZ 91] R. E. Johnson, J. M. Zweig, "Delegation in C++," Journal of Object-Oriented Programming, 4:3, November 1991.
[Lamping 93] J. Lamping, "Typing the Specialization Interface," Proceedings, OOPSLA'93, October 1993.
[Magnusson 91] B. Magnusson, "Code Reuse Considered Harmful," (Guest Editorial) Journal of Object-Oriented Programming, 4:3, November 1991.
[Meyer 92] B. Meyer, Eiffel - The Language, Prentice Hall, 1992.
[MMN 93] O. L. Madsen, B. Moller-Pedersen, K. Nygaard, Object-Oriented Programming in the BETA Programming Language, Addison-Wesley, 1993.
[Mössenböck 93] H. Mössenböck, Object-Oriented Programming in Oberon-2, Springer-Verlag, 1993.
[NextStep 93] NextStep, Inc., Object-Oriented Programming in the Objective C Language, Addison-Wesley, 1993.
[Om 94] Oberon microsystems, Inc, Oberon/F Tutorial and Reference, Zurich, Switzerland, http://www.oberon.ch/Customers/omi, 1994.
[Om 96] Oberon microsystems, Inc, Direct-To-COM Compiler for Oberon/F, Zurich, Switzerland, http://www.oberon.ch/Customers/omi, 1996.
[SOM 94] C. Szyperski, S. Omohundro, S. Murer, "Engineering a Programming Language: The Type and Class System of Sather," Proceedings, International Conference on Programming Languages and System Architectures, LNCS 782, March 1994.
[Stein 87] L. A. Stein, "Delegation is Inheritance," Proceedings, OOPSLA'87, October 1987.
[Sun 95] Sun Microsystems, Java Language Specification, http://java.sun.com, file /JDK-beta2/psfiles/javaspec.ps, first released in 1995.
[PMG 95] MIT Programming Methodology Group, Theta Language Specification, http://www.pmg.lcs.mit.edu/Theta.html, 1995.
[Pree 95] Wolfgang Pree, "Design Patterns for Object-Oriented Software Development," Addison Wesley, 1995.
[US 87] D. Ungar, R. B. Smith "Self: The Power of Simplicity," Proceedings, OOPSLA'87, October 1987.