Inheritance Using Contracts & Object Composition

Wolfgang Weck

Turku Centre for Computer Science (TUCS) & Åbo Akademi, Turku, Finland

Abstract. Normal class-based code inheritance across component boundaries creates a dependency between the involved components. To avoid this, a specification of the inherited class must be part of the respective component's contract and the inheriting class must be specified with reference to this specification only. With this, inheritance can be replaced by object composition without sacrificing the possibility of static analysis, yet being more flexible.

1 Introduction

One of the distinguishing properties of component-oriented programming is the notion of *late composition*. This is to say, that component manufacturing and component compostion are two separate steps, carried out one after the other. During component manufacturing, other components are refered to by interfaces, or contracts, only. Actual implementations are selected at composition time [12].

Object-oriented programming is a foundation technology for components. A typical component will specify a couple of classes or objects. To access services, other components will obtain objects from the providing component and send requests to them. In a running system, the hierarchy of components is complemented by a mesh of objects. These objects and the references between them are constructed, changed, and destructed at run time.

Object reuse and modification is a key tool for component reuse. In this paper we investigate language support for inheritance across component boundaries under the aspect of late composition.

We do, however, not discuss the semantical problems of inheritance, such as the fragile base class problem. We are interested solely in technical support of late composition. The semantical problems of inheritance can be treated separately, since our proposals apply also to various kinds of disciplined inheritance. In the extreme, the compiler may restrict the power of inheritance to that of forwarding.

2 Object Composition versus Class Composition

Two notions of classes and inheritance exist in the object-oriented programming community. Many programming languages and their underlying models are *class-based* (e.g. Eiffel, C++, Java, or Smalltalk). Others, such as Cecil [5] or Self [16] are *prototype-based.*

Class-based approaches abstract from the many instances of objects by grouping them into classes. All the objects of one class have the same attributes, accept the same messages, and exhibit the same behavior. Every new object is created as an instance of some specified class, and it will remain an object of this class throughout its life time.

With prototype-based approaches, objects are created by cloning an existing object, the prototype, and modifying the clone. Here, classes are sometimes seen as a dynamic equivalence relation that can be infered at run time. By modifying an object's state, or structure, objects can be migrated from one class to another at run time. This approach has the advantage of being more flexible. It allows to change an object's behaviour or to assign class membership via predicates on the state [4].

The flip side of this flexibility is that static checking and reasoning becomes almost impossible. It may not even be clear, which messages a given object accepts, unless its complete history is examined. In a modular environment, this makes static analysis very difficult. Still, in a closed system, complete flow analysis may allow to make up for this [1], but in an extensible system this is not possible anymore [15].

With object-oriented programming, it is common to express composition and reuse by means of *inheritance*. In short, some inheriting entity inherits from one or several inherited entities by copying their implementation. Additionally, the inheriting entity may specify some modifications of the copied material. In principle, inheritance is equivalent to copying and modifying source code.

The above two views on object-orientation use inheritance between different kinds of entities. Class-based approaches support inheritance between classes, whereas prototype-based approaches support inheritance between objects. The latter is, for instance in Self [16], implemented through reference to a *parent object*, to which the handling of unknown messages is delegated.

Class-based approaches fix the inheritance relations at compile time. Since inheritance relates to implementation, class-based inheritance fixes the implementation to be inherited at compile time, i.e. before composition time in a component-oriented context. This makes state-of-the-art class-based inheritance unsuitable across components, because we want to delay the selection and binding of the base-class implementation until composition time.

We can conclude that, depending on the school you follow, you will get from object-oriented programming either support of static analysis or the possibility to compose implementations later than at compile time, but not both. A question of interest is, whether a middle ground can be found, on which you get both static analysis and late composition of implementations. Such a middle ground would be necessary for component-oriented programming to allow for inheritance across components.

3 Contracts

On the component level, the above dilemma between static analysis and late composition is well known. There, the answer is the definition of contracts, which specify the obligations component providers must meet and the expectations component clients may have. For every component, it must be documented according to which contracts it offers or requires services. Two components can be composed, if one offers services that are requested by the other component according to the same contract. Each component can be analysed separately, based on the contracts it participates in. At composition time, one only needs to check whether the two components actually claim to stick to the same contract. If they don't, the composition can be rejected.

In short: at compile time only specifications are bound, whereas implementations are bound at composition time. The contract provides the necessary separation of the specification from the implementation.

The practical effect of this separation is that static reasoning is still possible, because the yet unknown partner can be substituted by the contract. Still, bindings between implementations are established only at composition time, retaining full flexibility of selecting components to the composer. Thus, with components, we managed to eat the cake and have it too.

4 Class Composition With Contracts

The same technique can be used with inheritance. Instead of refering to an implementation, the inheriting class refers to a contract only. The contract states what to expect from the inherited class; the inheriting class can be statically analysed. We get the safety we want.

Only when an object is instantiated, the binding to a concrete base class implementation must be established. Any class meeting the required contract can be bound. Like with component composition, the contract can be used to check at composition time, whether the specific composition is feasible. In addition to safety, we get the late composition we want.

Note that it is because of the separation between specification and implementation that the semantical problems of inheritance become so acute with component-oriented programming. The inheriting class must simply be able to cooperate with *any* base class that meets the specification. In this paper, however, we postulate specifications to be detailed enough and/or one of the many approaches to disciplined inheritance to be in place.

Class inheritance with contracts is implemented in IBM's SOM and in modular object-oriented programming languages, such as Modular Smalltalk [19] or Oberon-2 [9], an extended version of Oberon [18]. These languages feature separate constructs for modules and classes. Modules are separately compilable, similar to Modula-2. It is possible to compile several modules implementing the same interface. This allows for alternative implementations of the same specification.

In Oberon-2, for instance, classes are implemented as record types. The latter are extensible inside as well as outside the module in which they are defined. Furthermore, procedures can be bound to such record types. Such type-bound procedures resemble methods. In an extending type, they can be redefined, otherwise they are inherited from the base type.

Exported record types resemble contracts for classes, because module interfaces contain only link information to be exploited at binding time. To avoid confusion, note that the contract syntactically consists only of signatures. However, we postulate some semantics being attached, e.g. as a comment, i.e. we use types in the spirit of behavioural types [8]. This semantics specification just happens not to be checked by the compiler.

New classes can be specified, programmed, and compiled refering to a base type. This happens whenever a record type is extended in a separate module, because compilation relies on the imported module's interface. The implementation is bound at load time only and therefore needs not to be selected earlier.

With modular object-oriented programming, only one component (module) per contract can be used in a running system. As a consequence, all classes meeting a given contract have the same implementation. It is just that this implementation is selected very late.

Still, this has been used effectively for system refactoring, done one module at a time, as long as the module's interface had not to be changed. In some cases of modules implementing device drivers, alternate implementations of a single contract were provided to allow adoption to different hardware. (These modules seldomly actually define classes, but they could do so.)

5 Can We Go Further?

Can we get rid of the aforementioned restriction and support different implementations of a contract simultaneously? One could allow several modules implementing the same interface to coexist in a running system. This would collide with some assumptions being generally made about modules. Also, we would need to find a way to refer to the different module implementations. Currently, identifications are made by refering to the module name, i.e., the contract, which is implemented by exactly one module.

Another, probably better, approach could be based on separating subclassing from subtyping, as for instance done in Sather [11]. Types would be employed as contracts whereas classes would be bound as implementations later. At component manufacturing time, subclasses would be specified by refering to a type instead of to a class. (Sather does not support this to avoid the semantical problems of inheritance. It rather defines class inheritance to be exactly equivalent to textual inclusion and application of text editing operators. Obviously, this binds an implementation at compile time.)

With this, the base classes to be used with an object would need to be specified at object allocation time; the allocation procedure would need to accept the respective additional parameters.

To make this useful, we would have to turn classes into first order objects. Otherwise, the programmer of a component that contains a creation statement would have to wire in which implementation to use. This would again create a dependency between components, this time between those containing the creation statement and implementing the base class.

6 Object Composition With Contracts

This can be taken one step further by composing objects instead of classes: one can specify a base object instead of a base class when allocating an object.

If a contract is used to specify statically the properties required from the parent object, and if the parent object is bound for ever when the refering object is created, static analysis is possible to exactly the same degree as before, i.e. as with class-based inheritance with contracts. Though each object may be of a class of its own now, the same information as before is available from the object itself and the contract specifying the parent.

To retain the full amount of static information as with class inheritance, we must prohibit to re-assign the parent or base object. Otherwise, unexpected changes of state and/or behaviour of the composed object could occur.

This construction is indeed on a middle ground between static class inheritance and full dynamic inheritance as used with prototype-based object-orientation. Compared to the former, we gain flexibility, even more than with class composition based on contracts. Compared to prototype-based object-orientation, we get more static information because of two restrictions. First, only objects satisfying the required specification can be used as a parent. Second, the parent, once assigned, cannot be changed anymore.

Still, we get much of the flexibility of prototype-based object-orientation. For instance, the user can interactively specify a parent object to be wrapped. An example are wrappers for editors that add some functionality. The user can compose these graphically. For instance, a normal text editor can be extended to send notifications about text changes to other users. Note that this wrapper would work with *any* text editor that implements the required contract. Other applications of such wrappers can be found within the BlackBox component framework [10].

We take it as strong support of our proposal that it implements the formal model used by Cook and Palsberg to describe inheritance [6]. There, inheriting classes are specified as *wrappers* that only refer to the base class' signature. A concrete base class is bound at instantiation time only.

An implementation inbetween our proposal and Self was proposed as "Delegation Through a Pointer" to be added to C++ [14]. Compared to Self, the pointer to the parent object is typed. Viewing types as approximations for specifications again, we see that one of our two restrictions applies. The parent has to meet a certain specification. In contrast to our's, Stroustrup's proposal would still allow to re-assign the parent object.

An implementation that fixes the parent object can be found in Modula-3 [3]. Its allocation procedure allows to specify a list of methods to be bound to the new object. However, the base object's class (i.e. its implementation) is fixed statically by the type of the variable passed to *NEW*. Thus, Modula-3 does not allow a dynamic selection of the parent object at run time. In this sense, it is less flexible than our construction.

In the rest of this section we sketch a simple implementation for proof of concept. More efficient implementations may be possible, in particular to short-cut in deeply nested compositions. One way to approach this problem can be found in Microsoft's COM aggregation.

The wrapping (inheriting) object can refer to the parent (inherited) object. Method calls, not handled by the wrapper, are forwarded to the parent. Cecil [5] translates an inheritance-like syntax to such object composition. To achieve the same kind of self-recursiveness as with inheritance, this scheme can be enhanced to delegation by passing an extra self parameter as shown in [7]. [13] shows that delegation is as powerful as inheritance. With our proposal, the extra parameter needs not to be visible to the programmer. The compiler would generate what elsewhere the programmer would need to do explicitly.

A contract may specify how to access instance variables of a base class. The compiler would translate such access to superclass variables to dereferential access of the parent object's variables. Further, the compiler could easily enforce certain restrictions, e.g. granting access to subclasses but not to clients.

The benefit of compiling inheritance into delegation is that objects can be composed instead of classes without losing the possibility of static analysis. For the latter, the compiler asserts that the reference to the parent object cannot be changed after creation.

It would further be possible, that the compiler enforces some kind of restricted inheritance to avoid the semantic problems caused by inheriting an invisible implementation (see above). In the extreme, the run time data structure could support plain forwarding only, but the programmer can still use the more convenient inheritance notation. Such an approach is also attractive for a programming language to be compiled to Microsoft COM's aggregation.

7 Summary

To be applicable as a foundation technology for component-oriented programming, object-oriented programming with inheritance must support a middle ground between class-based and prototype-based object-orientation. Traditional class-based object-orientation is not flexible enough, unless class specifications rather than implementations are bound at component manufacturing time. On the other hand, prototype-based object-orientation is too flexible, thereby prohibiting effective static analysis.

As the above middle ground we suggest syntactical support for inheritance but using only a specification of the base class together with an implementation as object composition "under the hood". The compiler would have to hide the details of the latter to assert that the objects are not composed arbitrarily. In particular, the parent object must match the specification. Also, the compiler would have to prohibit that the parent object is re-assigned, once the composed object has been created.

This scheme allows for full static analysis, limited only by the amount of information stated with the specification of the inherited object. It gives the best possible flexibility, since the selection of the inherited code is delayed not only until component composition time, but until object generation time. The latter allows even the user to pick the code to be bound (see Fig. 1).

Our proposal allows both to compose objects using delegation, somehow restricted delegation, or plain forwarding. Delegation has the same power as class-inheritance. Depending on the future solutions to the semantical problems of inheritance and encapsulation, restrictions up to plain forwarding may become appropriate. Our suggestion can be adopted as needed.

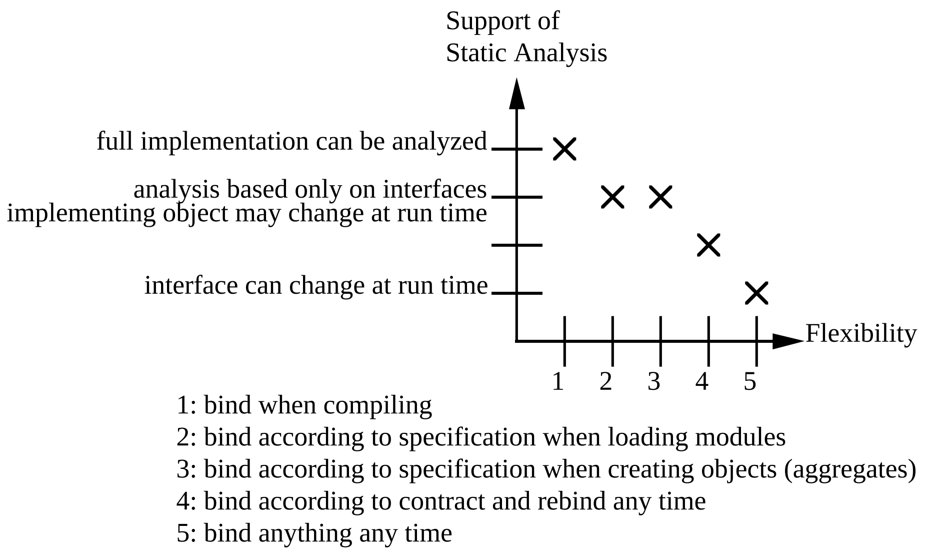

Fig. 1. Static Analysis versus Flexibility

References

1. Agesen, O., Palsberg, J., Schwartzbach, M.I.: Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 247-267.
2. Brockschmidt, K.: Inside OLE 2. Microsoft Press, ISBN 1-55615-618-9, 1994.

3. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 Report (revised). SRC Report 52, Digital Systems Research Center, Palo Alto, California, 1989.
4. Chambers, C.: Predicate Classes. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 268-296.
5. Chambers, C.: The Cecil Language, Specification and Rationale, Version 2.1. Department of Computer Science and Engineering University of Washington, Seattle as available at April 25, 1997 from http://www.cs.washington.edu/research/projects/cecil/www/Papers /cecil-spec.html.
6. Cook, W., Palsberg, J.: A Denotational Semantics of Inheritance and its Correctness. In Norman K. Meyrowitz (ed.), Proc. OOPSLA'89, SIGPLAN Notices 24:10.
7. Johnson, R.E., Zweig, J.M.: Delegation in C++. Journal of Object-Oriented Programming 4:3, November 1991.
8. Liskov, B., Wing, J.M.: A New Definition of the Subtype Relation. In O.M.Nierstrasz (ed.), Proc. ECOOP'93, LNCS 707, Springer-Verlag Berlin, ISBN 3-540-57120-5, pp. 118-141, 1993.
9. Mössenböck, H.: The Programming Language Oberon-2. Structured Programming 12:4, 1991.
10. The Oberon/F User's Guide. Oberon microsystems, Inc., Basel, CH, (http://www.oberon.ch/customers/omi), 1994.
11. Szyperski, C., Omohundro, S., Murer, S.: Engineering a Programming Language: The Type and Class System of Sather. In Proc. International Conference on Programming Languages and System Architectures, LNCS 782, March 1994.
12. Szyperski, C.A., Pfister, C.: Proc. first international Workshop on Component-Oriented Programming (WCOP'96). In M. Mühlhäuser (ed.), Special Issues in Object-Oriented Programming, dpunkt Verlag Heidelberg, ISBN 3-920993-67-5, 1997.
13. Stein, L.A.: Delegation is Inheritance. In Proc. OOPSLA'87, October 1987.
14. Stroustrup, B.: Multiple Inheritance for C++. In Proc. EUUG Spring Conference, May 1987.
15. Szyperski, C.: Independently Extensible Systems - Software Engineering Potential and Challenges. In Proc. 19th Australasian Computer Science Conference, Melbourne, Australia, 1996.
16. Ungar, D., Chamber, C., Chang, B.-W., Hölzle, U.: Organizing Programs Without Classes. In Lisp and Symbolic Computation, July 1991 4:3, Kluwer Academic Publishers, pp. 223-242, 1991.
17. Wirth, N., Gutknecht, J.: Project Oberon. The Design of an Operating System and Compiler. Addison-Wesley New York, ISBN 0-201-54428-8, 1992.
18. Wirth, N.: The Programming Language Oberon. Software - Practice and Experience 18:7, pp. 671-690, July 1988.
19. Wirfs-Brock, A., Wilkerson, B.: An Overview of Modular Smalltalk. In N.K.Meyrowitz (ed.), Proc. OOPSLA'88, SIGPLAN Notices 23:11, 1988.