Reusability Problems of Object-Oriented Software Building Blocks

Wolfgang Pree
Institut für Wirtschaftsinformatik, University of Linz, Altenbergerstr. 69, A-4040 LINZ, Austria
Tel.: ++43-732-2468-9433; Fax: ++43-732-2468-9430
E-mail: pree@swe.uni-linz.ac.at

Abstract

Object-oriented programming is a new glimmer of hope on the horizon to allow the production of reusable software components and thus to overcome the *software crisis* [McIl76]. The reusability aspect of software systems is viewed to be crucial since many software quality attributes depend on reusability (e.g. correctness, reliability, maintainability, efficiency).

There is no doubt that the object-oriented paradigm (encapsulation, inheritance, polymorphism and dynamic binding) encourages the building of extensible systems and software reuse. Nevertheless, the desired reusability of software components often cannot be achieved even by using state-of-the-art class libraries/application frameworks or object-oriented programming in general. Based on a large-scale project (object-oriented implementation of a graphic user interface prototyping tool) we identified some problems that arise in reusing object-oriented software building blocks. This paper explains the concepts of object-oriented software building blocks and discusses the most striking reusability problems of such components.

Keywords:
reusability, efficiency, maintenance, object-oriented programming, class libraries, application frameworks, C++

Introduction

We presuppose that the reader is familiar with basic object-oriented concepts (independent of a specific language): encapsulation, data abstraction, inheritance, polymorphism and dynamic binding. Application frameworks like MacApp [Schm86, Wils90], AppKit [NeXT90] and ET++ [Wein88, Gamm89, Wein89] apply these concepts in order to provide reusable object-oriented software building blocks. These frameworks were developed for the implementation of graphic user interfaces and gave rise to great hopes that the object-oriented concepts would make possible to rationalize software development in general.

We implemented DICE[1] [Pree90, Pomb91] (**D**ynamic **I**nterface **C**reation **E**nvironment, a tool that supports the graphic specification of user interface layout and that offers several ways to enhance its functionality) with the framework ET++. DICE extends ET++ in the direction of prototyping. ET++—a framework implemented in C++ that runs under UNIX and either SunWindows, NeWS, or the X11 window system—was chosen especially for the following reason: Compared to other available application frameworks, ET++ was the cleanest object-oriented implementation based on a small set of basic mechanisms. The design and implementation of ET++ is described in detail in [Wein89].

Taking an application framework as the basis of DICE can be seen as a constellation that is often considered to be optimal for object-oriented system development. Further facts that allow the

[1] This project was supported by Siemens AG Munich

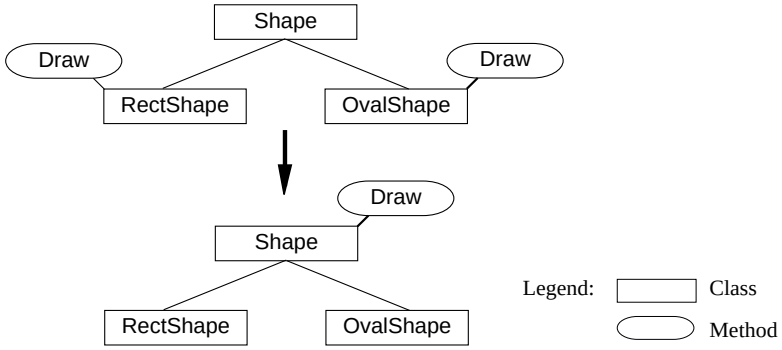

Figure 1: Code factoring

derivation of general implications (like reusability problems of object-oriented software building blocks) from the development of DICE are:

- The project size itself (DICE having been developed with an effort of about two person years) provides a sufficient empirical basis for drawing conclusions.
- DICE is a "pure" object-oriented software system consisting only of classes. A typical feature of a hybrid language like C++ is not used, namely the ability to mix conventional routines that are not members of any class with object-oriented method calls.
- DICE is a "typical" application-framework-based software system (i.e., its structure—the application model—as well as components defined in ET++ satisfy the needs of DICE to a high degree).

We first discuss concepts of object-oriented software building blocks (=class libraries) before defining the term *application framework*. This explanation is a precondition for the understanding of reusability problems that are still inherent in object-oriented software.

Concepts of Object-Oriented Software Building Blocks

Compared to conventional routine libraries, class libraries are hierarchical with the most general class at the top of the hierarchy tree (if single inheritance is used). This hierarchical organization helps to reduce the complexity of a library. An important principle behind the design of a class hierarchy is that the common behavior of classes is factored out into their superclasses. For example, classes implementing various graphic objects (e.g., the classes RectShape and OvalShape) will all have—among other commonalities—a method Draw(). As shown in Figure 1, commonalities can be factored out into a superclass Shape.

Although the method Draw() cannot be implemented in class Shape, its name and parameters are standardized (because of the inheritance mechanism of object-oriented languages) for all subclasses of Shape. For instance, a comment accompanying this method describes what behavior is expected from it.

Classes which factor out common behavior of other classes typically contain some methods that cannot be implemented. Any class that contains one or more "empty" methods (i.e., methods with some kind of dummy implementation) is termed *abstract class*. It doesn't make sense to generate instances of them. Nevertheless abstract classes may also contain methods that can already be implemented in advance for all subclasses. Abstract classes form the basis of *extensible* and *reusable* software systems:

Extensibility

Sometimes it is possible to realize even whole software systems using only the methods of abstract classes. If subclasses of abstract classes are added to the class library, these software systems need not be changed. They also work with the new components, since objects that are instances of subclasses of the abstract classes (on which other software systems are based) have at least all the methods (though implemented in a specific manner) defined in their (abstract) superclasses. The methods of abstract classes are dynamically bound, so that the corresponding

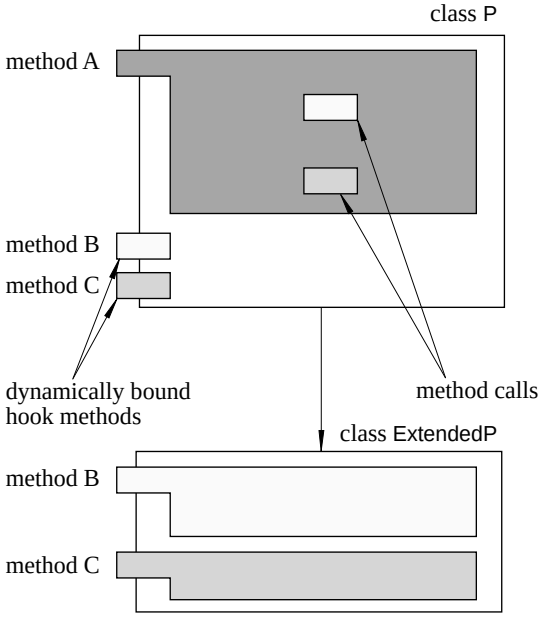

Figure 2: Concept of extensible classes

methods of the objects which are instances of the new classes are called at run time.

Figure 2 shows an example of an extensible class P which has the three methods A, B and C. The methods B and C are application-specific and cannot be implemented in the abstract class P (only their interface is specified). These dynamically bound methods are called in method A. In order to add specific behavior to class P, a subclass (for instance, ExtendedP in Figure 2) must be defined which implements the hook methods accordingly. The point is that software systems which are based on objects of type P still work if objects of type ExtendedP are supplied, since subclasses are compatible to their superclasses. Due to dynamic binding, the method calls depend on the object's run time type.

Reusability

New subclasses can reuse all the code that was already implemented in their superclasses. Class libraries are called *application frameworks* if they apply the ideas presented above in order to provide a software system which is a generic application for a specific domain. Applications based on such an application framework are built by customizing its abstract and concrete classes. Thus a given framework already anticipates much of an application's design which is reused in all applications based on the classes of that application framework. User interface application frameworks, for example, provide a reusable, blank application that implements much of a given user interface look-and-feel standard.

Frameworks are not limited to the construction of interactive, graphic-oriented user interfaces, but can be applied to any area of software systems. Examples are frameworks for VLSI routing algorithms [Goss89] and for controlling real-time psychophysiology experiments [Foot88]. Nevertheless almost all of the publicized frameworks focus on graphic user interfaces. User interface frameworks are domain-independent, are useful to most programmers, and correspond to a traditional computer science area of specialization. They are one of the main reasons why object-oriented programming enjoys such a good reputation for promoting extensibility and reuse.

Weinand [Wein89] states that frameworks are especially attractive if a standard user interface should be encouraged, for it is possible to completely define the components that implement this standard and to provide these reusable components as building blocks to other developers. This is an advantage over the (conventional) toolkit approach where most user interface look-and-feel standards are explained textually rather than "wired" into the software. A user interface application framework defines much of an application's standard user interface, behavior, and

operating environment, so that the programmer can concentrate on implementing the application-specific parts.

Reusability Problems

DICE is based on the application framework ET++. During the development of DICE we identified some problems that make the reuse of software components (though implemented with object-oriented programming techniques) rather difficult:

Inefficient Reuse

It is often unclear how specific behavior can be added to predefined software components:

- by the extension of classes (creating subclasses),
- by the implementation of new classes, or
- by using mechanisms (e.g., change propagation as described in [Kras88]) that are supported by the particular class library/application framework.

The root of this problem is the high complexity of class libraries and especially application frameworks. In order to decide how specific behavior can best be implemented on the basis of a class library/application framework, a high degree of familiarity with the respective class library/application framework is mandatory. The user of classes must know exactly which features the individual classes provide and even sometimes how the behavior of individual classes is implemented.

Thus applications based on powerful, complex class libraries/application frameworks will sometimes be inefficient: application programmers who reuse software components often do not (and should not) have intimate knowledge about implementation aspects of particular features provided by reusable components. Although one can assume that particular services of software components rely on efficient implementation techniques, it is difficult to assess whether certain combinations are still efficient.

Our experience has shown that application-specific solutions are often chosen that are too cumbersome, even after a designer has accumulated extensive experience with a particular application framework. For example, in DICE's window specification editor ET++'s change propagation mechanism was used to recompute the layout after making size changes in user interface elements. After consulting one of the developers of ET++, it was determined that recomputation of the layout could be invoked by directly calling a particular method (at the right time), which is much more efficient than using change propagation.

Difficult Maintenance of Reusable Software Components

If classes are reused in other projects, the need for more general versions becomes apparent. To address this issue, Winblad et al. [Winb90] suggest dividing programmers into two groups: *class programmers* and *application programmers*. Class programmers are preferably highly qualified software engineers who develop classes and enhance them for reusability in future projects. Application programmers produce applications as quickly as possible by means of reusable components and give the class programmers necessary feedback to make these components more reusable.

Nevertheless, the evolution of class hierarchies by the class programmers can also have subtle side effects. The "advance warning" of application programmers by class programmers about changes in reusable software components is almost impossible since the effects of (even slight) modifications in class libraries/application frameworks often cannot be predicted. A report of all changes is also unpracticable because application programmers would have to understand many implementation details in order to comprehend all changes and their effect on applications based on these modified classes.

For example, DICE was always ported to more recent versions of ET++ as soon as these versions were available in order to profit from improvements. The migration to more recent ET++ versions was usually no problem, although class interfaces often changed radically. The benefits (unification of concepts in new classes, better performance, etc.) outweighed the effort (in average 1-2 person days). But one example demonstrates that even almost negligible changes can have enormous effects on applications that are based on a class library/application framework: ET++, for example, implements a hook method

```
virtual bool IsEqual(Object *op);
```

in class Object, the root of ET++'s class hierarchy. IsEqual is used in several other classes—

especially in collection classes like SortedObjList and Set—in order to compare two objects. The implementation of this method in class Object can only check pointer identity. Subclasses (e.g., TextItem) have to override this method accordingly.

All editing operations of DICE's window specification editor are based on pointer identity of DICEItem instances. (DICEItem is the abstract superclass of all classes that implement concrete user interface elements supported by DICE. DICEItem is derived from the ET++ class VisualObject which is in turn a sublass of ET++'s root class Object.) Since the desired behavior of IsEqual is already implemented in Object, it was not necessary to override this method in DICEItem. After DICE was ported to an improved version of ET++, some of the editing functions did not work properly (user interface elements that were moved within a window specification editor were removed together with others, etc.). Only a careful examination of the source code of ET++'s collection classes made it obvious that comparison of objects by means of IsEqual was wrong: This method of class Object was not overridden in VisualObject in earlier versions of ET++. The new ET++ version implements a method IsEqual in VisualObject. So DICEItem inherits a behavior of IsEqual from VisualObject which is different from pointer identity. The elimination of such errors is easy (in this example a proper IsEqual method in DICEItem), but their detection can sometimes mean hard work that requires knowledge of implementation details of reusable classes.

Restricted Extensibility of C++ Classes

C++ requires that the programmer explicitly declare methods to be dynamically bound. Furthermore, instance variables of a class can be protected so that subclasses have no access to them. These characteristics of C++ imply that C++ classes can often only be reused and extended without problems if the designer has forseen the wishes of future users. In many cases reuse is only possible if the source code of the class to be modified is available. These facts directly contradict the promises of object-oriented programming.

Let us illustrate this statement. Class behavior can be modified by overriding dynamically linked methods (called hook methods) in subclasses. This procedure is possible only at locations where the class designer provided hooks. Application frameworks like ET++ in particular provide a structure in their "empty" application which is to be extended accordingly for an application that is based on it. Certain extensions are necessary for almost every application. These extensions are prescribed by the framework (by means of hook methods) and forseen by the designer of the framework classes. In this case it is easy to reuse and extend those building blocks. For extensions that were not foreseen, the source code of the respective superclass method(s) must be copied and complemented by the desired statements. This procedure is only possible if

- the source code of the class to be modified is available; and
- the instance variables used in the respective method(s) can be accessed in their subclasses.

Another design principle of application frameworks restricts its extensibility independent of the particular implementation language. State-of-the-art application frameworks usually adhere to the idea of a narrow inheritance interface [Wein89]: Behavior that is spread over several methods in a class should be based on a minimal set of methods that have to be overridden in subclasses. Thus a client deriving subclasses from an existing class has to override just a few methods in order to adapt its behavior. Not adhering to this narrow inheritance principle often means that too many methods have to be overridden, resulting in ugly and bulky code.

As a consequence modifications not forseen by a class designer imply a reimplementation of considerable parts of the respective class.

Source Code Required for Modifications

Experience has proven that almost no software component is free of errors. This is also true of components that are intended to be reused. In order to correct errors or to eliminate restrictions in the library classes by creating subclasses, the source code is often necessary to understand class behavior and to find the methods that have to be modified.

The comprehension of a class to be modified can become more difficult due to the fact that objects often send messages to themselves, which may

cause the execution of methods up and down the class hierarchy [Taen89].

Conclusion

If one considers application frameworks as an effort to apply the object-oriented programming paradigm as cleanly as possible, then one must conclude that this paradigm is not sufficient for the achievement of systems that are extensible/ reusable without considerable problems. Extensibility/reusability still has limits that cannot satisfy a system developer.

References

[Foot88] Foote B.: Designing to Facilitate Change with Object-Oriented Frameworks; master thesis, University of Illinois at Urbana-Champaign, 1988.

[Gamm89] Gamma E., Weinand A., Marty R.: Integration of a Programming Environment into ET++: A Case Study; Proceedings of the 1989 ECOOP, July 1989.

[Goss89] Gossain S., Anderson D.B.: Designing a Class Hierarchy for Domain Representation and Reusability; Proceedings of Tools ‘89 (Paris, France, 1989).

[Kras88] Krasner G.E., Pope S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80; Journal of Object-Oriented Programming 1, 3 (Aug./Sept. 1988).

[McIl76] McIlroy M.D.: Mass-produced Software Components; in “Software Engineering Concepts and Techniques” (1968 North Atlantic Treaty Organization (NATO) Conference on Software Engineering), eds. Buxton J.M., Naur P., and Randell B., 1976.

[NeXT90] NeXT, Inc.: 1.0 Technical Documentation: Concepts; NeXT, Inc., Redwood City, CA, 1990.

[Pomb91] Pomberger G., Bischofberger W., Kolb D., Pree W., Schlemm H.: Prototyping-Oriented Software Development, Concepts and Tools; in Structured Programming Vol.12, No.1, Springer 1991.

[Pree90] Pree W.: DICE—An Object-Oriented Tool for Rapid Prototyping; in Proceedings of Tools ‘90 (Sydney, Australia, 1990).

[Schm86] Schmucker K.: Object-Oriented Programming for the Macintosh; Hayden, Hasbrouck Heights, New Jersey, 1986.

[Taen89] Taenzer D., Ganti M., Podar S.: Problems in Object-Oriented Software Reuse, Proceedings of the 1989 ECOOP, July 1989.

[Wein88] Weinand A., Gamma E., Marty R.: ET++ - An Object-Oriented Application Framework in C++; OOPSLA’88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988.

[Wein89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework; in Structured Programming Vol.10, No.2, Springer 1989.

[Wils90] Wilson D.A., Rosenstein L.S., Shafer D.: Programming with MacApp; Addison-Wesley, 1990.

[Winb90] Winblad A.L., Edwards S.D., King D.R.: Object-Oriented Software, Addison-Wesley, 1990.