On Feature Protection in C++, Java and Eiffel

Markku Sakkinen

University of Jyvaskyla[*], Department of Computer Science and Information Systems
P.O. Box 35, FI-40014 University of Jyvaskyla, Finland
markku.j.sakkinen@cs.jyu.fi, sakkinenm@acm.org

Abstract

The protection facilities (access levels) of Eiffel, C++ and Java are briefly explained and compared. Several problems are pointed out and remedies to them suggested. A few possible enhancements are also proposed.

Categories and Subject Descriptors D.3.3 [*Inheritance*]; D.3.2 [*Object-oriented languages*]; D.3.3 [*Classes and objects*]; D.3.3 [*Modules / packages*]

Keywords protection, access levels, access control, inheritance, packages, Eiffel, C++, Java, composite objects

1. Introduction

This article discusses the feature protection facilities of Eiffel, C++ and Java, which belong to the currently or during the previous decades most important and interesting Simula-like (at least in their object-oriented aspects) languages. In C++ and Java access control is even very similar to that of Simula. It is applied on the class level, in strong contrast to Smalltalk, where the protection of features works on the object level — and is hard-wired in the language. In Eiffel, both class-level and object-level protection can be applied.

The main goal of this article is to point out problems and suggest remedies, as well as some new ideas. The terminology of Eiffel will be used mainly in the rest of this paper: feature, method, attribute, parent vs. heir, ancestor vs. descendant (the reflexive and transitive closures of the previous terms). Most of the viewpoints on C++ are based on my articles (Sakkinen 1992a, 1992b). I had not yet seen the papers by Ardourel and Huchard (2002, 2004) before submitting the previous version of the current paper.

It may well be that some details here are not correct for the newest versions of the languages. Especially in the evolution of Eiffel there have been rather radical backward incompatible changes. However, the most important points made in this article should hold, because major design decisions have usually been impossible to change years later even if the language designers admit that they had not been very good.

[*] It is actually Jyväskylä, but atavistic indexing systems might not handle that correctly.

2. Protection and inheritance

In all three languages, protection is strongly connected to inheritance. A feature (method or attribute) defined in a class *A* is always accessible to the methods of *A* itself, of course. Access may be granted also to all its descendants, or all classes (and non-class code in C++).

In C++ and Java, the keyword *private* causes a feature to be accessible only to the defining class itself, *protected* also to descendant classes, and *public* to everybody. In Eiffel, there is no equivalent of *private* — descendants always have the same rights as the class itself. The accessibility to other classes can be defined by an *export clause*.

The meaning of *protected* was as simple as that originally in C++. However, later it became more complicated and restrictive: a method of a class *B* can access an inherited protected feature only in an object that is statically known to be of class *B* or its descendant. Java adopted the same rule.

For *virtual methods* in C++, protection does not apply to redefinitions. This means that although a class cannot call inherited private methods, it can redefine them.

In C++, the same keywords apply also to parent classes in inheritance, or more exactly, to the inheritance relationships. This means, e.g., that if *B* inherits *A* as protected, it appears as an heir of *A* only to its own descendants.

The protection of an inherited *feature* is the stricter of its protection in the parent and the protection of the parent — except for private properties, which cannot be visible in the heir class in any case. The original protection can be restored by a declaration, however. In other cases the protection level of an inherited feature cannot be changed in a descendant. However, this does not apply to the redefinition of an inherited virtual method.

In Eiffel, the protection levels of inherited features can be redefined freely. In newer versions of the language, there is the possibility of *non-conforming inheritance*. It has largely the same effect as protected inheritance in C++, but I have yet to sort out its ramifications.

3. Protection and other language features

In Java, the main purpose of a *package* is to form a unit of protection. Thus, in addition to the three previously described protection levels, features can be accessible to classes in the same package as their defining class. This package level is even the default, and there is no keyword for it. Furthermore, all *protected* features are also accessible within the package.

Nested classes (inner classes) in C++ and Java are also features like attributes and methods, so the same protection levels apply to them. In Java, they apply also to *top-level classes*, because they are contained in packages. Obviously, only the default package level and *public* are applicable to them.

The access rights *from* methods of a nested class are fixed in both C++ and Java — there are no qualifiers that can be given. However, their principles are opposite to each other. In C++, a nested class cannot access any features of its enclosing class unless it is a *friend* (see below). In Java, it can access all features that are logically possible; for instance, a *static* nested class can access only *static* features of the enclosing class.

In C++, a class *A* can declare any methods, functions and other classes as *friends*; they will then have access to all features of *A*. In Eiffel, the export clauses can similarly give access to any other classes, but they can be defined for each feature separately. All descendants of the designated classes will inherit the access rights. Friend status in not inherited in C++: if *A* declares *B* to be a friend, that will not cause the heirs of *B* to be friends of *A*, nor *B* to be a friend of the heirs of *A*.

In Eiffel, object-level protection can be given to a feature by explicitly not exporting it even to the defining class itself. In any case, an attribute of an object can be modified only by the object itself, so the protection level affects only read access. In C++ and Java there is not even a possibility to restrict the defined access level further to read-only.

4. The problem of *protected*

The current meaning of *protected* is intended to prevent some possible cases of bad programming. However, I have not seen any convincing examples of its usefulness, and the prevention of bad programming is a hopeless goal in general.

Unfortunately, the current C++ and Java rule completely prevents many useful and natural programming idioms. It is extremely common that a class has some attributes that would be handled also by some methods of its descendants. Another common case is that an abstract class declares some abstract binary helper methods that are then implemented in descendant classes. The current rule forces such features to be declared public, which is obviously harmful. One possibility would be to offer both the original and the current, stricter *protected* access level.

One common situation that the current rule does allow is that a method accesses protected features of the current object (*this*). It seems that a much better way against some accidental programming mistakes would be to allow also object-level protection. This would be just as useful for private as protected features.

5. Other C++ issues

The possibility to redefine inherited private virtual methods although they cannot be called may be useful in some rare situations. However, it is somewhat dubious, because it works even for such remote ancestor classes that are not accessible because of private inheritance. A much more useful possibility would be the opposite: to freeze an inherited virtual method so that it cannot be further redefined. This is possible in Eiffel and Java (and Simula).

The way to restore the original protection of a feature when it has been restricted by protected or private inheritance, has a flaw that could be avoided very easily: with overloaded methods, it affects all of them — and is impossible if the original access levels are different.

In my opinion, the most severe problem with access control in C++ occurs in *fork-join multiple inheritance* (diamond inheritance). Namely, every inheritance relationship, independently of its protection level, can be defined to be either *virtual* or not (the default), This will then in further inheritance determine whether a common ancestor *subobject* (see Section 8) will be shared or replicated.

My firm opinion is (still) that the choice between sharing and replication should depend fully on the protection levels of the inheritance relationships (Sakkinen 1992a). In short, subobjects that are accessible to the class of a complete object should be shared, while inaccessible subobjects should be replicated. I hope to present the reasoning for this in the discussion on multiple inheritance at this workshop.

6. Other Java issues

To me it makes no sense that accessibility to descendants implies accessibility to classes in the same package. Inheritance and packaging are orthogonal dimensions, so the protection should be defined independently for each. That would add only one more case to the current four protection levels: descendants only. However, a sixth case could also be useful: descendant *and* in the same package. A reviewer remarked that it has actually been there in some version of Java.

If one really wanted to keep a minimal and totally ordered set of access levels, the order between protected and package should at least be inverted. The relationship between parent and heir class is rather intimate — they coexist within the same objects. The relationships between classes in a package are in general weaker; in Java this is particularly so, because its packages are no closed modules like those of Ada and Modula-2.

From this viewpoint, the meaning of *protected* looks even more illogical in Java: heir classes have a more restricted access to *protected* features than classes in the same package.

7. Eiffel issues

I my opinion it is a drawback that private features are not possible in Eiffel. However, it seems to be an important part of the philosophy behind Eiffel that classes cannot hide anything from their descendants.

Another important part of Eiffel philosophy is Design by Contract, which includes class invariants as well as pre- and postconditions for methods. However, the pre- and postconditions are expected to hold only with *external calls* (from another object[1]) of a method; *internal calls* are those addressed implicitly to the current object of the invoking method, i.e., object-level recursion.

Now, an executing method does not know whether it has been called externally or internally — thus it does not know whether its precondition holds initially and it needs to fulfill its postcondition, or not. This can clearly be problematic quite often.

To alleviate this problem, it could be useful to allow inverse object-level protection, i.e., that a method could be declared to be callable only externally. The root problem does not actually depend on explicit pre- and postconditions in the language; thus this addition would be useful also in other languages. Of course, it would not help against *indirect* object-level recursion, but that is one of the truly tough problems of OOP.

8. Composition and protection

C++, like such languages as Pascal, Modula-2 and Ada, offers both value and reference semantics also for all complex data types, including classes. Eiffel has only value semantics for *expanded classes* and reference semantics for other classes. Java and most other OO languages support only reference semantics for classes.

It is natural that C++ supports true composition, i.e., objects as parts of other objects. Just as naturally, Eiffel does that for parts that are instances of expanded classes. In contrast, Java and most other OO languages do not support true composition — although that would be possible even with reference semantics.

Inheritance can be regarded as a special case of composition (Sakkinen 1989), where the parent classes are part classes and the heir class the composite class. This model fits very well C++ and even Pascal, but it does not fit Eiffel. One reason for that is that the inherited part in an heir class object is not always like an object of the parent class, because the types of inherited attributes can be redefined.

It seems sensible to me that a class could declare some features to be accessible also to all composite classes in which it is used as a part. However, access should clearly be object-based, within a composite object. Because inheritance means a stronger coupling between classes, this access level should imply object-based (but not class-based) protected access.

This idea is very new, so I have not yet thought out an example to convince others about its usefulness, but I am sure such examples can be presented.

A table of the access levels of C++, Java and Eiffel will be posted soon at `http://users.jyu.fi/~sakkinen/maspeghi-2015/levels.pdf` .

[1] Calls through an explicit reference are considered external even if that reference happens to be to the current object.

References

G. Ardourel and M. Huchard. Access graphs: Another view on static access control for a better understanding and use. *Journal of Object Technology*, 1(5):95–116, 2002. . URL `http://www.jot.fm/issues/issue_2002_11/article1`.

G. Ardourel and M. Huchard. Class-based visibility from an MDA perspective: From access graphs to Eiffel code. *Journal of Object Technology*, 3(4):177–195, 2004. . URL `http://www.jot.fm/issues/issue_2004_04/article10.pdf`.

M. Sakkinen. Disciplined inheritance. In S. Cook, editor, *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989.*, pages 39–56. Cambridge University Press, 1989. ISBN 0-521-38232-7.

M. Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69–110, 1992a. URL `http://www.usenix.org/publications/compsystems/1992/win_sakkinen.pdf`.

M. Sakkinen. Corrigendum to "A critique of the inheritance principles of C++". *Computing Systems*, 5(3):361–363, 1992b. URL `http://www.usenix.org/publications/compsystems/1992/win_sakkinen2.pdf`.