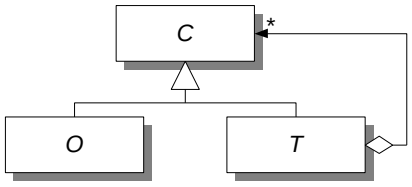
On the Role of Language Constructs for Framework Design

COT/4-10-1.0



Centre for Object Technology

Centre for
Object Technology

Revision history: 07-07-97 v1.0 Submitted version

Author(s): Görel Hedin
Jørgen Lindskov Knudsen

Status: Public

Publication: ACM Computing Surveys, Symposium on Object-Oriented Application Frameworks, (forthcoming June 1999).

Summary:

This paper focus on the role of language constructs in the design of frameworks, with emphasis on support for encapsulation of the stable part of the design, and on support for capturing its intentions in a precise and preferably statically checkable way. Framework design is a balance between flexibility and safety. However, in order for frameworks to be industrially acceptable, the structural and behavioural properties of a framework must be enforceable (mostly statically). We will show that well-known static language constructs offers strong support for industrial framework design, providing that they are generalized.

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute

Submitted to: ACM Computing Surveys == Symposium on Object-Oriented Application Frameworks
July 7, 1997

On the Role of Language Constructs for Framework Design

GÖREL HEDIN

Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Gorel.Hedin@dna.lth.se

JØRGEN LINDSKOV KNUDSEN

Dept. of Computer Science, Aarhus University
Ny Munkegade, DK-8000 Aarhus, Denmark
e-mail: jlknudsen@daimi.aau.dk

A framework encapsulates a reusable stable design and provides hooks for extending and varying this design and is *planned* for reuse. It's whole reason for existence is to be reused in different applications. A framework realizes a coherent software architecture, consisting of classes and objects with well-defined structural and behavioral properties. The framework is intended to be varied in given ways, and a well-designed framework will allow these variations to be easy to write correctly, and at the same time provide sufficient flexibility in varying the design. Good language support will allow a framework designer to use the language to set up rules for the intended use of the framework. For example, it is desirable to have precise control over how framework classes may be specialized. We will here focus on the role of language constructs for the design of frameworks with emphasis on support for encapsulation of the stable part of the design, and on support for capturing its intentions in a precise and preferably statically checkable way. Framework design is a balance between flexibility and safety. However, in order for frameworks to be industrially acceptable, the structural and behavioral properties of a framework must be enforceable (mostly statically). We will show that well-known static language constructs offer strong support for industrial framework design, providing that they are generalized. Our starting point is to look at current object-oriented languages which are both safe and flexible, exemplified by Eiffel [1], BETA [2], and Java [3]. These languages are all based on mainly static type checking and garbage collection which we find as basic prerequisites for being able to design safe frameworks. We will discuss the role of four generalized language mechanisms in supporting framework design: generalized block structure, generalized inheritance, generalized virtuality, and singular objects. These mechanisms are all available in BETA, and partly in several other languages.

GENERAL BLOCK STRUCTURE

Most programming languages exhibit some form of block structure, where block constructs like classes, records, procedures, and functions can be nested within each other. With *general block structure*, we mean the possibility to nest any kind of block construct within any other kind of block construct to an arbitrary nesting depth. This was pioneered in Algol whose block constructs are the procedure and statement block, which could be nested arbitrary and to any depth. Some function-oriented languages are also built on general block structure, most notably Scheme [4].

For object-oriented languages, the general tendency has unfortunately been *not* to provide general block structure, and to have severe restrictions on how blocks may be nested. Typical block constructs in object-oriented languages are class and method constructs. For most object-oriented languages, classes may contain methods, but classes cannot contain local classes, and methods cannot contain local classes or methods. In contrast, Simula (which was designed as an extension to Algol) did keep the general block structure and allowed arbitrary nesting of classes and methods to any depth. However, there are certain restrictions in Simula for how nested classes may be used and how nested classes may inherit from other classes. These restrictions were removed in BETA. Different uses of nested classes in BETA is discussed in detail in [5]. Java has recently adopted the BETA style of allowing classes to be nested (called *inner classes*) [6]. C++ [7] namespaces allow a very limited form of nested classes where a local class can only access static variables and methods of its outer class, not dynamic

ones. There are also experimental object-oriented languages which have adopted BETAs ideas of more general block structure [8].

General block structure is useful in frameworks because it supports the notion of what we may call *nested hooks*. A hook is a location in the framework which can be specialized by the application programmer. The normal kind of hook is an abstract class which can be specialized by subclassing, and which contains abstract methods (hook methods) which can be specialized by providing overriding methods in the subclasses [9]. This normal kind of hook is thus a 2-level nested entity. However, by relying on general block structure it is possible to support *nested hooks*: A hook (class or method) may contain any number of local hooks (other classes or methods) each of which may contain any number of local hooks, and so on to any suitable depth. This provides the framework designer with excellent possibilities for describing precisely what can be extended and specialized in a framework. Nested hooks will thereby be truly architecture and behavior preserving since they can be statically enclosed directly within the framework.

General block structure also allows the framework itself to be described as a block. In most object-oriented languages, a framework is a collection of classes which form some kind of package or library. However, general block structure allows the framework itself to be described by a class. The framework class can then contain local classes and methods, some of which may be hooks. This is useful because the framework class will itself be a hook, and one may obtain a default application simply by instantiating the framework class as is.

Allowing frameworks to be designed as classes gives another important property, namely design of an inheritance hierarchy of frameworks, with the general framework at the root of the hierarchy, and the very application specific frameworks at the leaves of the hierarchy.

This use of general block structure is omnipresent in the BETA library frameworks. For example, in Lidskjalv, the framework for graphical user interfaces, Lidskjalv is itself a class, containing local classes for windows which in turn contain local classes for menus, etc.

General inheritance

Inheritance is often described as an “incremental modification mechanism” [10], allowing individual instance variables and operations to be added in subclasses. However, the possibility to add or override operations gives fairly coarse-grained incremental modification. Fine-grained incremental modification can be achieved by supporting inheritance also for methods, i.e. a method can have submethods in analogy to a class having subclasses. BETA supports inheritance for methods in the following way: The supermethod may contain a statement *inner* which causes the code of the submethod to be executed. Submethods may declare additional input and output parameters (return values). (See [11] for more details.) The *inner* construct originates from Simula where it is used in class bodies. The idea of submethods based on Simula’s *inner* mechanism was originally proposed in [12].

If inheritance is supported for all kinds of block constructs in a language, we say that the language has *general inheritance*. The fine-grained incremental modification which can be obtained in languages with general inheritance is important in framework design because it gives the framework designer the possibility to define fine-grained hooks which give more control over how the framework is used.

By using method inheritance, a hook in the form of an *inner* statement can be added directly into a control structure in the framework, allowing the application programmer to extend the behavior directly in the context of the hook. For example, assume that the framework contains a method for iterating through some internal data structures. By method inheritance, the application programmer will be able to extend this iterating behavior, using all the local structures defined for the supermethod.

Method inheritance can partly be simulated by using the design pattern Template Method which factors out sub-behavior of a template method to virtual hook methods [13]. However, because the hook methods are virtual, they can only be specialized once in each subclass. In contrast, method inheritance allows the framework to define also *non-virtual* abstract methods, such as the iterator mentioned above, for which the application programmer can provide any number of submethods. This cannot be handled by the Template Method pattern.

The *inner* construct combines actions top-down (from superclass to subclass) and is similar to the *call-next-method* construct for *around* methods in CLOS, which however combines actions

bottom-up [14]. For framework design, top-down combination is often more appropriate because it gives the framework designer more control over the behavior, and relieves the application programmer of having to remember informal programming conventions, such as "when overriding this method, you must call *super* or *call-next-method* at the end of the method".

General virtuality

A virtual method is a well understood concept in object-oriented programming: a class defining a virtual method gives incomplete information about the implementation of that method. The complete information is in general not known until run-time. By taking a more general view on virtuality we can define it as a mechanism for supplying incomplete information about an entity at a given level of abstraction. Taking this view, we see that virtuality applies only to *methods* in mainstream object-oriented programming. By *general virtuality*, we mean a language where virtuality can be applied to *all* kinds of block constructs in the language. In BETA, the unification of methods and classes has lead to the notion of *virtual classes* [15] in analogy to virtual methods. A class defining a local virtual class declares that the local virtual class must be a subclass of some specific class. However, the exact subclass may not be known until run-time. Virtual classes correspond to a kind of type parameters (bounded polymorphism), and the mechanism can be used as an alternative to parameterized classes in Eiffel, or templates in C++. A recent proposal shows how virtual classes can be added to Java [16].

The covariant properties of general virtuality gives an elegant separation of the statically available information, defined in the abstraction, and the dynamically available information, defined in the specializations (in contrast to Eiffel which mixes these issues [17]).

Virtual classes are very powerful when combined with general block structure. They allow virtuals (or incomplete information) to be described at any level in the program. This is very useful in framework design, because it allows incomplete descriptions to appear at any level in the design. For example, the framework may itself contain a virtual class. This will then serve as a type parameter to the entire framework, provided as a single point for specialization by the application programmer. The alternative using ordinary main-stream parameterized classes would be for the application programmer to consistently parameterize all abstract classes in the framework (or give special instantiation operations for these abstract classes) which make use of this virtual class. This is cumbersome and error-prone for the application programmer, leading to possible structural or behavioral problems in the usage of the framework.

At the fine-grained level, general block structure combined with general virtuality allows individual methods to be parameterized by other methods or classes. Another interesting difference is that individual virtual attributes can be further specified individually, whereas in templates, all template arguments need to be bound simultaneously. Also, virtual attributes can be further specialized in several steps, whereas template arguments can only be bound once (since an instantiated template is not a template, but a class or method).

Singular objects

In traditional object-oriented languages, objects are always instances of previously defined classes. In framework design, and especially framework usages, this imposes an extra burden on the application programmer, since in order to create an object which is not just a plain instance of a framework class, the programmer needs to define a subclass of this framework class, and then instantiate the object from this new class.

Singular objects offers an elegant solution by allowing object instantiation and class specialization to be done in the same declaration, removing the need for introducing auxiliary subclasses. Singular objects were originally introduced in BETA and are now also available in Java (called *anonymous classes*).

Frameworks are often characterized as being mainly *blackbox* (where classes are instantiated rather than specialized) or mainly *whitebox* (where classes are intended to be specialized) [18]. It is common to strive for making frameworks more blackbox by eliminating the need for subclassing by adding instantiation parameters as suggested in [19]. However, such parameterization is less flexible than subclassing, and often cumbersome to use. Singular objects provide a more elegant and flexible solution.

Furthermore, it is often claimed that blackbox frameworks are easier to use than whitebox frameworks because using a method call protocol interface is easier than using a specialization

protocol. While this may be true in Smalltalk where there are very weak mechanisms for controlling specialization, it is not true in general. On the contrary, it is important that the language supports encapsulation so that only details relevant to the application programmer are exposed for specialization, thereby providing a blackbox interface also for specialization. Many languages have information hiding constructs like *private, hidden,* etc. for this. Another important aspect is the possibility to define non-virtual methods which the application programmer cannot override, and the possibility to stop a virtual method from being further overridden in subclasses, as is possibly using the *final* construct available in BETA and Java.

CONCLUSIONS

Advanced and mission-critical frameworks impose modeling and safety requirements on the programming languages to be used. In particular, there is a growing need for providing flexibility in a statically, type-safe manner. We have in this paper shown how generalizing ordinary language constructs can provide the framework designer with greater possibilities to encapsulate the stable parts of a design in a type safe way, giving the framework designer fine-grained possibilities to control how the framework can be varied, and providing a very high degree of flexibility in applying the framework. These generalized language constructs have already been realized and tested in real languages, most notably in BETA.

REFERENCES

[1] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[2] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.

[4] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press. 1985.

[5] O. L. Madsen. Block Structure and Object-Oriented Languages. *ACM SIGPLAN Notices*, 21(10), pp. 133-142, October 1986.

[6] *Inner Classes Specification.* http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/

[7]B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1993.

[8]C. Rapin. Block Structured Object Programming. *ACM SIGPLAN Notices*. April 1997.

[9]W. Pree. Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design. *ECOOP'94*. pp 150-162. LNCS 821, Springer-Verlag, 1994.

[10] P. Wegner and S. Zdonik. Inheritance as an Incremental Modification Mechanism. In *ECOOP'88*. pp.55-77, LNCS 322, Springer-Verlag, 1988.

[11] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Classification of actions or Inheritance also for methods *ECOOP '87.* pp.98-107, LNCS XX, Springer-Verlag, 1987.

[12] J. Vaucher. Prefixed Procedures: A structuring concept for operations, *INFOR*, 13, 3(Oct.) 1975.

[13]E. Gamma, R. Helm, R. Johnson, and J. O. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[14]G. Braha and W. Cook. Mixin-based Inheritance. In *ECOOP/OOPSLA'90, ACM SIGPLAN Notices*, 25(10), pp. 303-311, Oct. 1990.

[15] O. L. Madsen and B. Møller-Pedersen. Virtual Classes - A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA' '89.* pp. 397-406, *ACM SIGPLAN Notices*, 24(10), Oct. 1989.

[16] K. K. Thorup. Genericity in Java with Virtual Types. In *ECOOP'97*. pp 389-418. LNCS 1241, Springer-Verlag, 1997.

[17] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. In *ECOOP/OOPSLA'90, ACM SIGPLAN Notice*s, 25(10), pp. 140-150, Oct. 1990.

[18]R. E. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, 1(2), pp. 22-35. June/July 1988.

[19]D. Roberts and R. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. To appear in *Pattern Languages of Program Design 3*. (Edited by R. Martin, D. Riehle, and F. Buschmann.) Addison-Wesley, 1997.