To appear in ACM Computing Surveys == Symposium on Object-Oriented Application Frameworks

On the Role of Language Constructs for Framework Design

GÖREL HEDIN[1]

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

JØRGEN LINDSKOV KNUDSEN[2]

Dept. of Computer Science, Aarhus University, Ny Munkegade 116, DK-8000 Aarhus C, Denmark (jlknudsen@daimi.aau.dk)

ABSTRACT

The relationship between framework design and language constructs are discussed for two reasons: firstly, designing frameworks requires the ability to give the framework designer precise control over aspects of the framework extensions; secondly, the framework constraints should be specified such that they are statically checkable. Four existing language constructs are discussed: generalized block structure, generalized inheritance, generalized virtuality, and singular objects. It is discussed how these language constructs give precise means for controlling the framework extensions in statically checkable ways.

FRAMEWORKS AND LANGUAGES

A framework encapsulates a reusable, stable design and provides hooks for extending and varying this design and is *planned* for reuse. Its 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 [Fayad and Schmidt 1997]. 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). Such enforcement can be supported by mechanisms external to the language as suggested in [Hedin 1997] and [Minsky and Pal 1997], but it is better if the language is able to directly enforce these framework properties. 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 [Meyer 1992], BETA Madsen *et al.* 1993], and Java [Arnold and Gosling 1996]. 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:

[1] This work has been supported by NUTEK, the Swedish National Board for Technical Development.

[2] This work has been supported by the Danish National Centre for IT-Research

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. General block structure also implies that each instance of a block (activation record or object) will exist in the context of an instance of its enclosing block and will have access to all attributes (variables, methods, and classes) of that enclosing instance. This was pioneered in Algol whose block constructs are the procedure and statement block, which could be nested arbitrarily and to any depth. Some function-oriented languages are also built on general block structure, most notably Scheme [Abelson *et al.* 1985].

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) kept the general block structure and allows 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 are removed in BETA. C++ [Stroustrup 1997] allows a limited form of nested classes since a nested class can only access static members of its outer class (i.e. cannot access non-static members of the outer class). Java has recently adopted the BETA style of allowing classes to be nested (called *inner classes*) [Sun Microsystems 1996].

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 [Pree 1994]. This normal kind of hook is thus a 2-level nested entity. However, by utilizing 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.

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 as 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 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" [Wegner and Zdonik 1988], 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). The *inner* construct originates from Simula, and submethods combined with the *inner* mechanism was originally proposed in [Vaucher 1975].

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 [Gamma *et al.* 1994]. 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 [Keene 1988]. 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

Virtual methods 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. With this view, we can see that virtuality in mainstream object-oriented languages is limited to *virtual methods*. By *general virtuality* we mean that 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* [Madsen and Møller-Pedersen 1989] 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 [Thorup 1997].

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) [Madsen *et al.* 1990].

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, e.g. using the factory patterns [Gamma *et al.* 1994]) 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 class specialization and object instantiation 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). It is common to strive for making frameworks more blackbox by eliminating the need for subclassing by adding instantiation parameters as suggested in [Roberts and Johnson 1998]. 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 such 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 discussed 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 BETA.

REFERENCES

ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. 1985. *Structure and Interpretation of Computer Programs*. MIT Press.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison-Wesley.

FAYAD, M. E. AND SCHMIDT, D. C. 1997. Object-Oriented Application Frameworks. *CACM*, 40(10), (Oct.).

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. O. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley.

HEDIN, G. 1997. Attribute Extension - A Technique for Enforcing Programming Conventions. *Nordic Journal of Computing*, Vol. 4, 93-122.

KEENE, S. E. 1988. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley.

MADSEN, O. L. AND MØLLER-PEDERSEN, B. 1989. Virtual Classes - A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA'89. ACM SIGPLAN Notices*, 24(10), (Oct.). 397-406.

MADSEN, O. L., MAGNUSSON, B., AND MØLLER-PEDERSEN, B. 1990. Strong Typing of Object-Oriented Languages Revisited. In *ECOOP/OOPSLA'90*, *ACM SIGPLAN Notice*s, 25(10), (Oct.), 140-150

MADSEN, O. L., MØLLER-PEDERSEN, B., AND NYGAARD, K. 1993. *Object Oriented Programming in the BETA Programming Language*. ACM Press.

MEYER, B. 1992. *Eiffel: The Language*. Prentice Hall.

MINSKY, H. AND PAL, P. 1997. Law-Governed Regularities in Object Systems. *Journal of Theory and Practice of Object Systems*, 3(2).

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

ROBERTS, D. AND JOHNSON, R. 1998. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Pattern Languages of Program Design 3*. Addison-Wesley.

STROUSTRUP, B. 1997. *The C++ Programming Language (3th edition)*. Addison-Wesley.

SUN MICROSYSTEMS. 1996. *Inner Classes Specification*. URL: http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses

THORUP, K. K. 1997. Genericity in Java with Virtual Types. In *ECOOP'97*. LNCS 1241, Springer-Verlag. 389-418.

VAUCHER, J. 1975. Prefixed Procedures: A structuring concept for operations, *INFOR*, 13(3), (Oct.)

WEGNER, P. AND ZDONIK, S. 1988. Inheritance as an Incremental Modification Mechanism. In *ECOOP'88*. LNCS 322, Springer-Verlag. 55-77.