Draft, April 1998
The final version was published as a chapter in: Implementing Application Frameworks: Object-Oriented Frameworks at Work. Edited by M. E. Fayad, D. C. Schmidt, R. E. Johnson. Wiley 1999. ISBN 0471252018.

Language Support for Application 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.

1 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,

[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

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: generalized block structure, generalized inheritance, generalized virtuality, and singular objects. These mechanisms are all available in BETA, and partly in several other languages.
This paper is partly based on previous works as reported in [Hedin and Knudsen 1998].

2 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. (In C++, an instance of an inner class is not automatically linked to an instance of the outer class, and it can therefore not access ordinary non-static members of the outer class.) Java has recently adopted the BETA style of allowing classes to be nested (called *inner classes* in Java) [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. Normally, a 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.

2.1 Nested class hooks

The use of general block structure is omnipresent in the BETA frameworks [Knudsen *et al.* 1993]. As an example of nested hooks using classes within classes, consider a GUI framework with a window class defined in figure 1[1]:

Here, the outermost class `Window` contains an instance variable `wCanvas`, a method `setTi-`

```
Window: class {
   wCanvas: Canvas;
   void setTitle(t: text) { ... };
   Button: class {
      void onMouseUp() virtual;
      void drawAt(c: Canvas) { ... };
      void draw() { drawAt(wCanvas) };
   }
}
```

Figure 1. Window framework with local `Button` class

`tle` and a local class `Button`. Class `Button` contains a virtual method `onMouseUp` (a hook method) and two non-virtual methods `drawAt` and `draw`. Because of the class nesting, a `Button` object will exist in the context of a `Window` object, and can access attributes and operations in that `Window` object. For example, the `draw` method draws the button on the enclosing window's canvas. The framework in this case uses three levels of hooks: a `Window` hook containing a `Button` hook, containing an `onMouseUp` hook.

An application may use the framework to implement a calculator tool as shown in Figure 2:

```
CalcWindow: class Window {
   theCalculator: instance Calculator;
   plusButton: instance Button {
      void onMouseUp() extended {
         theCalculator.plus();
         setTitle(theCalculator.result());
      };
   };
   minusButton: instance Button {
      void onMouseUp() extended {
         theCalculator.minus();
         setTitle(theCalculator.result())
      };
   };
   ...
}
```

Figure 2. Application using the Window framework

Here, the framework is extended at all three hook levels: `CalcWindow` is a subclass to `Window`; `plusButton` and `minusButton` are defined as instances of the local class `Button`; and their `onMouseUp` methods are given appropriate implementations.[2] Because of the class nesting, the `plusButton` and `minusButton` objects will exist in the context of a `CalcWindow` object.

1.The language used in this paper is similar in syntactic structure to the Java language to ease the reading of the code examples. The language constructs are all found in the BETA language, and there is a 1-1 mapping between the syntax used here, and the syntax of the BETA language.

This allows their implementations of `onMouseUp` to access e.g. `theCalculator` (an object defined in `CalcWindow`) and `setTitle` (a method in `CalcWindow`).

The example shows how the framework imposes a structure where a `Button` is viewed as something local to a `Window`, i.e. it can exist only in the context of a `Window`. This allows `Button` objects to easily access attributes and operations of their window. This imposed structure makes application programming much easier than if the connection between buttons and windows had had to be handled explicitly.

Refactoring nested classes

Local classes are useful in particular when the local class is meaningful only inside the context of its enclosing class. However, if parts of it are meaningful also outside this context, it may be advantageous to refactor the framework to define those parts outside the enclosing class. This structure is shown in figure 3. The parts of `Button` which are not dependent on `Window` are factored out into a new top-level class which may be reused in other contexts than `Window`. This refactoring thus achieves both the reusability of non-nested classes and the tight coupling of nested classes.

```
Button: class {
   void onMouseUp() virtual;
   void drawAt(c: Canvas) { ... };
};
Window: class {
   wCanvas: Canvas;
   void setTitle(t: text) { ... };
   WindowButton: class Button {
      void draw() { drawAt(wCanvas) };
   };
};
```

Figure 3. Refactored Window framework

Simulating nested classes

In a language without nested classes, the nesting can be simulated by declaring the local class at the same level as the outer class, and giving the local class an explicit `context` reference to an object of the outer class, as shown in Figure 4. For a framework, this has several drawbacks, however. Firstly, the `context` reference needs to be explicitly administered by the application. Secondly, the `context` reference will be qualified by the outer class in the framework. This has the effect that extensions of the local class cannot safely access attributes and operations in extensions of the outer class, but have to resort to casting.

The use of general block structure thus allows the framework to capture more of the architecture of the system, and allows safer and easier application programming.

Nested method hooks

General block structure also allows methods to be nested within methods. For frameworks, this

2.The `plusButton` and `minusButton` are defined as *singular objects*, a concept discussed in Section 5.

Nested classes	Simulated implementation
OuterClass: **class** {	OuterClass: **class** {
v: Type;	v: Type;
InnerClass: **class** {	};
void m(...) {	
... v ...	InnerClass: **class** {
};	context: OuterClass;
};	**void** m(...) {
};	... context.v ...
	};
	};

Figure 4. Nested classes vs. simulated implementation

allows a hook method to contain a finer structure of local hook methods. We will discuss this in more detail in Section 3 since the full advantages of this builds on the notion of method inheritance.

2.2 The Framework as a Class

General block structure allows the framework itself to be described as a class. 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.

Framework specialization hierarchy

Modelling the framework as a class is useful because it allows a specialization hierarchy of frameworks to be defined with the general framework at the root of the hierarchy and the very application specific frameworks at the leaves of the hierarchy. An example of this is Simula's standard class `Simulation` which is a general framework for discrete-event simulation. It contains a local class `Process` for modelling processes in a simulation and maintains a queue of such processes, as sketched in Figure 5.

More specialized simulation frameworks can be built by subclassing the `Simulation` framework and introducing more specialized local classes. Figure 6 shows an example of such framework specialization, taken from the Simula documentation from 1970 [Dahl *et al.* 1970].

Multiple framework instantiation

Modelling the framework as a class allows data global to the framework to be modelled as ordinary instance variables. For example, in class `Simulation`, the process queue reference (`SQS`) is an instance variable. Languages without general block structure usually have special language constructs for global variables, for example "static" variables in C++ and Java. However, in contrast to such framework packages, a framework class can be instantiated more than once. Each instance of the framework will then obtain its own set of data local to the framework, but freely accessible (as global data) to the local classes in the framework. Such multiple framework instantiation is often useful. For example, in an instance of a simulation framework it is possible for an individual process to have its own instance of the framework in order to perform a local simulation. This tech-

```
Simulation: class {
  SQS: list of Process;

  Process: class {
    ...
  };

  Process current() { ... };

  void hold(T: double) { ... };
  void activate(... X: Process ...) { ... };
  void passivate(...) { ... };
  ...
}
```

Figure 5. The framework as a class - Simula's class `Simulation`.

```
JobShop: class Simulation {
  Crane: class Process {
    ...
  };

  Machine: class Process {
    ...
  };
}
```

Figure 6. A specialized simulation framework for job-shop analysis.

nique is used for example by Islo [Islo 1994].

3 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 capture more of the common architecture in the framework.

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.

3.1 Method Inheritance

As an example of when method inheritance is beneficial, consider the construction of a framework for concurrent programming, including a class for *monitors* [Hoare 1974] which provides mutually exclusive access to its encapsulated data by means of *entry methods*.[3] Each entry method must first lock the monitor (possibly involving waiting for the lock to become available), then access data, and finally unlock the monitor. This common behavior for entry methods can be captured in the framework by an abstract method `entry` as shown in Figure 7. A `Semaphore` object is declared in the `Monitor` class and is used by the `entry` method to lock and unlock the monitor (`mutex.P`, `mutex.V`). In applications of the framework, application-specific monitors can be defined by subclassing `Monitor` and providing suitable access methods as submethods to `entry`. The access methods will extend the behavior of `entry` at the point of `INNER`, thereby ensuring that the access to the monitor data is done while the monitor is locked.

```
Monitor: class {
  mutex: instance Semaphore;
  void entry() { mutex.P(); INNER entry; mutex.V() }
}
```

Figure 7. A framework for concurrent programming.

Figure 8 shows an example application defining a `FIFOqueue` using the `Monitor` class in the framework. The `FIFOqueue` contains a list of elements, `L`, i.e. the encapsulated data. Two access methods, `put` and `get`, are defined as submethods of `entry`. These methods extend `entry` both by providing additional parameters (`put` provides an input parameter and `get` a return value), and by extending the code of the method (by actual accesses to the encapsulated list).

```
FIFOqueue: class Monitor {
  L: list of Element;
  void put(e: element) entry { L.insertLast(e) };
  Element get() entry { return L.removeFirst() };
}
```

Figure 8. Application using the `Monitor` class in the framework to define a FIFOqueue.

In executing a method, e.g. `put`, which is a submethod of some other method, e.g. `entry`, the execution starts in the most general method (i.e. in `entry` in this case) and methods are combined top-down in the method inheritance hierarchy. At the place of an INNER, the code of the immediate submethod is executed. Figure 9 shows the full behavior of the `put` method when super- and submethods are combined.

This example shows that subclassing and submethoding allows the framework to factor out all that is specific to monitors as such: the monitor encapsulation, the locking and its implementation

3.This example is in part directly taken from the original paper by Vaucher [Vaucher 1975], and in part directly from [Madsen *et al.* 1993]

```
// transfer input parameters (e in this case)
mutex.P()
L.insertLast(e)
mutex.V()
//transfer return values (none in this case)
```

Figure 9. Full behavior of `put` after method combination.

using a semaphore. The application defines an application-specific monitor, with data to encapsulate and access methods to that data, much as if a built-in language construct for monitors were available.

3.2 Implementation of rendez-vous communication with method inheritance

The framework for concurrent programming can be extended by adding facilities for synchronous communication similar to *rendez-vous* in Ada [US Department of Defense 1980]. We do this by adding a `Port` concept as shown in Figure 10.

```
Port: class {
   mutex, sync: Semaphore;
   void entry() { mutex.P(); INNER entry; sync.V() };
   void accept() { mutex.V(); sync.P() }
}
```

Figure 10. Port framework for concurrent programming.

By combining block structure and method inheritance, this framework offers elegant support for rendez-vous communication. Figure 11 illustrates an application of the framework, using the rendez-vous facilities for synchronizing web browsers with a shared network server. `Browser1` and `browser2` both utilize the shared server `theHTTPServer`. All three objects run in separate threads. Assume that both browsers at the same time wish to download a web document. They will then at the same time execute essentially `theHTTPServer.getURL(url)`. Since the `getURL` operation in `HTTPServer` is a submethod of `entry`, the very first thing that happens in both browsers in this case will be the execution of `mutex.P()` on the `mutex` semaphore instance in `theHTTPServer`. Their execution will therefore be postponed until `theHTTPServer` accepts a call of one of the `entry` operations in the port `HTTPport` by executing `HTTPport.accept()`. As soon as this has been executed, one of the two browsers will be allowed to continue its execution, whereas the second browser is still awaiting the `mutex` semaphore, and `theHTTPServer` is awaiting the `sync` semaphore (see the code of the `accept` operation in `Port`). Just before the first browser has finished the `server.getURL` operation, it will release the `sync` semaphore, which in turn will release the `theHTTPServer`, making it possible for `theHTTPServer` to continue execution. `TheHTTPServer` will then execute another `HTTPport.accept()`, thereby allowing the second browser to download a web document.

This synchronization behavior is totally encapsulated, and controlled by the port framework, made possible by the two language constructs generalized block structure and generalized inheritance (especially inheritance for methods).

More extensive examples of defining and using concurrency constructs are given in [Madsen *et al.* 1993], including definition of monitors with conditions and more advanced ports. These exam-

```
HTTPserver: class {
   CommPort: class Port {
      HTMLdocument getURL(url: text) entry {
         ... // get the document from WWW
      };
      void putURL(url: text; doc: HTMLdocument) entry {
         ... // download ‘doc’ at location ‘url’
      };
   };
   HTTPport: instance CommPort;

   while (true) {
      HTTPport.accept()
   }
}
Browser: class {
   server: HTTPserver;
   void connectToServer(WEBserver: HTTPserver) {
      server = WEBserver; ...
   }

   ... when the user clicks a link ...
      server.getURL(linkURLaddress);
   ...
};

theHTTPServer: instance HTTPserver;
browser1, browser2: instance Browser;

...
browser1.connectToServer(theHTTPServer);
browser2.connectToServer(theHTTPServer);
...
```

Figure 11. Application of Port framework: a HTTP communication example.

ples all show that by the use of subclassing and submethoding, mechanisms can be built in a framework where other languages require built-in language constructs to give the same degree of support for the application programmer.

3.3 Further illustration of method inheritance

We can further illustrate method inheritance by extracting the common behavior of the different buttons in the `CalcWindow` class of Figure 2 and use method inheritance to reuse this generalized behavior in all `CalcWindow` buttons as illustrated in Figure 12. This shows how the use of method inheritance to factor out common behavior in methods, in analogy to how class inheritance is often used to factor out common behavior in classes. Other examples of the use of submethoding include the definition of control structures such as iterators for generic data structures. We will return to this issue in section 4.

```
CalcWindow: class Window
  {
    theCalculator: instance Calculator;
    UpdatingButton: class Button
      { void onMouseUp() extended
        { inner onMouseUp; setTitle(theCalculator.result) };
      };
    plusButton: instance UpdatingButton
      { void onMouseUp() extended { theCalculator.plus() } };
    minusButton: instance UpdatingButton
      { void onMouseUp() extended { theCalculator.minus() } };
    ...
  }
```

Figure 12. CalcWindow example.

3.4 Top-down combination of virtual methods

The two previous examples have shown how the *inner* mechanism is used to combine methods in submethoding. In BETA, the *inner* mechanism is used also to combine implementations of virtual methods. The *inner* mechanism combines virtual method implementations top-down, starting execution in the method implementation in the most general class. This is opposite to the *super* mechanism in Smalltalk [Goldberg and Robson 1983] and Java which combines virtual method implementations bottom-up. The top-down combination means that virtual methods are never overridden - they can only be extended. For frameworks, such top-down combination is appropriate since it gives the framework control over how methods are extended which is essential in order to ensure that invariants in the framework are not broken by the application programmer. In contrast, bottom-up combination and free method overriding is suitable for *unplanned* reuse where an application programmer reuses an implementation in order to recast it to some other purpose than originally intended. (To support frameworks better, many languages with bottom-up method combination have other facilities to give the framework more control. For example, in C++ it is possible to declare non-virtual methods, and in Java it is possible to declare methods as *final*, meaning that they cannot be overridden in subclasses.)

We will now discuss three different examples of top-down method combination in relation to frameworks.

Virtual method extension

When an application programmer defines a subclass to a framework class, it is common that the methods of the class should be extended as well. For example, consider a GUI framework supporting the *Decorator* design pattern which allows the functionality of an object to be extended dynamically [Gamma *et al.* 1994]. In the GUI framework a window can be decorated with, for example, scrollbars and borders. A decorator keeps track of its component to which it forwards all messages. In addition, the decorator may perform some extra behavior. For example, when a decorator receives the message `draw` it will first draw its component and then draw itself. Figure 13 shows an example of the framework code for `draw` in `Decorator`, capturing the common behavior of forwarding the message to the component. An `INNER` is placed last in `draw` to allow subclasses of

`Decorator` to perform their extra behavior.

```
Component: class {
  draw() virtual { INNER draw };
};

Decorator: class Component {
  myComponent: instance Component;
  draw() extended { myComponent.draw; INNER draw };
};
```

Figure 13. GUI framework with support for the Decorator pattern

The GUI framework may provide some standard decorators, like scrollbars and borders, but it is also possible to define specialized decorators in the application. For example, if the application programmer is not satisfied with the standard scrollbars, a new specialized decorator for narrow scrollbars can be defined as shown in Figure 14. The extension of the virtual method `draw` simply implements the drawing of the scrollbar. The application programmer does not have to worry about forwarding the message to the component, this is already taken care of in the framework. This solution for the decorator pattern differs from the standard implementation using bottom-up method combination where the application would typically need to remember to call `super`.

```
NarrowScrollbarDecorator: class Decorator{
  draw() extended {
    ... // draw the scrollbar
  };
};
```

Figure 14. Application of GUI framework defining specialized decorator.

The need for virtual method extension is particularly apparent for operations which in some way deal with the *complete* set of data in an object, e.g. initialization methods, clone methods, print methods, etc. Here top-down combination makes sure that the framework can perform all its actions without the risk of these actions being overridden by application code. Another use of virtual method extension is in instrumentation of framework code, e.g. in order to animate computations taking place in the framework.

Pre-conditions

Checking of preconditions for methods is common practice in order to make sure that framework operations are called by the application when in an appropriate state and with appropriate arguments, thereby supporting safe use of the framework. Top-down method combination allows preconditions of virtual methods to be checked at the declaration of the virtual method rather than to have to be repeated in each implementation of the method. Of course, a special language mechanism for preconditions, like in Eiffel, serves the same purpose.

Default behavior

It is common that virtual methods in a framework define default behavior which is *intended* to be overridden if desired in the application. In this case, the usual style of overriding virtual methods works fine. If top-down method combination is used, like in BETA, the framework method implementing the default behavior needs to check if the method is extended or not. In BETA this can be done by means of so called pattern variables (variables holding the class value of an object or analogously for methods). If the method is not extended, the default behavior will be executed. While this is not as straight-forward for the framework application as in traditional method overriding, the application program will be the same in both cases. In addition, the top-down method combination allows default behavior to be combined with preconditions as discussed above.

3.5 Comparison of top-down method combination with other techniques

We have argued that top-down method combination using *inner* is more appropriate for frameworks than the usual bottom-up method combination using *super.* The use of *super* leads to informal programming conventions, such as "when overriding this method, you must call *super* at the start of the method". In contrast, top-down combination using *inner* gives the framework precise control over how methods may be extended and/or overridden, thus supporting planned reuse and supporting that framework invariants are not broken by the application.

The *inner* construct is somewhat similar to the *call-next-method* construct for *around* methods in CLOS [Keene 1988]. However, CLOS combines actions bottom-up, so it is always possible for an application programmer to override both *around* methods and the ordinary *primary* methods defined in the framework, thus possibly destroying the semantics of the framework.

Top-down method combination for virtual methods can 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]. This can be used to replace `INNER` with a call to a virtual procedure. However, this leads to a proliferation of virtual methods. For example, for a virtual method `init` which is extended at each level in a hierarchy of classes `A`, `B`, and `C`, there would be a need to introduce three new virtual methods, e.g. called `initInnerA`, `initInnerB`, and `initInnerC`. While this is possible, it is cumbersome, errorprone, and leads to a more complex framework specialization interface.

Note, however, that *inner* can be simulated by the Template Method pattern only in the case of *virtual* methods. Sub-methoding, as discussed in section 3.1, cannot be simulated by the Template Method. E.g., if `INNER` in the `entry` method were replaced by a virtual method `entryInner-Monitor`, this would not help because `put` and `get` are not virtual implementations, but *submethods* of `entry`. The best we could do with the Template Method would be to define two template methods `put` and `get` in class `Monitor`, and let them call virtual methods `putInner` and `getInner`. These virtual methods would then be implemented by the application in subclasses to `Monitor`. However, this would restrict the monitor functionality to monitors with exactly two entry methods, and would furthermore make it necessary to decide on the number and types of parameters for these methods already in the framework. In contrast, in a framework based on submethoding, applications can define monitors with any number of entry methods and with parameters decided by the application.

4 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.

4.1 Virtual Classes

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].

Let us illustrate by a simple example of a framework with a bounded polymorphic list data type as shown in Figure 15[4]. The `List` class contains a class `ElementType` which is the class of the elements in the list. `ElementType` is *virtual*, meaning that at this level we don't know exactly which class `ElementType` is - we only know that it is at least `Object` (either `Object` or a subclass to `Object`). The `List` is implemented using a local class `Node` with `next` and `previous` references.

```
List: class {
  ElementType: virtual class Object;
  Node: class {
    element: ElementType;
    next, previous: Node
  };
  first: Node;
  <<listLib: attributes>>
}
```

Figure 15. Framework with polymorphic list data type.

In an application of the framework, we can describe a list of houses as shown in figure 16. The virtual class `ElementType` is now *extended* to `House`, meaning that it is ensured to be at least `House` (either `House` or a subclass of `House`). This implies that all elements in a `HouseList` will be at least `House` objects and when accessing attributes of an element we can safely access for example the `taxRate` attribute as shown in the figure. The method `getElement()` here symbolizes any operation in `List` which returns an object of `ElementType`. Since `aHouseList` is of type `HouseList` where `ElementType` is bound to `House`, the expression `aHouse-List.getElement()` has the type `House` and the access to `taxRate` can be statically type checked.

4.2 Virtual classes and method inheritance

The combination of virtual classes with method inheritance is very powerful because it allows abstract methods specified in a framework to be parameterized by types using the virtual class

4. `<<listLib: attributes>>` is inserted in Figure 15 for future reference. We will below add some operations to this `List` class. To reduce the space, we will not repeat the entire `List` declaration, but just give the declaration of the new operations. Just think of these new operations as being inserted textually at this place in the `List` class declaration.

```
House: class {
  taxRate: float;
  void display() { ... };
  ...
};

HouseList: class List {
  ElementType: extended class House;
  <<HouseListlib: attributes>>
}

// access to element in a HouseList

aHouseList: instance HouseList;
...
print (aHouseList.getElement().taxRate);
```

Figure 16. Application defining specialized list

mechanism. We will illustrate this by extending the `List` class in the framework with a number of operations. Figure 17[5] shows `scan`, an abstract method iterating over all the elements in the list. `Scan` calls `INNER` for each element in the list (see Section 3.1 on method inheritance) and `current` is a reference denoting the current element in the iteration.

```
--- listLib: attributes ---
void scan() {
  // Iterates over all elements in List
  pos, next: Node; current: ElementType

  pos = first;
  while (pos<>null) {
    current = pos.element; next = pos.next;
    INNER scan;
    pos = next
  }
}
```

Figure 17. Extension of the framework `List` class

The `scan` method can be specialized by the application to for example display all houses in a `HouseList` as follows:

5.`--- listLib: attributes ---` in Figure 17 specifies, that this new `scan` operation is to be inserted at the `<<listLib: attributes>>` place in the `List` class declaration in Figure 15.

```
--- HouseListLib: attributes ---
void print() scan { current.display() }
```

Note, that the virtual extension of `ElementType` in `HouseList` ensures that it is statically known that `current` is of type `House`, and therefore, that `current.display()` is legal. We can also modify the attributes of the objects through the `current` reference in `scan` as illustrated by the following `raiseTax` method:

```
--- HouseListLib: attributes ---
void raiseTax() scan {
  current.taxRate = (current.taxRate*1.01)
}
```

which will raise the tax rate of all houses in the `HouseList` by 1 %.

Submethoding can be used to define more advanced operations on `List` in the framework. Figure 18 shows the definition of an operation `select` which is a submethod of `scan`, and operations `find` and `remove` which are submethods of `select`.

The `select` method also shows the use of general block structure for methods. It defines a local method `predicate` which is virtual and used to decide which elements to include in the iteration. The `select` method can be used in the application to display all houses with a tax rate at more than 10% by defining the `printExpensive` method:

```
--- HouseListLib: attributes ---
void printExpensive() select {
  void predicate() extended { return current.taxRate>0.1 }

  current.display()
}
```

The `find` method (in Figure 18) is a submethod of `select` which returns the first element satisfying `predicate`. If no such element is found, the method will return `null`. (The `return` statement sets the return value of a method but does not alter the execution control. The `leave` statement is a structured goto statement which returns control to the caller. We can now find the first house in the list with a tax rate at more than 25% by the `findHighTaxed` method:

```
--- HouseListLib: attributes ---
void findHighTaxed() find {
    void predicate() extended { return current.taxRate>0.25 }
}
```

The `remove` method (in Figure 18) is also defined as a submethod of `select` and removes all elements satisfying `predicate`. We can use this operation to remove all houses with a 0% tax rate by the `removeZeroTaxed` method:

```
--- listLib: attributes ---
void select() scan {
// iterates over all elements in List which
// satisfies the predicate
  boolean predicate() virtual { INNER predicate }

  if (predicate) { INNER select }
}

ElementType find() select {
// returns the first element in List which satisfies
// the predicate
  INNER find; return current; leave find;
}

void remove() select {
// removes all elements in List which satisfy the
// predicate
  if (pos.next<>null) {
    pos.next.previous = pos.previous
  } else {
    pos.next.previous = null
  }
  if (pos.previous<>null) {
    pos.previous.next = pos.next
  } else {
    pos.previous.next = null
  }
  if (pos=first) { first = pos.next }
  pos.next = null; pos.previous = null
}
```

Figure 18. Operations implemented as submethods of `scan` in the framework `List` class

```
--- HouseListLib: attributes ---
void removeZeroTaxed() remove {
  void predicate() extended { return current.taxRate=0.0 }
}
```

The above discussion illustrates the elegancy and powerful static constraints that can be encapsulated in a framework when the framework design is supported by strong, static language mechanisms like general block structure, general inheritance, and general virtuality.

4.3 Virtual Classes in Frameworks

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.

We can illustrate this by the framework for business applications shown in Figure 19. This framework defines a set of cooperating classes, each implementing aspects of the business, such as the financial aspects (`Accounting`), the advertising, etc. (`Marketing`), and the order and shipment handling (`Operations`). Important for the proper cooperation of these classes within the framework is that they share the same understanding of the concept of a customer. This is in this framework expressed by the framework defining one common definition of `Customer` as a virtual class. `Customer` is virtual in the `BusinessFramework`, since it should be possible to create specialized business frameworks in which there is a specialized understanding of the concept of a customer.

```
BusinessFramework: class {
  Customer: virtual class Object;
  CustomerDatabase: class Database {
    ObjectType: extended class Customer
  };
  theDatabase: instance CustomerDatabase;
  Accounting: virtual class {
    void invoice(c: Customer) virtual {
      ... INNER invoice ...
    }
    ... functionalities for accounting ...
  };
  Marketing: virtual class {
    ... functionalities for marketing ...
  };
  Operations: virtual class {
    ... functionalities for operations ...
  };
}
```

Figure 19. Business framework

A specialized business framework `ITbusiness` is shown in Figure 20. Here, the `Customer` class is extended to include e.g. information about the customer's favorite operating system. The local classes for `Accounting` etc. are extended to make use of that information, e.g. in the `invoice` method.

Figure 21 shows how we can go even further by specializing this `ITbusiness` framework into a framework for security software. This specialization is done in a similar way by extending the definitions of `Customer`, `Accounting`, etc.

The business framework example above illustrates the power of combining virtual classes with general block structure: a virtual class (`Customer`) provides a single point of type parameteriza-

```
ITbusiness: class BusinessFramework {
  Customer: extended class {
    void favoriteOS() {
      ...
    }
  };
  Accounting: extended class {
    void invoice() extended {
      ... c.favoriteOS; ...
    }
  };
  ... similar extensions for marketing and operations ...
}
```

Figure 20. Specialized framework for IT business

```
ITsecurity: class ITbusiness {
  Customer: extended class {
    securityLevel: integer;
    ...
  };
  Accounting: extended class {
    void invoice() extended {
      ... c.securityLevel; ...
    }
  };
  ... similar extensions for marketing and operations ...
}
```

Figure 21. Further specialized framework

tion for a complete framework of different classes (like `Accounting`, `Marketing`, `Operations`, and so on). In a system with traditional type parameters such as C++ templates or Eiffel parameterized classes, one would model `Accounting`, `Marketing`, and so on as templates each with a `Customer` type parameter. These templates would then have to be individually instantiated to classes, providing a `Customer` subclass as a parameter.

The example further illustrates how virtual classes can be extended in several steps: The virtual classes `Customer` and `Accounting` defined in the general `BusinessFramework` are extended in the `ITbusiness` framework (a subclass of `BusinessFramework`) and again in `ITsecurity` (a subclass of `ITbusiness`). Such stepwise extension is not possible in traditional type parameterization.

4.4 Virtual classes and covariance

Virtual classes lead to what is known as a *covariant* type system: Consider a class `C` with a local variable `v` of the virtual class `T`, and a subclass `D` which extends `T`. This leads to a situation where the type of `v` will be more special in a `D` object than in a `C` object. I.e., the types of `v` and its enclosing class vary in the same direction (hence the term *co*-variance). The usefulness of covariance in frameworks was illustrated e.g. in Section 4.1 where `List` was specialized to `HouseList` and local references to elements were specialized from `Object` to `House` using the virtual class `ElementType`. This allows a framework to capture general aspects of a system without fixing the types of the entities described in the framework.

Many papers have discussed type problems for covariant type systems, e.g. [Cook 1989], but they have usually taken a type system based on type redefinition as a starting point where argument types may be redefined and there is no distinction between a virtual class and an ordinary class. In contrast, the use of virtual classes allows the use of covariance without type problems. Figure 22 shows the difference between covariance in a type system based on redefinition and a type system based on virtual classes.

Type system based on redefinition	**Type system based on virtual classes**
`List: **class** {`	`List: **class** {`
`  **void** insert(e: Object)`	`  ElementType: **virtual class** Object;`
`    { ... }`	`  **void** insert(e: ElementType)`
`};`	`    { ... }`
	`}`
`HouseList: **class** List {`	
`  **void** insert (e: House)`	`HouseList: **class** List {`
`}`	`  ElementType: **extended class** House;`
	`}`

Figure 22. Covariance in different type systems

In the ordinary type system, the argument `e` to `insert` is thought of as having the type `Object`, and a call to `aList.insert(new Object)` would seem correct. But the type system will break if `aList` happens to be a `HouseList` object which redefines the argument type to `House`. In the virtual class type system, the argument `e` does not have the type `Object`. Instead, `e` has the type `ElementType`, and the meaning of `ElementType` is (in general) not known until runtime since it is a virtual class. Therefore, a call to `aList.insert(new Object)` will (in general) result in a runtime check, checking the type of `e` against the value of `ElementType` for `aList`. However, in many cases, runtime checks are unnecessary because the value of `ElementType` can be determined at compile time. In the following example, `aHouseList` is a constant reference and its value of `ElementType` is `House`, which can be determined at compile time. Thus, no runtime check is needed at `insert(new House)`.

```
aHouseList: instance HouseList;
aHouseList.insert(new House); // statically typesafe
```

Another possibility of allowing the value of `ElementType` to be statically determined is to use *final extensions* of virtual classes. A final extension of a virtual class is an extension which prohibits further extension in subclasses. If `ElementType` in `HouseList` was defined as a final extension, this would disallow subclasses to `HouseList`, e.g. `SummerHouseList`, to further extend `ElementType`. A `SummerHouseList` will have to accept any `House` in its list. This would allow calls to `insert` to be statically type checked also for dynamic references qualified by `HouseList`. Figure 23 shows an example of this. At the call to `insert`, the runtime type of `aHouseList` is not known: it could be `HouseList` or any subclass to `HouseList`. But since `ElementType` is defined as a final extension in `HouseList`, we know its value for `aHouseList` at compile-time, namely `House`, and there is no need for a runtime check at the call to `insert`.

```
HouseList: class List {
  ElementType: final class House; // Final extension
}

aHouseList: HouseList;
... // aHouseList is assigned to an object
aHouseList.insert(new House); // Statically typesafe
```

Figure 23. Statically checkable covariance using final extensions

The covariant properties of general virtuality thus 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 type systems based on redefinition which mixes these issues. Covariance and virtual classes are treated in more detail in [Madsen *et al.* 1990].

5 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.

An elegant solution to this problem is to allow class specialization and object instantiation to be done in the same declaration as shown in Figure 24. An object `s` defined in this way is called a *singular object*. The class of `s` is an anonymous subclass of the class `C` used in the declaration. To the right in the figure, a traditional implementation is shown where the class of `s` has to be named (`sClass`) and declared explicitly. The use of singular objects thus leads to substantially simpler application code.

Singular object	Traditional implementation

`s: instance C {`
`  void v( ...) { ... }`
`}`
	`sClass: class C {`
	`  void v( ...) { ... }`
	`}`
	
	`s: instance sClass;`

Figure 24. Singular objects vs. traditional implementation

Singular objects were originally introduced in BETA and are now also available in Java (called *anonymous classes*). We have already seen the use of singular object in section 2.1 where `plusButton` and `minusButton` in the calculator were defined as singular objects. Without singular objects, the application programmer would have had to first define two classes `PlusButton-`

Class and MinusButtonClass containing the respective implementations of onMouseUp, and then defining the two objects plusButton and minusButton as instances of these classes.

Note that singular objects should not be confused with *singletons*, i.e. classes of which there is only one global instance [Gamma *et al.* 1994]. Singular objects are not necessarily globally defined, and if their definition is nested in some other block, there may be several instances of the singular object - one for each instance of the enclosing block. For example, plusButton and minusButton are singular objects, but if we create several CalculatorWindow objects, there will be one plusButton and one minusButton for each of the windows.

5.1 Singular objects as adapters

There is often a need to *adapt* framework classes to work together with classes defined in an application. As an example, consider an MVC-like framework (Model-View-Controller [Krasner and Pope 1988]) with general View classes for displaying information in window panes. To build an interactive tool in an application, different View classes are adapted to work together with application specific model classes. One example of a View class is ListView implementing a pane containing a scrollable list with a current selection. The specialization interface to ListView contains three hook methods as shown in Figure 25. To use a ListView in an application, we need to adapt it to fill in these hooks. Singular objects provide a very elegant solution to this adaptation.

```
ListView: class {
   List getList() virtual;
      // Should answer list of alternatives to display
   integer getCurrentSelection() virtual;
      // Should answer current selection to display
   void changeModelSelection(index: integer) virtual;
      // Should change the current selection.
      // Called when the user performs a new selection.
}
```

Figure 25. MVC framework: Specialization interface for ListView.

As an example, consider an application implementing a music tool. One of the panes in the tool should be a scrollable list displaying a number of different music styles: “reggae”, “jazz”, “classical”, and so on. An application class MusicModel contains the data to be displayed in the list pane. The tool can be constructed by attaching instances of the framework class ListView to a MusicModel instance. Figure 26 shows how this is done using singular objects.

The object LV in Figure 26 is an *adapter*: it adapts the behavior of a framework class (ListView) to work with a class in the application (MusicModel). This way to implement adapters is an alternative to the more traditional implementation techniques discussed in [Gamma *et al.* 1994]. The implementation relies on block nesting: the object LV is nested inside the object musicTool. Since LV is a (specialized) instance of ListView, it has direct access to the information in *both* of two classes which need to be connected: to ListView by subclassing and to MusicModel by block nesting. This makes it straight-forward to define the adaptation: virtual methods in ListView can directly be implemented to call appropriate methods in MusicModel.

This use of singular objects as adapters is omnipresent in applications using the BETA frameworks. The key motivation for the introduction of class nesting and singular objects in Java (called “inner classes” and “anonymous classes” in Java) was to support this implementation of adapters.

```
MusicModel: class Subject {
   selection: integer;
   musicList: List = (“reggae”, “jazz”, “classic”);

   List getMusicList() {return musicList};
   integer getMusicSelection() {return selection};
   void setMusicSelection(index: integer) {
      selection := index;
      ...
   };
   ...
};

// Code creating music tool:
musicTool: instance MusicModel {
   LV: instance ListView {
      List getList() {return getMusicList(); };
      integer getCurrentSelection() {
         return getMusicSelection();
      };
      void changeModelSelection(index: integer) {
         setMusicSelection(index);
      };
   };
   ...
};
```

Figure 26. Application of MVC framework: a music tool.

5.2 Singular objects vs. pluggable objects

Singular objects is an alternative to so-called *pluggable objects* as introduced in the Smalltalk Model-View-Controller framework [Krasner and Pope 1988]. A pluggable class is a framework class which can be adapted by the application by providing parameters rather than by subclassing. The goal is to avoid having to write trivial subclasses in the application. Typically, many of the parameters to a pluggable class are method names, and the framework class will call these methods by using the Smalltalk *perform* mechanism. In the MVC framework there are for example pluggable `View` and `Controller` classes which can be instantiated and attached to an application model without the need for subclassing these framework classes. Figure 27 shows the Smalltalk instantiation interface of a pluggable version of the `ListView` discussed in Section 5.1. The figure also shows the use of the *perform* mechanism in the implementation of `PluggableListView`.

A problem with Smalltalk’s pluggable objects is that it is a fairly complex technique for the application programmer to understand, relying on many informal programming conventions. For example, the framework assumes a certain signature for each of the methods whose names are passed in the creation message, but these signatures are not a formal part of the framework interface. Figure 28 shows a Smalltalk application program for constructing the music tool discussed in Section 5.1, but now using the pluggable `ListView` class.

By comparing the solution using singular objects in Figure 26 with the solution using pluggable

```
PluggableListView

// class methods for creation

   on: anObject list: getListSel selected: getSelectedSel
         changeSelected: setSelectionSel

// instance methods ...

   ... anObject perform: setSelectionSel with: index
```

Figure 27. Smalltalk instantiation interface to pluggable ListView.

```
MusicModel // (the same as in figure 26)

// Code creating music tool:
| aMusicModel, aLV |
aMusicModel <- MusicModel new.
aLV <- PluggableListView
   on: aMusicModel
   list: getMusicList#
   selected: getMusicSelection#
   changeSelected: setMusicSelection#.
...
```

Figure 28. Smalltalk implementation of music tool using pluggable objects

objects in Figure 28, we see that the size of the application code is practically the same. The main technical difference is in static checkability. In the Smalltalk solution, the framework uses *perform* to invoke methods in the application model. This call can go wrong if the application programmer provides the wrong arguments in the instantiation of the `PluggableListView`. For example, it is not statically checkable that `anObject` actually has a method with the name given by `setSelectionSel`. Even if there is such a method, it is not statically checkable if it has the right number of arguments. In contrast, in the solution using singular objects, all this information is statically checkable. Singular objects provides an alternative solution to pluggable objects which also avoids explicit trivial subclasses, but which provides a safer interface, not relying on passing method names as arguments, and where the signatures of all methods are part of the framework interface.

Whitebox vs Blackbox Frameworks

In the Smalltalk community, frameworks are often characterized as being mainly *blackbox* (meaning that classes are instantiated rather than specialized) or mainly *whitebox* (meaning that classes are intended to be specialized) [Johnson and Foote 1988]. The introduction of pluggable objects is seen as a way of making a framework more blackbox [Roberts and Johnson 1998]. However, these definitions of blackbox/whitebox frameworks may be suitable for Smalltalk where there are very weak possibilities for controlling specialization. For languages where static checkability is of prime importance, we find it desirable that frameworks can provide blackbox interfaces also for specialization. I.e., that parts of a framework class can be encapsulated and not be used or affected by the subclasses in the application. Many languages have information hiding constructs like *private, hid-*

den, etc. that support this. Another very important aspect is the possibility for the framework to control where overriding may take place. This can be done by mechanisms such as non-virtual methods as in Simula, BETA, and C++, by the *inner* mechanism in BETA, or by using the *final* construct available in Java.

As shown in Section 5.1, the specialization interface is clear and singular objects makes it easy to use. In contrast, the use of pluggable objects to turn a specialization interface into an instantiation interface may lead to an interface which is both unsafe and difficult to understand.

5.3 Singular method specialization

By building on method inheritance as discussed in Section 3, the idea of singular objects can be applied also for methods. Whereas singular instances of classes are normally declared as attributes of another object or class, a singular instance of a method can be used to specialize a method call.

In Section 4.2, we defined a class `HouseList`, and a series of special purpose methods: `print`, `raiseTax`, `printExpensive`, `findHighTaxed`, and `removeZeroTaxed`. If we assume, that these methods were to be used only once, we could avoid having to define these auxiliary methods, and invoke the methods as singular specializations of the original list operations. And further, if we assume that we only needed one house list, we could avoid the auxiliary `HouseList` class definition. We could then define `aHouseList` as:

```
aHouseList: instance List {
   ElementType: final class House; // Final extension
}
```

and with this definition, we could instead of defining the auxiliary methods, just invoke the singular method specialization shown in Figure 29:.

This shows how the combined use of method specialization and singular objects can be used to in effect define new control abstractions in the framework which can be used in the application. And it should be noted, that the singular object `aHouseList` and the singular method specializations are equally statically type-checkable as the `HouseList` class and the auxiliary methods defined in Section 4.2.

6 CONCLUSION

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 checkable, type-safe manner. The traditional object-oriented language constructs of classes, inheritance, and virtual methods, provides the basic mechanisms for constructing frameworks. In this paper, we have shown how generalizing these 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. The generalized language constructs discussed in this paper are not new, they have beed realized and tested in real languages for many years, most notably in the BETA language. What we have done in this paper is to illustrate precisely how these generalized constructs give support in framework design and application. We hope to have illustrated how these generalized constructs give support for *planned reuse* where it is the framework which controls how it can be extended and where framework invariants can be encapsulated so they cannot be compromised by the application. This is somewhat in contrast to mainstream object-orientation where the focus is on how an application can

```
// print
aHouseList.scan { current.display() }

// raiseTax
aHouseList.scan { current.taxRate = (current.taxRate*0.01) }

// printExpensive
aHouseList.select {
  void predicate() extended { return current.taxRate>0.1 }

  current.display()
}

// findHighTaxed
theHouse =
  aHouseList.find {
    void predicate() extended { return current.taxRate>0.25
}
  }
// removeZeroTaxed
aHouseList.remove {
  void predicate() extended { return current.taxRate=0.0 }
}
```

Figure 29. Singular method specialization.

freely override and replace parts of the framework.

The introduction of new languages, or the adoption of new language constructs into existing languages, is a difficult process which takes very long time. Recall that classes and inheritance were introduced by Simula in 1967 and it took 15-20 years before these constructs came into widespread use through Smalltalk and C++. All of the generalized language constructs presented in this paper have been in use for many years in BETA, and we are happy to see that several of these language constructs are beginning to make their way into popular programming languages, such af Java.

Acknowledgements

Thanks to Patrik Persson and Anders Ive for comments on the readability of this paper.

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.

COOK, W. R. 1989. A Proposal for Making Eiffel Type Safe, In *ECOOP'89, European Conference on Object-Oriented Programming*, Cambridge University Press.

DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. 1970. *SIMULA 67 Common Base Language*. NCC Publ. S-22, Norwegian Computing Centre, Oslo.

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.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk 80: The Language*, Addison Wesley.

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

HEDIN, G., AND KNUDSEN, J.L. 1998. On the Role of Language Constructs for Framework Design, In *Symposium on Object-Oriented Application Frameworks*. ACM Computing Surveys, Forthcoming.

HOARE, C. A. R. 1974. Monitors: An Operating System Structuring Concept. *CACM* 17, 10(Oct), 1974, 549-557.

ISLO, H. 1994. Simulation models built on nested quasi-parallel systems, In *Proceedings of the 20th Conference of the ASU (Association of Simula Users)*, Prague, pp 80-95.

JOHNSON, R. E. AND FOOTE, B. 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming*, pp 22-35, 4, 2(June/July), SIGS Publications.

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

KNUDSEN, J. L., LÖFGREN, M., MADSEN, O. L., AND MAGNUSSON, B. (EDS.) 1993. *Object-Oriented Environments: The Mjølner Approach.* Prentice Hall.

KRASNER, G. E. AND POPE, S. T. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, 1(3), pp. 26-49, August/September 1988, SIGS Publications.

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.

US DEPARTMENT OF DEFENSE. 1980. *Ada Reference Manual*.

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.