# Parameterization versus Inheritance

**Thomas Kühne**
Department of Computer Science
University of Darmstadt
Magdalenenstr. 11c, D-64289 Darmstadt
Germany

Email: kuehne@isa.informatik.th-darmstadt.de

## Abstract

Using an object-oriented language is not a guaranty for reusable software. Perhaps even more important than the language chosen, are the methods, patterns, and idioms used for design and implementation. In this respect we propose to realize behavior refinement by parameterization instead of inheritance. We show applications for behavior parameterization, avoiding disadvantages related to subclassing. We demonstrate that the so constructed micro-architectures, originating from the world of functional programming, integrate into standard object-oriented programming. Moreover, we argue that they improve object-oriented virtues such as modularity, locality of definition, and reusability.

## 1 Introduction

Good design is not a necessary consequence of object-oriented analysis or the result of using a particular language. It is crafted by experienced programmers, applying advanced techniques to specific domains.

Recently, effort has been put into the discovery and documentation of general design patterns, repeatedly used by good designers [Beck 94, Coplien 92, Gamma 93, Johnson 92, Pree 94].

We propose to take advantage of the design principles used in functional programming. Higher order functionality and partial parameterization have already proven their usefulness in this area. There is evidence for their need and justification in object-oriented programming as well [Baker 93, Dami 94, Klagges 93, Sargeant 93, Watt 94].

In contrast to creating a hybrid language, we use standard object-oriented languages for applying functional style. This is feasible since objects essentially are closures. A closure is a function with an associated environment. Therefore an object with one method (the function) and several attributes (the arguments to the function) can be perfectly used as a function closure. As a result, function passing translates to object passing and the whole world of lambda-calculus is within the reach of an object-oriented programmer [Kühne 94].

Although the title of our paper is an homage to Meyer's "Genericity versus Inheritance" [Meyer 86], we applied a different paper organization. In the following we will show when it is beneficial to use object passing ("takes-a") instead of inheritance ("is-a"). First, we examine inheritance and parameterization as alternatives for supplying behavior to incompletely specified classes. Second, we investigate the appearance of the "takes-a" relation in related work. We conclude by summarizing the results of this paper.

Appeared in Conference Proceedings of TOOLS 15, Melbourne, Australia, editors Christine Mingins and Bertrand Meyer, pages 235–245, 1995; Proceedings contain premature layout version.

# 2 Inheritance

## 2.1 Principle

The standard way to complete the behavior of a class is called "subclass responsibility" [Cox 86]. Methods in an abstract superclass (called base or template methods) rely on the implementation of abstract methods to be defined in concrete subclasses (called abstract or hook methods).

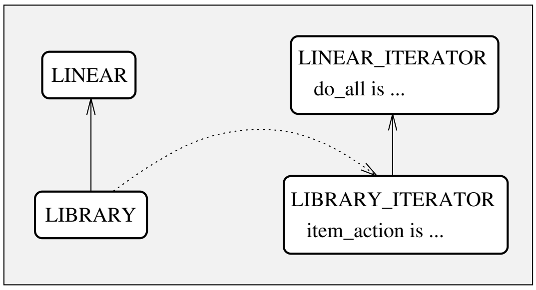


Figure 1: Iteration with Inheritance

An example are the `ITERATOR` classes in the Eiffel libraries [Meyer 92, Meyer 94]. The class `LINEAR_ITERATOR` can iterate over linear structures, that are descendants of class `LINEAR`. It offers a variety of iteration facilities like `do_all`, `do_if`, `do_until`, etc. These methods depend on the abstract methods `item_action` and `item_test`. Figure 1 shows the classes involved in a library example. To iterate over a list of books, one creates a class `LIBRARY_ITERATOR` that inherits from `LINEAR_ITERATOR` and provides definitions for `item_action` and `item_test`. On instantiation of a `LIBRARY_ITERATOR` one passes the desired list of books (dotted line in Figure 1). Then any of the available iteration schemes may be invoked. Note that the action and the test to be performed on books is fixed in `LIBRARY_ITERATOR`. To have alternative actions and tests we may:

- Create more subclasses (siblings to `LIBRARY_ITERATOR`) like `LIBRARY_SPECIAL_ACTION_ITERATOR`.
- Let class `LIBRARY_ITERATOR` inherit repeatedly from class `LINEAR_ITERATOR`, each time providing different actions and tests.

## 2.2 Evaluation

Inheritance used for behavior refinement as above, does not fall into the category of non recommended uses [Rumbaugh 93] and conforms to both subtype- and "is-a"-relation [LaLonde 91]. Nevertheless, it also implies common problems of the inheritance relation [Snyder 86].

Aspects to be considered when using inheritance for behavior refinement are:

*Implicitness.* The actual methods to be performed are implicitly specified. Given the call `library.do_if`, the actual action and test depend on the dynamic type of `library`. The actual versions must be deduced from the actual subclass used.

*Many Subclasses.* For each desired action an additional subclass is needed. The name space for classes is unnecessarily cluttered with many identifiers for concrete subclasses.

*Locality.* Using repeated inheritance [Meyer 92], it is possible to collect all potential actions in one class, but this results in non-local design. Moreover, it requires verbose renaming of the repeatedly inherited methods. Unless inherited from action classes (requiring multiple inheritance), the actions defined in an iteration subclass can not be used in other contexts.

*Change-sensitivity.* All concrete subclasses are subject to be affected by changes to the superclass. This demands programmer discipline (not to depend on superclass details) or languages with a protecting inheritance interface (e.g. private part of C++ classes [Stroustrup 86]). Given an adjustable inheritance interface, it is still difficult to know in advance, which parts of the class should be visible to future heirs. A solution for these problems is proposed in [Mens 94].

*Encapsulation.* Refining behavior with subclassing corresponds to white-box reuse, considered to be inferior to black-box reuse [Johnson 88]. White-box reuse requires knowledge of the reused components. Due to the required modifiability of the reused components, their encapsulation is poor. Black-box reuse requires explicitly passing arguments, but demands less implementation-detail knowledge of the composer. Nevertheless white-box reuse is more flexible and therefore very suited for the early and middle stages of development. To use the words of Johnson and Foote: *"Black-box relationships are an ideal towards which a system should evolve."* [Johnson 88].

*Flexibility.* Inheritance allows radical redefinitions. In early phases of design it is a welcome fact, that by means of subclassing it is possible to radically change the behavior of a class, while reusing parts of its implementation.

# 3 Parameterization

Previous to the explanation of dynamic behavior refinement, we embed it into a classification of parameterization types. Figure 2 applies a general view on parameterizable classes.

## 3.1 Classification

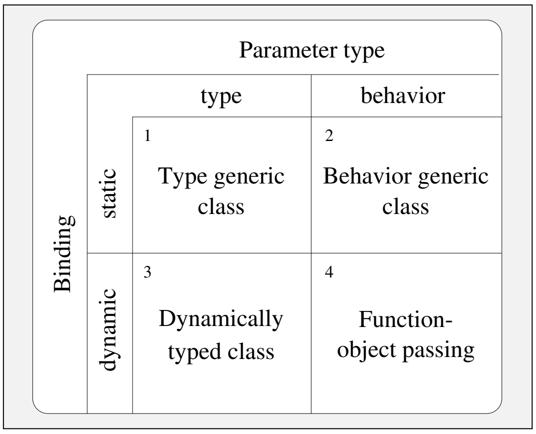


Figure 2: Parameter classification

Standard generic classes (1) are parameterized with types. Generic classes are conventionally used as code templates only. A generic class is used with different types, each time exposing identical behavior. Typical examples are container classes (e.g. `SET`, `LIST`, `ARRAY`), with passive generic elements.

It is in fact possible, to achieve behavior parameterization with standard generic classes. Consider a class `SEARCH` with type parameter `DISPENSER`, that explores nodes in a graph. Assuming the dispenser is used to accept encountered nodes, and to dispense memorized nodes in order to continue the search. Supplying a `QUEUE` or a `STACK` as `DISPENSER`, yields either breadth-first-search or depth-first-search. This might be regarded as static behavior parameterization (2), because the behavior passing must be done statically at declaration. In contrast to conventional uses of generic classes, the type parameter actively participates in the class's behavior.

Dynamically assigning a type to a class (3), is possible only in a few languages. For instance, Cluster [Shang 91] allows to assign a type to a generic class at runtime, allowing subsequent type safe uses.

As opposed to dynamic typing, *Function-object*-passing (4) is feasible in any object-oriented language. We discuss this parameterization type in the next section. Another potentially dynamic behavior parameterization method is the *handle-technique* [Meyer 94], also known as *Cheshire-cat implementation*. It is a parameterization of implementation alternatives. The *handle* is an attribute of an interface class. It holds a reference to an implementation of the interface. Of course, in this context, implementations are to be interchanged *preserving* behavioral identity.

Even inheritance may be viewed as a parameterization mechanism. In this sense, superclasses represent incomplete specifications that are parameterized with possibly multiple subclasses. This could be modeled by an attribute `subclass` in the superclass. The superclass resends all messages that address unspecified methods to the subclass attribute. In some languages (e.g. Objective-C [Cox 86]) it is even possible to dynamically check for the supported protocol of the `subclass` instance. This model demonstrates that parameterization introduces client-relationships only, since the dependency between super- and subclass is established via an attribute. The interface between the classes is not unrestricted, as is often the case with inheritance. It is determined by the export policy of the subclass. Parameterized classes therefore cause looser class-coupling than subclassing with unrestricted inheritance interfaces.

### 3.2 Principle

We recommend to implement variations of class behavior with parameterized classes. The parameterized class implements the common behavior. The specific variant is supplied by dynamically passing an object to an instance of a parameterized class. This *Function-object*[1], encapsulates the method to be performed. Both parameterized class and *Function-object* have to agree on a method name. This name is used by the parameterized class to call the encapsulated method in the *Function-object*. The *Function-object* might contain arguments in its attributes. These are supplied to the *Function-object* before it is passed.

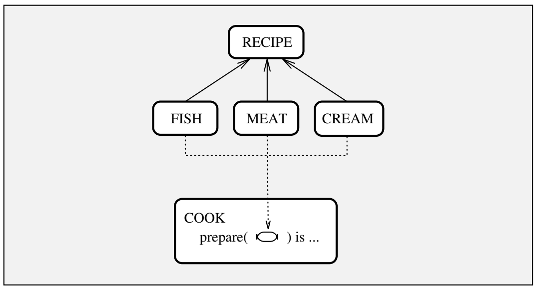


Figure 3: Dynamic behavior specification

An illustrative example for such a "takes-a" relation is depicted in Figure 3. The parameterized class `COOK` takes one of various `RECIPE`-*Function-objects*, in order to prepare the desired dish.

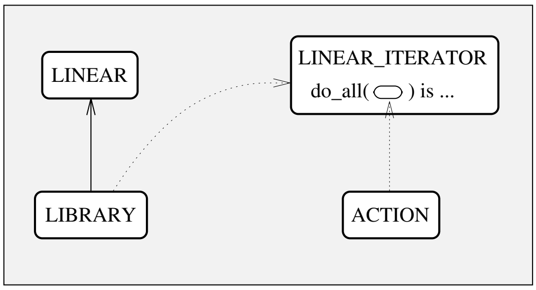


Figure 4: Iteration with Parameterization

[1]In [Kühne 94] we called it *Agent-Object*. Although an *Agent* may correctly be interpreted as a function with state [Nierstrasz 91a], we changed the name to avoid confusion with concurrent processes.

We now use *Function-objects* to model the iteration example from section 2.1. Figure 4 shows the corresponding situation. As before, class `LINEAR_ITERATOR` provides many useful iteration schemes. However, it does not have abstract methods `item_action` and `item_test`, to be defined by subclasses. Instead, the iteration methods are not parameterless, but take arguments `item_action` and `item_test`. The parameterized class `LINEAR_ITERATOR` and the *Function-objects* for actions and tests agree on a method name for calling the method encapsulated by a *Function-object*. Typically, this will be something like `evaluate`. A code scheme for `do_all` then looks like:

```
do_all(action : FUNCTION) is
  Loop over all elements
      action.supply(current_element);
      action.evaluate;
```

Alternatively, the method `evaluate` may take `current_element` as an argument. This would save the `supply` call in our example. Choosing the parameterless version of `evaluate`, however results in an homogeneous interface for *Function-objects*. Accordingly `supply` is restricted to one parameter only. Multiple parameters are supplied with subsequent `supply` calls. Consequently, all *Function-objects* conform to one protocol, that can be specified in an abstract *Function-object* class (see also section 3.4).

Instead of using side effects, `evaluate` may as well be a function returning a new `current_element`. This is preferable if the *Function-object* should also be applicable in cases when side effects on the argument are not admissible.

Given the above, it is possible to move the iteration facilities of `LINEAR_ITERATOR` to `LINEAR`. `LIBRARY` then directly inherits the iteration methods. This is feasible because we do not need to have different library types in order to apply various actions. Figure 5 depicts this simplified setting for both design alternatives. Now a typical iteration call looks like:

```
lib.do_if(book_lent, title_to_uppercase);
```

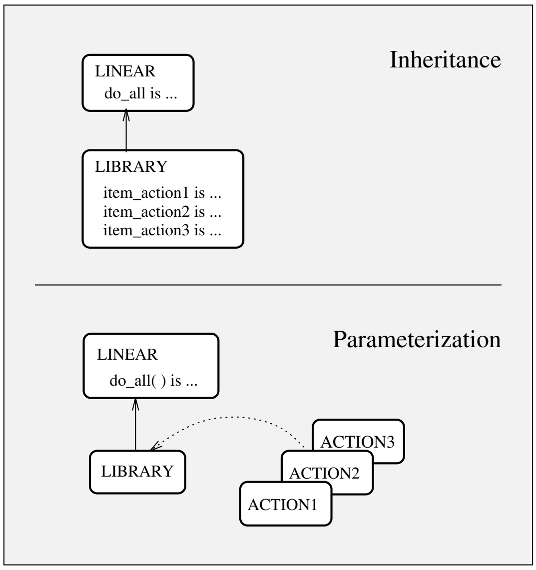


Figure 5: Simplified iteration

It is possible to express `do_if`, by `do_all`. This is done by parameterizing the action with the test.

```
lib.do_all(title_to_uppercase_if(book_lent));
```

Before applying the uppercase operation, the *Function-object* tests if the book is lent. Note that this example uses a `()`-operator instead of `supply`, in order to pass the test to the action.

The examples nicely demonstrate the behavior explicitness of the iteration calls. Remember that the original iteration call would have been `library.do_if`.

Note that there is no need for a `LIBRARY_ITERATOR` class. In order to apply various tests or actions we simply pass the appropriate *Function-objects*. In our inheritance example, we needed subclasses like `LIBRARY_SPECIAL_ACTION_ITERATOR` to apply different actions.

In fact, exploiting repeated inheritance, class `LIBRARY` might also directly inherit from `LINEAR` (see Figure 5), because action selection is possible via different names for one iteration scheme. The next section reveals the still remaining difference.

## 3.3 Evaluation

At first, the necessary agreement on the *Function-object* method name (`evaluate`) seems to be inconvenient. On the other hand, standard protocols and naming conventions are everywhere in object-oriented programming [Goldberg 83, Meyer 94, Rochat 86].

It may considered as a drawback that, prior to their usage, *Function-objects* need to be declared and instantiated. Instantiating the correct *Function-object* however corresponds to instantiating the correct subclass in the inheritance model. Many declarations caused by heavy usage of *Function-objects* might be alleviated by providing a globally know *Function-object* variable. This could be instantiated to the desired type for each usage.

We now precise the difference between inheritance and parameterization. In the simplified library iteration example, inheritance refinement combined with repeated inheritance seems to involve the same degree of inheritance relations, as is the case with *Function-objects* (see Figure 5). Nevertheless, there is a difference. In both instances, `LIBRARY` inherits from `LINEAR` in order to reuse the iteration schemes available in `LINEAR`. The inheritance example however additionally exploits the inheritance relation to provide various actions.

Now consider a very special class `COLORED_HYPER_GRAPH` that needs to implement its own iteration schemes. In that case, there is no inheritance relation, using *Function-objects*. Actions are passed directly to `COLORED_HYPER_GRAPH`.

Behavior refinement with inheritance needs to subclass `COLORED_HYPER_GRAPH` at least once in order to provide tests and actions.

Summarizing the above observations, *Function-objects* in principle allow to eliminate all iteration classes (`LINEAR_ITERATOR`, `CURSOR_TREE_ITERATOR`, etc.), putting all iterating schemes directly into the structure classes (`LINEAR`, `CURSOR_TREE`, etc.). This is feasible because heirs (`LIBRARY`) of structure classes (`LINEAR`) need not to provide iteration action refinements.

Inheritance refinement would require to provide iteration actions in the heirs (`LIBRARY`) of structure classes (`LINEAR`). This is not desired and in fact, worsens the more different actions are needed.

So, advantages of *Function-objects* are:

*Explicitness.* The action to be performed is explicitly visible in the calling statement. The purpose of the call is thus visible at once.

*Adaptability.* The action to be performed can be easily replaced, and may be changed dynamically. In addition, *Function-objects* may dynamically supplied with data (e.g. user input).

*Simplicity.* Just as constrained genericity saves subclasses (present in Eiffel, although redundant to inheritance [Meyer 86]), no refinement-subclasses are needed using *Function-object* passing. Simpler class hierarchies are the consequence (compare Figure 1 with Figure 4).

*Security.* There is no danger in internally changing the parameterized class, because there are no dependent subclasses.

*Decoupling.* *Function-objects* introduce a client-server relationship ("takes-a"), instead of an ancestor-heir relationship ("is-a"). Classes are therefore less tightly coupled.

*Restriction.* In contrast to subclassing, *Function-object* passing guarantees that nothing apart from the superclass's action behavior is redefined. Unless desired this restriction can be a disadvantage compared to the flexibility of subclassing.

*Reusability.* *Function-objects* are useful and therefore reusable in more than one context. A comparison predicate may be used for e.g., sorting and searching. Imagine a predicate used to sort a list of books, according to the author-names. Partially parameterized with one author-name the same predicate can be used to navigate in a search-tree of books, in order to find books written by the previously supplied author; e.g. `lib.search(compare.supply("Wells, H.G."))`. Innumerable uses are possible for a predicate `negate` that, wrapping any other predicate, inverts a boolean result.

*Generality.* Iterating schemes like `do_if` can be expressed by `do_all`. `do_if` acts like `do_all`, applying the action only if a test is satisfied [Meyer 94]. The test, executed by `do_if`, can be made a parameter to the action-*Function-object* (see the example in section 3.2).

*Compositionality.* *Function-objects* can be composed. A *Function-object* wrapping at least two sub-*Function-objects* can compose the single behaviors to a more complex one. For instance, the *Function-objects* for comparing books according to title and author-name, can be composed to a *Function-object* that first compares the author-names and on equality, compares the titles.
A multi-media *Function-object* may be composed of visual- and audio-*Function-objects*. This compositionality of *Function-objects* further enhances the flexibility of behavior parameterization.

*Abstraction.* Functional programming provides many useful abstractions, such as *fold* [Bird 88]. These should be used in object-oriented programming as well. The justification for these types of *tools* or *design objects*, like iterators and parsers, is given in [Meyer 94].

## 3.4 Applications

Possible applications for *Function-objects* are:

*Algorithm selection.* Depending on current requirements (e.g. speed or accuracy) one of several (e.g. Postscript) display engines, can be selected at runtime.

*Adaptable tools.* For instance, a `SORTING`-class can be parameterized with the compare method. Then changing the sorting criterion (e.g. from book-title to book-author) simply consists of passing a different *Function-object*.

*Higher order programming.* *Function-objects* provide higher order functionality for applications in the functional (e.g. *map*-, *filter*-, and *fold*-functions) and mathematical domain (e.g. integral- and differential calculus). Predicates and actions for higher order functions may be taken from a typed hierarchy of

*Function-objects* [Kühne 94]. We think that the above higher order functions are exactly the type needed for reusable design. They comply with the object-oriented seek for generalized classes. These are more general than required for a particular application. Depending on the function supplied, *fold* may act as *map*, *filter* or some completely different function, like *sum* or *length* [Bird 88].

*Functional mappers.* Using functional mappers instead of iterators facilitate the usage of parallel threads [Baker 93]. Similarly, mappers enhance maintainability by their referential transparency.

*Partial parameterization.* Partial parameterization allows to extend functionality, while providing uniform interfaces. Excellent examples are given in [Dami 94], which introduces curried functions as a language extension to C and C++. For instance, a *Function-object* printing a string to a tty-terminal, is easily replaced by a *Function-object* popping up an alert-window containing the string. The invoker of the *Function-object* remains unchanged. It still supplies the string only. Additional arguments, like window size, etc., are passed to the *Function-object* in advance. Partial parameterization can be implemented in two ways. The received arguments can be counted and are assigned to internal arguments according to their number. Alternatively, the result of a `supply` call can be a new *Function-object* (of different class type), that stores the received argument and expects the rest of the argument list.

*Multi-method dispatch.* It is possible to emulate multi-method functionality with standard single dispatch. Using the second implementation scheme for partial parameterization (as described above), the class type of the new *Function-object* can depend on the received argument type. As a result, the reaction to arguments supplied to the *Function-object*, depends on the type of arguments previously supplied. The multi-method call `plus(a, b)` translates to `plus.supply(a).supply(b)`, resulting in an object, suited to add the actual types of `a` and `b`. This combination of a curried object and type-sensitive object creation is facilitated by languages that allow overloading method names, though feasible in e.g. Eiffel as well.

*Behavior extraction.* Although it seems to be contra-object-oriented to extract behavior from data, there are useful applications for this technique. Consider a set of geometric objects needed to be output on both textual and graphical terminals. Instead of putting both display methods (and in general, all possible behavior) in all geometric objects, one can pass the appropriate *Function-object* to the set of geometric objects. The *Function-object* accesses the object to be displayed in a controlled manner through its interface. It is easy to add a *Function-object* (e.g. displaying colored-graphics) when needed. Essentially, this example demonstrates the separation between model (the data) and view (the display alternatives).
Behavior extraction may also be useful, in order to deal with behavior (e.g. to store or undo actions). Meyer extracts the commands of an editor into a `COMMAND` class to implement an undo mechanism [Meyer 88].

*Callback functions.* Event-driven applications like user-interfaces are typically customized using function values. *Function-objects* make pointers to (e.g. C++ member-) functions with their related problems superfluous.

*Frameworks.* Black-box frameworks provide plug-compatible components [Johnson 88], which may be composed by scripting [Nierstrasz 91b]. *Function-objects* are perfect candidates for such components.

*Language definition.* The generation of programming environments may demand that attributes of an attribute grammar need to store functions. Just like a functional language, *Function-objects* enable partial evaluation and direct implementation of denotational semantics.

*Object composition.* An extreme approach has been pursued by Zurek [Zurek 94], in order to compose functional descriptions. Zurek implements *every* action as a *Function-object*, thus stripping all actions from standard objects, leaving them as mere data containers. The purpose of this, is to easily and dynamically compose already existing objects to complex objects. The composed objects offer more complex behavior (provided by a *Function-object*), based upon the behavior (*Function-objects*) of the contained objects.

# 4 Related Work

We already criticized the behavior refinement approach of Meyer [Meyer 94] in section 2.2. Using inheritance, modularity and protection are not assured. Inheriting solely from abstract classes removes at least implementation detail dependencies, but all other mentioned problems remain. In addition, although desirable it is often impractical to inherit from abstract superclasses only [Huersch 94].
Under a closed-world assumption, classes, such as `ARRAYED_STACK` that incorporate implementation detail might be feasible. They are not acceptable in projects, which are still in the phase of development. If the chosen implementation (`ARRAY`) has to be changed then dependent classes (`ARRAYED_STACK`) might be affected. It is unrealistic to regard any class as perfect in any stage of development including the final product. As a consequence such unnecessarily tight coupling between classes should be avoided.
Another drawback of `ARRAYED_STACK` is that it might be substituted for an `ARRAY`, because it is a proper heir. Of course, this is unwanted. Moreover, class hierarchies that do not model subtype relations are likely to be misinterpreted by an uninformed user. In our opinion, applying system-level validity checking [Meyer 92] is too overpowered to detect misuses of inheritance relations as described above. It does not allow modular type checking and rejects some programs that would not produce runtime type errors.

In [Kühne 94] we gave a recipe how to translate functional expressions to object-oriented languages. We demonstrated partial parameterization and presented how to integrate a type system into the method.

The work presented here extends the above topic to a design issue. We thoroughly compare parameterization with inheritance, with a strong bias towards the usefulness of parameterization in object-oriented design and programming. Furthermore, we relate *Function-objects* to recently published design patterns.

Our method is familiar of course to Smalltalk programmers. Smalltalk offers blocks and therefore a means to pass behavior [Goldberg 83]. In contrast to *Function-objects*, blocks must be explicitly written every time they are used, because there is no such thing as a block library. Moreover, they can not easily be used for partial parameterization.

After writing [Kühne 94], we discovered that Coplien also uses objects as functions [Coplien 92]. The principle, however, is a bit hidden by a not very revealing example. The technique is proposed to avoid function pointers and to enable dynamic dispatch. It is not regarded as a design issue and therefore not compared to inheritance.

Watt et. al. [Watt 94], Dami [Dami 94], and Klagges [Klagges 93] see a need for integrating functional programming into object-oriented programming. However, they provide extensions, resulting in hybrid languages. In contrast, we suggest using existing languages without modification.

Sandberg introduces descriptive (generic) and parameterized classes to allow compile-time checking and to avoid subclassing [Sandberg 86]. His main emphasis is on the advantages of descriptive classes, but he also already introduces behavior parameterization in the *Strategy*-style.

Gamma et. al. [Gamma 93] enumerate some design patterns that solve similar problems as *Function-objects*. In particular, *Command*, *Strategy*, and *States* have like purposes and may all be modeled with *Function-objects*. Among the advantages the authors list for these patterns are: No subclasses are required and hence, the usage of foreign libraries is possible without access to source code. Naturally, these advantages also apply to *Function-objects*.
*Strategies* are a combination of *Commands* and the *handle-technique* [Meyer 94]. The

*handle-technique* allows to transparently change the implementation of a class. *Strategies* aim to dynamically provide variable behavior, rather than implementing the same behavior differently. Instead of passing a *Command* to the common behavior object (*Strategy Context*), this object contains an attribute (the *handle*) for the *Command*. As a result, the attribute must be set previous to client service calls. Thus, *Strategies* also provide dynamic instantiation of behavior, but unless the behavior is explicitly set prior to the service call, the chosen behavior remains implicit. Apart from these minor differences *Strategies* and *Function-objects* similarly address the same problems. Yet, *Function-objects* do not involve one disadvantage Gamma et. al. identify with *Strategies*. If the available *Strategies* range from simple to complex, the *abstract Strategy* must support the maximum parameter interface. Therefore, the whole parameter list must be given even with simple calls. *Function-objects* may be supplied with parameters before they are passed. Consequently, they always expose identical parameterless interfaces, independent from the number of arguments. Gamma et. al. recommend the use of *Strategies*, when many related classes differ only in their behavior, but do not provide the complete discourse of advantages and disadvantages.
*Commands* are essentially *Function-objects* for the specific application in toolkits. Gamma et. al. list the following advantages, which also apply to *Function-objects*. A *Command* decouples the object that invokes an operation from the one that knows how to perform it. *Commands* can be assembled to *MacroCommands*. The introduction of new *Commands* does not require to change the invoker [Gamma 94]. However, *Commands* are not intended to take arguments and do not feature partial parameterization. As with any design pattern, the *Command* pattern is *problem* oriented[2]. Therefore, the authors did not investigate on the potentials and possible applications of the particular *solution* underlying the *Command* pattern.

Pree reduces the number of relevant patterns for framework refinement to seven meta patterns [Pree 94]. Yet, he only considers template- and hook-method distribution, for the identification of *hot spots* in frameworks. Dynamic refinement with prefabricated *Function-objects* is not included in his restriction to a few basic meta patterns.

Summarizing, our presentation is strongly related to design patterns [Gamma 93], that solve various problems, also addressed by *Function-objects*. Yet, we elect to focus on a particular micro-architecture; the dynamic passing of behavior. Dynamic behavior passing is an available *solution*. We consider it to be finer in granularity of design than common design patterns, which are *problem-solution* pairs.

A consequence not mentioned by the authors of [Gamma 93] is the usefulness of *Commands* or *Function-objects*, to construct black-box frameworks [Johnson 88]. All the positive properties of programming with *Function-objects* do apply to the construction of frameworks in a larger scale. Conventional white-box frameworks rely on being refined by subclassing. Therefore, the programmer has to be aware of, and worse yet, is dependent on the implementation of the white-box framework. This disadvantage is also known as white-box reuse, since it is necessary to first modify a component in order to use it. By comparison, black-box frameworks provide plug-compatible components (some of which could be *Function-objects*) that allow to simply configure a specific application from available blocks. The configuration thus might be called *scripting*, instead of programming [Nierstrasz 91b]. Consequently, the major part of building an application from a black-box framework, might exclusively be done with graphical editors that use a visual scripting language. Note that white-box frameworks demand for the generation of subclasses. Scripts then would need to generate subclasses, and would need to provide the code to be inserted in subclasses. Ideally, a framework realizes a homomorphism between problem- and solution-space. If the real-world model is extended with a component, this should result in adding a component to the framework.

[2]For instance, *Strategy* and *State* are basically the same solution to different problems

Nierstrasz recognizes the lack of binding technologies for composing objects [Nierstrasz 91b]. Passing *Function-objects* is a method to compose existing objects without writing new classes, obeying the strict version of the *Law of Demeter* [Lieberherr 89]. The functional construction, using *Function-objects* and dynamically parameterizable classes, should provide the necessary precondition for composing reusable components via scripts.

Tsichritzis et. al. also postulate a compositional approach instead of inheritance [Tsichritzis 92]. They favor software development that is *capital-intensive* (producing reusable components) instead of *labour-intensive* (producing applications). Again, *Function-objects* may provide a basis for this compositional approach.

Budde et. al. introduce an interesting design metaphor. They divide classes into Tools and Materials [Budde 92]. This categorization is context sensitive. As a Tool, a class uses Aspect classes to modify Materials. As a Material, a class inherits from Aspect classes. Aspect classes provide a view and a manipulation-interface on Materials. Our method provides a useful extension to this metaphor. Parameterized classes are to be interpreted as adaptable Tools.

Last but not least, our method is strongly related to functional languages that are extended with object-oriented features. CLOS [DeMichiel 87] is a prominent example for these types of languages that naturally use functional style, regarding the object-oriented features as extensions. Our work is very different to these approaches of combining two paradigms. First, we start with an object-oriented basis. Second, we do not add anything to it. Moreover, we basically address imperative languages. Our method leads to encapsulated imperative entities that may be composed in object-oriented or functional style. Modeling state in functional languages, e.g. using monads [Wadler 90, Wadler 92] based on category theory, seems much harder. Of course our argumentation also applies to non-imperative languages, but these often natively use functional composition.

# 5 Conclusion

Apparently our idea to use objects as functions is not original, since there have been similar applications of it before [Coplien 92, Gamma 93]. We have however put the concept of dynamic behavior parameterization into the spotlight, in order to establish it as a basic and useful design technique. We presented many applications and explicitly demonstrated the potential advantages over conventional inheritance. Functional abstractions (e.g. *fold*) are of such general usefulness that they should not be neglected by the object-oriented community.

Of course inheritance is not replaceable by parameterization. White-box reuse is of particular value in experimental phases of development. In contrast to black-box reuse, it allows full control for changes avoiding early restrictions on possible variations. Premature decisions concerning the fixed and variable parts of a class will likely hinder reuse. In a strict black-box reuse situation the fixed parts cannot be altered without a redesign from scratch. In addition, the variable parts may evolve to fixed parts. However, if the design is understood well enough, we think it is more reasonable to reuse design than to reuse code. Ideally, classes should be specifications, providing true substitutability. This can be realized by using inheritance for subtyping as defined in [LaLonde 91]. Then reuse does not mean the introduction of dependencies. In practice, parameterization, subtype inheritance, and implementation inheritance will coexist. Each should be used accordingly, in order to meet requirements like simplicity, reusability, and flexibility.

While we do consider Eiffel as one of the best object-oriented languages available, we disagree with the behavior refinement for classes used in the Eiffel libraries. As we pointed out, this method produces many change-sensitive classes. Also, we have seen that repeated inheritance is not a remedy for behavior refinement with subclassing. The construction (should read *evolution*) of reusable software is an inherently iterative process. Avoiding dependencies between classes is therefore of vital importance.

*Function-objects* enhance the flexibility, maintainability, and reusability of applications. Class hierarchies become less complicated, more reusable, and less change-sensitive. *Function-objects* can be composed, are reusable, and have many more advantages (see section 3.3). So, why are *Function-objects* not widely used already? We can think of three reasons. First, classes are often regarded as implementations of abstract data types. Abstract data types usually do not involve higher order functionality. Second, inheritance is often the first choice. If inheritance can be used, it will be used, distinguishing the program from non-object-oriented ones. Third, the method to achieve higher order functionality is not apparent in most languages (Smalltalk that features blocks being an exception).

We demonstrated the applicability of our method to many different areas. Note that there apparently is a category of applications which exposes inversion of control. For instance, callback-functions for event-driven applications, framework refinement, and the parameterization of general algorithms. Each of these applications is traditionally solved with template methods relying on hook methods to be supplied. Hence, *Function-objects* provide a method to advance "reuse by inheritance" to "reuse by composition" [Johnson 94].

We regard behavior parameterization as a very natural extension to other forms of class parameterization already in use. We view type- and behavior-genericity, dynamic typing of classes, *Function-object*-passing, and even a restricted form of delegation as parameterizing classes with refining information. In contrast to inheritance ("is-a"), parameterization ("takes-a") creates client relationships. Consequently, inter-class coupling is much looser.

With this general view *Strategies* are a variant of the *handle-technique*. Instead of behavior conservation, by alternative implementations, the behavior is changed on purpose. *Function-objects* improve on *Strategies* by explicitness, the option of partial parameterization, and dynamic type change, e.g. for multi-method dispatch.

We argued that in order to have the most of both worlds "object orientation" and "functional programming", it is sufficient to use functional style in object-oriented design and programming. In this scenario the data-centered approach of object-oriented design still plays the dominating role, but is supplemented with functional techniques. These enhance the modularity, maintainability, and reusability of designs. The integration of functional techniques into object-oriented design is capable of producing encapsulated, imperative entities that may be composed in object-oriented or functional style. Behavior parameterization should therefore be part of the basis for plug-compatible components that can be composed by scripting.

As objects subsume functions, functional extensions to object-oriented languages are in principle unnecessary. We demonstrated this for higher order functionality and partial parameterization. By themselves, these of course do not cover all functional techniques applicable in object-oriented design. Future work could investigate further migration of functional techniques into object-oriented design. Although possible without language extensions, lazy evaluation and infinite data structures might better be tackled with language support.

## References


[Baker 93] Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS MESSENGER*, 4(3):18–25, July 1993.

[Beck 94] Kent Beck and Ralph E. Johnson. Patterns generate architectures. In Mario Tokoro and Remo Pareschi, editors, *ECOOP '94*, Lecture Notes in Computer Science, pages 139–149. Springer Verlag, July 1994.

[Bird 88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* C.A.R. Hoare Series. Prentice Hall International, 1988.

[Budde 92] R. Budde, M.L. Christ-Neumann, and K.H. Sylla. Tools and materials, an analysis and design metaphor. In *Prooceedings of the TOOLS 7*, pages 135–146. Prentice Hall, 1992.

[Coplien 92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

[Cox 86] B.J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Co, Reading, Mass, 1986.

[Dami 94] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, University of Geneva, April 1994.

[DeMichiel 87] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In J. Bezivin, J-M. Hullot, P. Cointe, and H. Liebermann, editors, *Proceedings ECOOP '87*, LNCS 276, pages 151–170. Springer Verlag, June 1987.

[Gamma 93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *ECOOP '93*, Lecture Notes in Computer Science, pages 406–431, July 1993.

[Gamma 94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1994.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Huersch 94] Walter Huersch. Should superclasses be abstract? In Mario Tokoro and Remo Pareschi, editors, *ECOOP '94*, Lecture Notes in Computer Science, pages 12–31. Springer Verlag, July 1994.

[Johnson 88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Johnson 92] Ralph E. Johnson. Documenting frameworks using patterns. *ACM SIGPLAN NOTICES, OOPSLA '92*, 27(10):63–76, October 1992.

[Johnson 94] Ralph E. Johnson. How to develop frameworks. In *ECOOP '94 Tutorial Documentation*, July 1994.

[Klagges 93] Henrik Anthony Klagges. A functional language interpreter integrated into the C++ language system. Master's thesis, Balliol College, University of Oxford, Oxford Univerity Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, United Kingdom, September 1993.

[Kühne 94] Thomas Kühne. Higher order objects in pure object-oriented languages. *ACM SIGPLAN NOTICES*, 29(7):15–20, July 1994.

[LaLonde 91] W. LaLonde and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[Lieberherr 89] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.

[Mens 94] Tom Mens and Marc Van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. Technical Report vub-prog-tr-94-10, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1994.

[Meyer 86] Bertrand Meyer. Genericity versus inheritance. In Norman Meyrowitz, editor, *OOPSLA '86*, pages 391–405, September 1986.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Meyer 92] Bertrand Meyer. *EIFFEL the language*. Prentice Hall Object-Oriented Series, 1992.

[Meyer 94] Bertrand Meyer. *Reusable Software*. Prentice Hall, 1994.

[Nierstrasz 91a] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612. Springer Verlag, July 1991.

[Nierstrasz 91b] Oscar M. Nierstrasz. Objects + Scripts = Applications. In *Esprit 1991 Conference*, pages 534–552. Kluwer Academic Publishers, 1991.

[Pree 94] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In Mario Tokoro and Remo Pareschi, editors, *ECOOP '94*, Lecture Notes in Computer Science, pages 139–149. Springer, July 1994.

[Rochat 86] Roxanna Rochat. In search of good smalltalk programming style. Technical Report CR-86-19, Computer Research Laboratory, September 1986.

[Rumbaugh 93] James Rumbaugh. Modeling & design. *Journal of Object-Oriented Programming*, 5(9), February 1993.

[Sandberg 86] David Sandberg. An alternative to subclassing. *OOPSLA '86*, 21(11):424–428, September 1986.

[Sargeant 93] John Sargeant. United functions and objects: an overview. Technical Report UMCS-93-1-4, University of Manchester, Manchester M13 9PL, England, 1993.

[Shang 91] Lujun Shang. Cluster: An informal report. *ACM SIGPLAN Notices*, 26(1):57–76, January 1991.

[Snyder 86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *Proceedings OOPSLA '86*, 21(11):38–45, November 1986.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tsichritzis 92] Dennis Tsichritzis, Oscar Nierstrasz, and Simon Gibbs. Beyond objects: Objects. *International Journal of Intelligent and Cooperative Information Systems*, 1(1), 1992.

[Wadler 90] Philip Wadler. Comprehending monads. In *Conference on Lisp and Functional Programming*, June 1990.

[Wadler 92] Philip Wadler. The essence of functional programming. *ACM POPL*, January 1992.

[Watt 94] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A first report on the A# compiler. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, July 1994.

[Zurek 94] Jan Zurek. Komposition funktionaler Beschreibungen. Diploma thesis (in german), Technische Hochschule Darmstadt, to appear 1994.