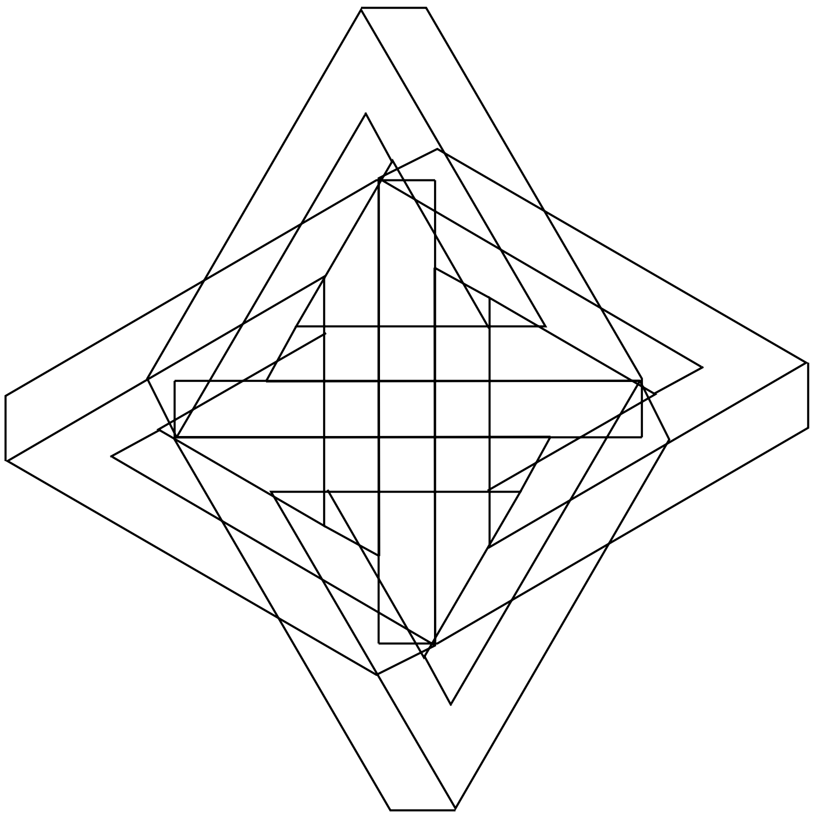


A Functional Pattern System for Object-Oriented Design

Thomas Kühne

◊

A child
does not dis-
cover the world by
learning abstract rules.
Instead it learns by looking
at concrete examples. An example
contains the rules as well. In contrast to
rules, the recognition of examples can be based
on tangible reality. The knowledge extracted from
an example serves as a *Pattern* that is used to remember
facts and to construct new solutions. When grown-ups
are about to learn something or have to apply
unknown tools, they are put into a child's
position again. They will favor concrete
examples over abstract rules. The
rules will happily be gen-
erated automatically,
for this is how
the brain
works.

◊

Thesis

Design patterns inspired by functional programming concepts can advance object-oriented design.

Problem

The object-oriented paradigm has undoubtfully raised our ability to design and maintain large complex software systems. However, it does not seem to have meet the high expectations concerning reuse and ease of evolution which have been promoted ever since its commercial success.

There are many potential reasons for the above observation such as unqualified staff, immature languages, inadequate methodologies, inappropriate business processes, etc.

The view presented here is that although the object-oriented paradigm is a powerful basis, it is incomplete in its inherent concepts and therefore restricts the design space to inappropriate solutions. It is assumed that both software development and language design are restrained from achieving their full potential when restricted to a purely object-oriented world view.

Solution

Since the complementary paradigm to object-orientation is represented by functional programming, I investigate high-level, well-known to work functional concepts and examine their suitability to enhance object-oriented design. I explore the software engineering relevance of each concept and present its intent, applicability, implementation, and consequences in the literate form of a design pattern.

My approach clearly motivates functional techniques for object-oriented design from a software engineering point of view. This is different to the usual procedure of designing a new language with an “ad-hoc” conglomeration of functional and object-oriented features. The latter case requires excellence in language design and makes it hard to find out and evaluate uses of the new language.

In contrast, design patterns are already widely used to improve design. As functional concepts constitute a powerful paradigm by themselves, it is more than suggestive to assume that design patterns expressing successful functional concepts will enhance the object-oriented paradigm with new capabilities.

Contribution

Feasibility

I demonstrate the feasibility of using functional techniques in object-oriented designs which are to be implemented by ordinary object-oriented programming languages. This is done at the level of a calculus comparison and in concrete design pattern implementation descriptions. I demonstrate synergetic effects caused by concept integration, which together with the advantages of functional patterns, thus, show the utility of the approach.

Software production

Object-oriented practitioners hopefully will use the names of functional design patterns as a vocabulary to discuss solutions in a new design space. I present a system of patterns which are connected by relations that describe how individual patterns may interact and collaborate with each other. This system, consisting of state-of-the-art mini-architectures, may allow thinking and designing beyond restrictions imposed by a dogmatic object-oriented approach. As a result, the quality of software is hoped to improve.

Language Design

Using functional patterns for object-oriented design can be regarded as dual-paradigm design. In this light, functional design patterns appear as language idioms that lift an object-oriented language to a dual paradigm implementation language.

It is very instructive to verify how well an object-oriented language supports the implementation of these idioms, since limiting properties are expected to interfere in other attempts to produce flexible and maintainable software as well.

Unless one is restricted to use a certain existing language, it is, however, only natural to consider direct language support in order to avoid the repetitive implementation of these idioms. A holistic language that encompasses both object-oriented and functional paradigms should provide more ease of use, increased safety, better initial execution efficiency, and higher optimization potential.

I consider each presented design pattern for its contribution to language constructs that support a dual paradigm language. The software engineering considerations, contained in each design pattern description, help to avoid “featurism” in favor of conceptually founded language principles.

Ultimately, impulses initiated by the functional pattern system lead to a reevaluation of the role distribution between a programming language and its associated environment. The result allows transcending the limitations of each paradigm by providing the optimal paradigm view on demand.

Preface

Example is the school of mankind, and they will learn at no other.
– Edmund Burke

A number of people directly or indirectly influenced my work and I am grateful for their contributions.

Norbert Ihrig made my early years in school a worthwhile experience and without him my way would have been much harder.

Gregor Snelting proposed an interesting master thesis to me and, thus, opened up the subsequent opportunity for a research position at the institute for “Praktische Informatik” at the Darmstadt University of Technology. I enjoyed his lectures and am thankful for his continued support.

Thilo Kielman introduced me to the world of being a scientist. I owe him many hints, paper reviews, and enjoyable hours.

Alberto Pardo was my roommate over a long period and shared many stimulating discussions with me. I would like to express my gratitude for his ever present willingness to help and for a lot of advice in the areas of calculi and algebras. He and Nick Dyson helped to improve the chapter on functional programming.

John Sargeant invested considerable effort to discuss the Void Value pattern with me and provided useful information on UFO.

Dirk Koschorek provided invaluable last minute advice on SMALLTALK.

Many scientists, who I had the honor to meet at workshops or conferences, discussed aspects of my work with me and made me reflect about it.

Gillian L. Lovegrove and Jürgen Ebert accepted to be external examiners and I am indebted to their timely efforts. I would like to thank Jürgen Ebert for his invitation to the “Bad Honnef” workshop series and for his personal interest in my work.

Finally, I would like to express my gratitude to my thesis supervisor Wolfgang Henhapl. He gave me all the freedom I could wish for and stimulated my work on several occasions. He asked the question to which function objects present an answer and challenged me to conceive the Translator design. Numerous discussions with him created an abundance of insights for me.

I am honestly grateful to the nameless cosmic particles that hit me when I had the idea of function objects, to replace Nil with a type specific value, to solve the forces of internal and external iteration with multiple consumable intermediate streams, to express functional ideas by devising a pattern system, and to rethink the roles of languages and their environments based on the notion of tiles.

Many thanks to Cordon Art for granting permission to use the images by M.C. Escher® “Circle Limit IV”, “Space Filling”, and “Day and Night”. © 1998, Cordon Art B.V., Nieuwstraat 6, P.O.Box 101, 3740 AC Baarn, Holland.

Last but not least, my appreciation goes to to Donald E. Knuth who developed TEX and put it into the public domain, Frank Mittelbach for LATEX 2$_\varepsilon$, Linus Torvald for LINUX, and all their companions and successors who enabled me to produce this document with copylefted software only.

Contents

List of Figures

List of Tables

Prologue

A paradigm is not a dogma.
– me

Paradigms define ways to perceive the world. Is a cannery a hierarchical structure of functions defined on passive data, or a set of interacting active objects, or a web of concurrent processes? In analogy to theories, the value of a paradigm is determined by the usefulness of creating models according to it. A paradigm, therefore, is a means to an end. In software development the goal is to avoid semantic gaps inbetween analysis, design, and implementation while achieving reusable components, extendable applications, and maintainable software.

Motivation

During its now three decades spanning history, the object-oriented paradigm has proved to be tremendously useful as a basis for development methodologies and as an implementation strategy. Object-oriented languages are truly general purpose languages since they may conquer any domain by defining the appropriate abstractions. Domain specific libraries or frameworks enable designers to define and use *application languages*, specifically tailored to model particular domains [Hudak96]. Application languages avoid an impedance mismatch between analysis and implementation. Object-oriented languages to a large extent allow a natural definition of application languages with the help of class abstractions. They, hence, enable a so-called *middle-out* development [Ward94], where one team implements the application language while another builds the application on top of it. It appears to be almost certain that object-orientation will not cease as a dead end in the evolution of programming and design paradigms.

However, there are signs of a disintegrating kingdom. The fact that the next Kuhnian paradigm shift [Kuhn70] is overdue [Quibeldey-Cirkel94] is not a strong indication, but in 1994 an issue of BYTE magazine devoted to component software shook the object-oriented community by stating that object-oriented technology failed to keep its promises and that VISUAL BASIC-like, component oriented languages offer higher productivity [Byte94]. Subsequently, the functional paradigm scored higher in analysis suitability [Harrison et al.94] and the supremacy of the object-oriented paradigm was found to be "an open research issue” [Menzies & Haynes95]. Further competition appeared promoting “Subject-oriented programming” [Harrison & Ossher93, ObjectExpert97].

It is not my intent to refute alternative approaches to be futile, but this thesis makes a definite statement that first, object-orientation is still well in the game and second, should be the basis for an evolution rather than a revolution. But why did object-orientation not live up to its promises? A possible answer is given the the August 1995 issue of IEEE COMPUTER:

> *“Poor design is a major culprit in the software crisis.”* – Bruce W. Weide

Or, more elaborate:

> *“But simply using an object-oriented programming language or environment does not, in itself, guarantee miraculous results. Like any other human endeavor, software design is an art: discipline, hard work, inspiration, and sound technique all play their parts. Object-oriented technology has much to offer, certainly. But how may it best be exploited? [Wirfs-Brock et al.90]”*
> – Rebecca Wirfs-Brock et al.

This question is still unanswered as evidenced by the 1997 call for papers of the OOPSLA ’97 workshop “Object-Oriented Design Quality”:

> *“Despite the burst in the availability of OO analysis and design methodologies, languages, database management systems, and tools, relatively little work has been done in the area of OO design quality assurance, assessment, and improvement. We badly need a better understanding of the properties of OO system design, in the small and in the large, and their effect on quality factors such as maintainability, evolvability, and reusability.”* – Rudolf K. Keller

Design patterns [Gamma et al.94], i.e., the documentation of successful mini-architectures, are hoped to be part of the solution to the persisting design problem. They can be used to hand down design excellence from experts to novices. Indeed, pattern books may contribute to make software engineering a true engineering science by representing engineering handbooks containing known to work solutions. In fact, patterns can make software more reusable, allow reusing of design, aid in designer communication, and can be used in teaching good design.

Paradigm integration

While object-orientation and design patterns — the buzzwords of the eighties and nineties respectively — receive a great deal of attention, it got rather quiet around a field which has mostly been of academic interest only: Functional programming is known for its elegance but also has the reputation of being an academic toy only. Quite often the desirable features of functional programming are believed to be inaccessible from other programming language types.

> *"Functional programming can not, apart from modest attempts, be applied with algorithmic programming languages in common use*[1] *[Schneider91]."*
> – H. J. Hoffmann

Especially higher-order functions, i.e., the ability to support closures, are a desirable feature, believed to be absent in programming languages like C^{++}, EIFFEL, and so on [Baker93, Kofler93, Gamma et al.94].

Continuing attempts to explain potential benefits of functional programming [Hughes87], this thesis uses the literate form of design patterns to capture the benefits of functional concepts and idioms. It, therefore, demonstrates the feasibility of supporting a functional programming style in the above mentioned languages. Of course, it is not possible to turn stateful languages into pure functional programming languages. This is not intended, however.

> *"This discussion suggests that what is important is the functional programming style, in which the above features are manifest and in which side effects are strongly discouraged but not necessarily eliminated. [Hudak89]."*
> – Paul Hudak

Although some of the functional patterns to be presented introduce restrictions to gain safety, they are intended to be additions rather than constraints to existing object-oriented practices.

Prior to the formulation of patterns an analysis of the two paradigms is performed with the view to embed functional programming into the object-oriented paradigm. This is, as such, a valuable endevour.

> *"We need to find, generalize, and integrate similarities in programming languages. Otherwise, there will be a proliferation of concepts which nobody will be able to overview and understand [Reynolds96]."* – John C. Reynolds

In addition to that, the world of object-oriented design is enriched with the functional paradigm. The result is a multi-paradigm design space. The author is in good company in his belief that multi-paradigm programming will play a crucial role in the future:

> *"... mixed language working — "integration of programming paradigms" is an important theme [Hankin et al.97]."* – Chris Hankin et al.

A multi-paradigm toolkit does not only represent an enhanced choice of tools but also enhances the tool user.

> *"Research results from the psychology of programming [Petre89] indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual that it is to the number years experience in programming [Budd95]."* – Timothy Budd

[1]Translated from German.

This can be explained by the fact that the available notions of a particular programming language restrict the solution space of a designer who is familiar with this language only.

> *"Wir müssen den starken und unbestreitbaren Einfluß unserer Sprache auf die Art unseres Denkens erkennen. Mit ihr wird in der Tat der abstrakte Raum definiert und begrenzt, in dem wir unsere Gedanken formulieren, ihnen eine Form geben."* – Nicklaus Wirth

In the context of this thesis it is appropriate to interpret the term "Sprache" (language) as "paradigm". The single paradigm possibly implied by even a group of programming languages, thus, restricts designers that are familiar with this paradigm only. The introduction of functional patterns, hence, enlarges the design space of formerly pure object-oriented designers. Interestingly, this is possible without requiring them to learn a new language, since the functional patterns are expressed with well-known object-oriented concepts.

The attempt to capture functional concepts as patterns, evaluate their software engineering properties and then document them as parts of a engineering handbook for design, is in tune with the primal definition of science.

> *"It is my aim to first enumerate experience and then to proof with reason why it must act in this way."* – Leonardo da Vinci

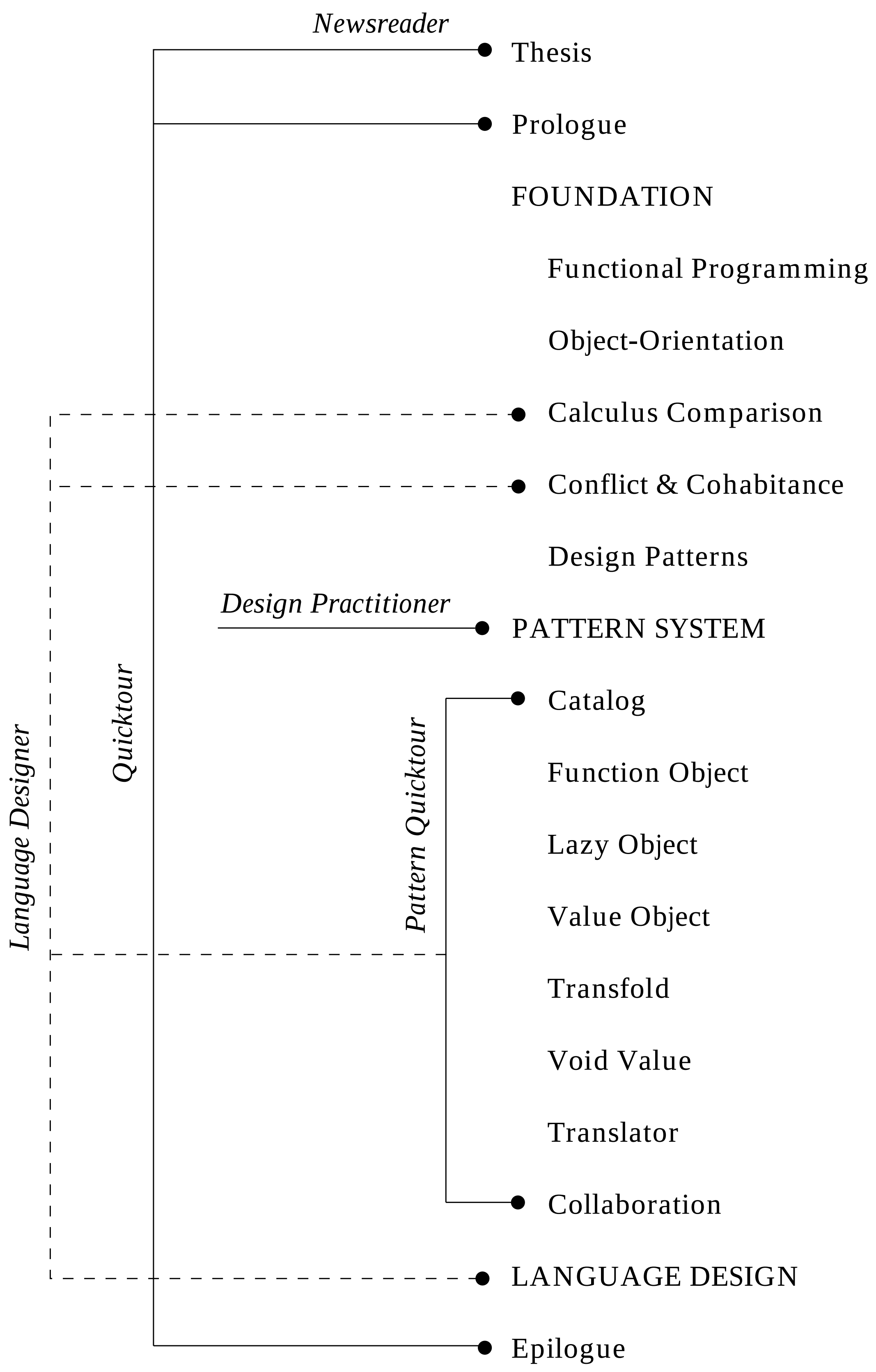
Design patterns can be a blessing for design practitioners but they are also often testimony to the weaknesses of implementation languages or underlying paradigms [Baumgartner et al.96, Seiter et al.96, Norvig96]. A further lesson to be learned from the functional pattern system is, therefore, how to better support it in future languages.

Hints on reading

The "Newsreader" approach to this thesis is to read chapter "Thesis" on page iii only. A more comprehensive quicktour is obtained by adding the prologue on page 1 and epilogue on page 261. Anybody interested in a glance at the patterns may take a pattern quicktour by reading chapters catalog on page 85 and collaboration on page 221. Design practitioners, however, will savor the whole pattern system part starting on page 85.

Chapters of interest to language designers are the calculus comparison on page 45, the analysis of paradigm integration on page 55, and the whole language design part beginning on page 233.

Chapters functional programming on page 9, object-orientation on page 29, and design patterns on page 69 are meant as an introduction to the uninitiated but also serve to establish and clarify terminology.



Newsreader
Thesis
Prologue
FOUNDATION
Functional Programming
Object-Orientation
Calculus Comparison
Conflict & Cohabitance
Design Patterns
Design Practitioner
PATTERN SYSTEM
Catalog
Function Object
Lazy Object
Value Object
Transfold
Void Value
Translator
Collaboration
LANGUAGE DESIGN
Epilogue
Language Designer
Quicktour
Pattern Quicktour

Part I

Foundation

1 Functional programming

This chapter introduces functional programming and defines some terms that are used in later discussions. I explain the underlying worldview (section 1.1) and the most important concepts (section 1.2 on the next page). Finally, section 1.3 on page 17 enumerates pros and cons of functional programming.

1.1 Worldview

The functional paradigm suggests to regard everything as an expression. This can be seen as the logical conclusion from Fortran (mathematical flavor through expressions), to functions in Pascal, to expression oriented style in Scheme [Hudak89]. Since the value of mathematical expressions does not change over time, there is no need for any notion of state. Also, there is no need for control flow, because the order of expression evaluation has no impact on the final result (as long as partiality is excluded).

The most important type of expression is called function application, i.e., some data is fed into a function and the function produces a result. We might say “The functional paradigm suggests to regard everything as a function” and loose only a little bit of truth. In fact, it is possible to model everything with functions only (e.g., with the pure λ-calculus [Barendregt84, Hankin94]), but functional programming languages typically provide built-in primitive data structures and operations.

Apparently, functional programming corresponds to two outdated software paradigms: On a small scale the transformation of data with functions corresponds to the Input/Processing/Output model [Pepper92], which has been replaced by more modern views such as distributed computation and event-triggered behavior. On a large scale the structuring of a program into a hierarchy of functions corresponds to structured analysis and design, which has been replaced by entity-relationship or object-oriented decomposition techniques.

The above should not create the impression that functional programming is of no use anymore. On the contrary, we will see that data transformation and functional decomposition are still very useful concepts. Also, apart from the ongoing academic interest in functional programming languages (e.g., [Hudak96, Gostanza et al.96, Läufer96]) the rise of new parallel computer architectures adds importance to languages with a strong potential for parallel execution models.

1.2 Concepts

The following sections are meant to briefly introduce concepts in functional programming. For a more thorough discussion and also for a history of functional programming languages the reader is referred to [Ebert87] and [Hudak89]. Some readers may miss the concepts of polymorphism, strong static typing, data abstraction, and garbage collection. These are not discussed since they can be regarded as orthogonal to language paradigms. Equally, all these concepts have been adopted by both functional and object-oriented languages. Type inference is not paradigm specific either but has not been successfully applied to object-oriented languages yet and therefore will be a stimulus for the extension of object-oriented languages (see part III starting at page 233).

Subsequently, I will use the syntax of the functional language HASKELL to depict functional concepts. The notation is very intuitive, but you may want to refer to an introduction like [Hudak & Fasel92].

1.2.1 Functional decomposition

I already mentioned the correspondence between functional decomposition and structured analysis and design. A program is viewed as a main function, that is decomposed into more functions with less work to accomplish (see figure 1.1).

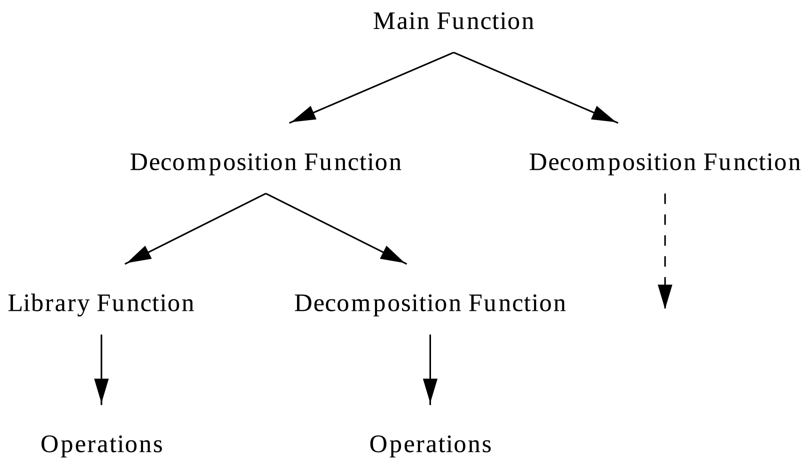


Figure 1.1: Functional decomposition

By functional decomposition I also include the separation between functions and data, i.e., the *tools and materials* metaphor. New functionality typically does not require to change existing data structures. It is simply added with external functions operating on top of existing data structures. However, changes to data structures often require to change existing functions. As functions depend on data, they must be adapted to properly react to, e.g., new constructors. See [Cook90] for a technical and Chapter 12 of [Meyer88] for a software engineering discussion of functional versus object-oriented decomposition.

1.2.2 Reduction Semantics

Computations within a functional programming language are performed just like reductions in a calculus. Instead of altering an implicit state with instructions, computation is defined as the reduction of expressions. Intuitively, we can think of a reduction step as a replacement of a term with a simpler term[1]. For instance:

$$1+2+4 \Longrightarrow 3+4 \Longrightarrow 7$$

Reduction does not always mean simplification in a naïve sense, i.e., expressions may as well grow in size in intermediate steps:

$$(1+2)^2 \Longrightarrow (1+2)*(1+2) \Longrightarrow 3*3 \Longrightarrow 9 \tag{1.1}$$

Yet, reduction is guaranteed to produce a unique so-called normal form, if it terminates at all[2]. An expression in normal form cannot be reduced any further and is what we intuitively think of as the value of an expression. Uniqueness of normal forms for the untyped λ-calculus is guaranteed by the Church-Rosser Theorem I [Hudak89].

In addition, the normal form may be reached through every possible reduction order, e.g.,

$$(1+2)^2 \Longrightarrow (3)^2 \Longrightarrow (3)*(3) \Longrightarrow 9 \tag{1.2}$$

yields the same result as the above calculation. We have to be more precise if errors (e.g., division by zero) or non-termination may also occur. For instance,

$$\text{False} \wedge (1/0 = 1) \Longrightarrow \text{False}$$

but only if logical conjunction ($\wedge$) is non-strict, that is, does not evaluate its second argument unless necessary. Here, the reduction order used in reduction 1.2 called applicative-order reduction[3] would not produce the normal form "False", but produce a division by zero error.

Fortunately, normal-order reduction[4], the strategy used in reduction 1.1, always yields the normal form, if it exists. This is stated by the Church-Rosser Theorem II for the untyped λ-calculus [Hudak89].

Reduction results, as well as any other expression, are immutable values. Adding an element to a list produces a new list copy with that element appended. Furthermore, the result is not distinguishable from a second list with identical element order that was obtained in a different way (e.g., by removing an element), that is, values have no identity. This is a remarkable property. To see why, consider the following (non-HASKELL) code:

[1]Though this view is a crude oversimplification it suffices for our purposes here.

[2]Termination is not guaranteed in general. However, there are special systems like the simply typed λ-calculus, the polymorphic typed λ-calculus, system F, and languages like FP that even guarantee termination.

[3]Evaluate arguments first; then apply function (just like call-by-value parameter passing).

[4]Evaluate function first; then arguments (akin to call-by-name parameter passing).

```
x := 1+6
y := 2+5
```

Here are some questions (adapted from [Appel93]):

- Is `x` the same 7 as `y`?
- If we modify `x` does `y` change?
- Do we need a copy of 7 to implement `z := x`?
- When we no longer need `x` how do we dispose of the 7?

You rightfully consider these questions as silly but what about posing them again in a different context?

```
x := aList.add(anItem)
y := aList.add(anItem)
```

We can conclude that numbers have value semantics[5] and many "silly" questions arise when reference semantics, i.e., references, mutable objects, and identity come into play.

For our purposes we may recall reduction semantics to mean:

1. No implicit state → No side-effects → Great freedom in reduction order.
2. No mutable objects nor identity → Aliasing is not an issue.

1.2.3 Higher-Order Functions

Functions are first-class values often called first-class citizens. A function can be an argument to another function (downward-funarg) as well as be the result of another function (upward-funarg). An example for passing a function as an argument is the *map* function:

$$\begin{aligned} map\ f\ [\,] &= [\,] & (1.3)\\ map\ f\ (x:xs) &= (f\ x) : map\ f\ xs & (1.4) \end{aligned}$$

It applies a function to all elements of a list. Hence,

$$map\ (+\,1)\ [1,2,3] \implies [2,3,4]$$

and

$$[\text{"}House\text{"}, \text{"}Boat\text{"}] \overset{map\ (\text{"}my\text{"}\ +\!\!+)^{6}}{\Longrightarrow} [\text{"}myHouse\text{"}, \text{"}myBoat\text{"}].$$

[5]Even in imperative languages!

Thus, downward-funargs are a perfect means to achieve behavior parameterization.

Let us look at a very basic function for an example of returning a function, namely function composition:

$$\textit{compose}\ f\ g \quad = \quad \lambda x \rightarrow f\ (g\ x)^7.$$

From two functions f and g a new function is produced that applies g to its argument and then applies f to the result of the former application. So,

$$7 \quad \overset{\textit{compose}\ (+21)\ (*3)}{\Longrightarrow} \quad 42,$$

since the result of 7 multiplied by 3 added to 21 yields 42. The *compose* function is an example for a functional, i.e., higher-order function, since it takes functions as arguments and produces a new function. Returning functions makes it easy to define new functions by a combination of existing functions. An interesting variation of creating a new function by supplying less parameters than needed is called partial application and uses a curried[8] function: Given

$$\textit{add}\ x\ y \quad = \quad x + y$$

several functions can be defined by supplying only one of two parameters:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\textit{inc} & = & \textit{add}\ 1 & \textit{inc}\ 98 & \Longrightarrow & 99 \\
\textit{dec} & = & \textit{add}\ (-1) & \textit{dec}\ 3 & \Longrightarrow & 2 \\
\textit{addTen} & = & \textit{add}\ 10 & \textit{addTen}\ 1 & \Longrightarrow & 11.
\end{array}$$

In fact, currying is possible with runtime values. We may create a function *addX* by supplying *add* with a number from user input. The function *inc* actually represents a closure. It represents function *add* but carries the value of x with it. Another typical way to create a closure is by capturing the value of a free (or global) variable as in

$$\begin{array}{l}
\textit{let}\ y = 13\ \textit{in} \\
\quad \lambda x \rightarrow \textit{add}\ x\ y.
\end{array}$$

Here we created a function that will add its argument (x) to 13. Variable y is said to be a free variable in the scope of the λ abstraction, as its value is determined by the function's environment rather than by a function parameter (Functions with no free variables are also called combinators). Closures are truly functions created at

[6] $(+1)$ and ("*my*" ++) (append) are called *sections* and denote the partial application of "+" to 1 and append to "*my*" respectively.

[7] λ builds a function abstraction and binds a variable (in this case x) to the value that will be received by a later function application.

[8] Named after the logician Haskell B. Curry, who used the notation $f\,x\,y$ to denote $(f\,x)\,y$, previously written as $f(x,y)$.

runtime which is also documented by the fact that they cannot be optimized by partial evaluation [Davies & Pfenning96].

The essence of the above is that higher-order functions provide more possibilities than functions normally do in imperative languages. In C, for instance, one can pass and return function pointers, but without a means to capture environment values. PASCAL allows closures as downward-funargs but (for reasons of language implementation) not as upward-funargs. With true first-class functions, however, the modularity of programs can be enhanced significantly [Hughes87].

1.2.4 Lazy evaluation

Most programming languages have strict semantics. We already got to know this type of evaluation strategy in section 1.2.2 on page 11 as applicative-order reduction and it is also known as call-by-value parameter passing. Yet another name for it is eager evaluation. Here is why: Given

$$\mathit{select}\, x\, y \;=\; x$$

we may obtain a reduction sequence like

$$\mathit{select}\; 1\; (2+3) \implies \mathit{select}\; 1\; 5 \implies 1.$$

The intermediate result 5 was computed although it was subject to be thrown away immediately. Even worse, eagerness may not only cause unnecessary work to be done it can also prevent the computation of meaningful results. Just imagine a non-terminating expression or $(1/0)$ in place of $(2+3)$. The result would be non-termination or a division-by-zero error, though the result 1 would be perfectly reasonable.

The solution is to use normal-order reduction, akin to call-by-name. With a non-strict *select* function the reduction sequence becomes

$$\mathit{select}\; 1\; (2+3) \implies 1,$$

which is both faster and safer. Indeed, almost any programming language contains at least one non-strict operator/statement in order to allow recursion or choice of side-effects. Typically, it is an if-statement that does not evaluate its branching arguments until one can be chosen.

Nevertheless, we already have seen an example when lazy evaluation causes more work than eager-evaluation. If you compare reduction sequence 1.2 on page 11 using eager evaluation to reduction sequence 1.1 you will note that it needs one addition operation less. The problem was created by duplicating an expression (by unfolding exponentiation to multiplication) which also duplicated the computation amount. Alas, all we need to do is to record the duplication and to share computation results. Speaking in calculi terms we replace string reduction by graph reduction. Then, reduction sequence 1.1, becomes

$$\begin{tikzcd}
(1+2)^2 \arrow[r, Rightarrow] & (\bullet \ast \bullet) \arrow[d] \arrow[r, Rightarrow] & (\bullet \ast \bullet) \arrow[d] \arrow[r, Rightarrow] & 9 \\
& \overbrace{(1+2)} & \overbrace{3} &
\end{tikzcd}$$

causing $(1+2)$ to be evaluated only once. The combination of normal-order reduction and sharing of expression results is known as call-by-need.

With call-by-need we may not only save some computations we may even reduce the computational complexity of algorithms. Let us obtain the minimum of a list by first sorting the list and then retrieving the first element:

$\mathit{min}\ xs = \mathit{hd}\ \circ$[9]$\mathit{sort}\ xs$

When we implement sorting with *insertion-sort* [Bird & Wadler88] the complexity of algorithm *min* with eager evaluation is $O(n^2)$, n being the number of elements in the input list. With lazy evaluation the complexity of *min* is reduced to $O(n)$, since *insertion-sort* finds the first element of the sorted list in $O(n)$ time and sorting of the rest of the list does not take place, as it is thrown away by the application of *hd*. In effect, what is thrown away is a function closure that could produce the rest of the list — item by item — if demanded. We can directly transfer the above observation for the implementation of Kruskal's algorithm for minimal spanning trees. This algorithm uses a sorted list of edges to construct a minimal spanning tree of a graph [Aho & Ullmann92]. If the number of edges in the original graph is large compared to the number of edges needed to constitute the resulting spanning tree it pays off to use lazy sorting (e.g., lazy quicksort) that just sorts as much as is requested by the algorithm [Okasaki95a].

Besides efficiency improvements, lazy evaluation opens up the world of infinite data structures. A definition like

$$\mathit{ones} \quad = \quad 1 : \mathit{ones}$$

is not possible with a strict version of cons (:)[10]. There would be no reasonable use of *ones*, in that every function accessing it would not terminate. With a lazy cons, however, the above definition provides an infinite number of ones. Only as much ones as needed are produced, e.g.,

$$\mathit{take}\ 5\ \mathit{ones} \Longrightarrow [1,1,1,1,1].$$

We can even perform infinite transformations as in

$$\begin{array}{rcl} \mathit{twos} & = & \mathit{map}\ (+1)\ \mathit{ones} \\ & (\Longrightarrow & [2,2,2,\ldots]). \end{array}$$

A more useful definition is the list of *all* prime numbers:

$$\mathit{primes} \quad = \quad \mathit{sieve}\ [2,\ldots] \tag{1.5}$$

$$\mathit{sieve}\ (p : xs) \quad = \quad p : \mathit{sieve}\ (\mathit{filter}\ ((\neq 0) \circ (\mathit{mod}\ p))\ xs) \tag{1.6}$$

[9]Infix notation for the *compose* function of section 1.2.3 on page 12.

[10]This operator produces a list by appending the second argument to the first.

The algorithm uses the famous Sieve of Erathostenes[11] and filters all non-prime numbers by testing an infinite amount of generated numbers ($[2,\ldots]$) against all already found prime numbers. Most importantly, it relies on an infinite supply of numbers to test against and produces an inexhaustible list of prime numbers. There is no need to speculate about the highest number a client could possibly request for.

Summarizing, we keep in mind that lazy evaluation avoids unnecessary non-termination or errors by creating a network of data dependencies and evaluating just that and nothing more. A considerable amount of computation can be spared this way to the amount of reducing the complexity of algorithms. Computation complexity, thus, may not depend on an algorithm's nature, but on its usage, i.e., which data in what order is requested. Finally, lazy constructors allow for infinite data structures [Friedman & Wise76]. Lazy functions imply lazy data but there are languages like HOPE that provide lazy data only in order to avoid non-strict semantics for functions [Burstall et al.80]. The reason for this is to avoid closure creation for suspended function applications and to retain an applicative order reduction, e.g., for side-effects or easier debbuging.

Anyway, lazy evaluation, similar to higher-order functions, can significantly enhance the modularity of programs [Hughes87], as they free data generators from knowing about the specifics of consumers. Termination control can be shifted to the consumer as opposed to a conventional mix of generation and control.

1.2.5 Pattern Matching

Pattern matching is a typical example for syntactic sugar. Instead of testing for argument values within the function body, e.g.,

$$\begin{aligned} fac\; n &= if\; n == 0\; then\; 1 \\ &\quad else\; n * fac\; (n-1) \end{aligned} \qquad \text{or} \qquad \begin{aligned} fac\; n &= case\; n\; of \\ &\quad n == 0 \rightarrow 1 \\ &\quad n > 0 \rightarrow n * fac\; (n-1) \end{aligned}$$

we may write:

$$\begin{aligned} fac\; 0 &= 1 \\ fac\; (n+1) &= (n+1) * fac\; n. \end{aligned}$$

Along with the idea of using equations as part of the syntax, pattern matching adds much to the declarative nature of functional programs. All case discriminations are clearly separated from each other and the implicit assembly of partial function definitions is very appealing. Note, however, besides a possibly psychological factor there is no real software engineering significance associated with pattern matching. The expressive power of a language [Felleisen91] is not augmented with pattern matching in any way.

[11]Eratosthenes was a Alexandrian Greek philosopher in the third century B.C.

1.2.6 Type inference

Among the many innovations of ML [Milner et al.90, Wikström87] was a type system that liberated a programmer from explicitly declaring the types of expressions. For instance, the type system would automatically infer the type of function *map* (see definition 1.3 on page 12) to be

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \tag{1.7}$$

that is, a function from type a to b and a list of elements of type a is used to produce a list with elements of type b. This is an example of parametric polymorphism, that is, a single function definition works in the same way for many types.

A strongly and statically typed language with type inference ensures the absence of any runtime type errors without putting any declaration burden on the programmer. For each expression the most general (so-called principle) type is computed. With explicit type declarations a programmer might over-specify types and thereby unintentionally narrow the use of a function. For instance, someone programming functions on integer lists may type *map* as

$$map :: (Int \rightarrow Int) \rightarrow [Int] \rightarrow [Int],$$

thus, excluding many other possible applications of map.

There is a different view on this matter that argues that type declarations are not just a service to compilers but constitute a part of the programs documentation and even constitutes an albeit weak specification contribution. The nice thing about type inference is that it allows a mixed style, i.e., explicit type declarations overrule computed types and affect the type inference of associated expressions. In effect, in the presence of type declarations the type system performs type checking, comparing the inferred with the declared type.

Like pattern matching there is no real impact of type inference in comparison with type checking on software designs. Notwithstanding, the impact of the readability of programs should not be underestimated. An example how type declarations can get in the way of an otherwise clean syntax is LEDA [Budd95].

1.3 Review

The following two sections give a subjective assessment of functional programming from a software engineering point of view.

1.3.1 Pro

Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover.
– C.A.R. Hoare

We already highlighted functional concepts (e.g., higher-order functions and lazy evaluation) and their ability to improve software properties (e.g., modularity of programs) in the preceding sections. Now, we turn to the foundations of functional programming and its positive implications.

1.3.1.1 Programming Discipline

We can think of functions as a disciplined use of gotos [Dijkstra76] and parameter binding as a disciplined use of assignment. In this light, the absence of imperative control structures and side-effects is not regarded as inhibiting expressiveness but as a discipline for good programming [Hudak89].

Referential Transparency Representing computations in terms of expressions promotes the specification of problems and establishes data dependencies rather than giving an over-specified algorithm and execution order. That is why functional languages are considered to be declarative. Maintaining programs is facilitated by the fact that equals may be replaced by equals, i.e., equivalent expressions can be interchanged without regard for introducing interferences through side-effects.

Furthermore, the absence of side-effects makes programs much more amenable to parallelization. With regard to the multi-processor and massive parallel architecture designs trend in hardware there is a great demand for languages with a high potential for parallel execution.

Finally, correctness proofs are much easier in a language with reduction semantics. Imperative languages require an additional calculus such as Dijkstra's predicate calculus or Hoare's weakest preconditions. Program transformations can be proven to be correct almost as easily as manipulating mathematical formulas.

1.3.1.2 Concise programs

Functional programs are typically shorter than their imperative counterparts; at least given an appropriate problem domain which is amenable to a mathematical description [Harrison et al.94].

Short Syntax The most important operation in functional programming — function application — is denoted as juxtaposition, that is, it does not need any syntactical elements! This economy in syntax pays back in short programs. Also, the frequent use of manifest constants reduces the amount for preliminary initializations. In

$$\mathit{take}\ 5\ (\mathit{filter}\ \mathit{even}\ [1..]) \tag{1.8}$$

we need five characters to denote an infinite list of integers. The creation of a new function by composing two existing functions (*filter* and *even*) solely requires a space. There is even no need for bracketing, since function application associates to the left. For the very same reason, we actually need to bracket the list argument for *take*, i.e., this time we need a space and parentheses in order to express the sequencing of functions[12].

The fact that the above short program involves the concepts of

- infinite data structures,
- lazy evaluation (by list construction, *filter* and *take*),
- higher-order functions (passing the predicate *even*), and
- type-safe instantiation of polymorphic functions (especially specialization of *take* and *filter* to integers)

clearly witnesses the expressiveness of functional programming syntax.

Another useful syntactical shortcut, inspired by mathematical notations to define the contents of sets, are list comprehensions:

$$[x * y \mid x \leftarrow [6,9..], y \leftarrow [1..9],\ odd\ y]$$

denotes a list of all products between two lists ranging from one to nine (but odd numbers only) and from six to whatever number is reached by picking numbers in steps of three. Note that due to the nesting semantics of ‘,’ in list comprehensions the first list produces unique numbers only while the second recycles again and again.

Compositionality As there is just one world of expressions in functional programming rather than two worlds of expressions and statements like in imperative programming [Backus78] the composition of program parts is especially easy. The result of calling *filter* in program 1.8 on the facing page is simply passed on to *take* without a need for a communication device. Compare the program fragments in figure 1.2. The functional version much more clearly expresses the intent of the

```
l1.add('a')
l2.add('b')
l1.concat(l2);        versus        ('a' : l1) ++ ('b' : l2)
Result:=l1;
```

Figure 1.2: Comparison between imperative and functional style

program, does not need to pass the result via `l1`, and leaves `l1` and `l2` intact for other uses.

During the examination of program 1.8 on the facing page we have already seen that not only values but also functions may be “glued” together[13]. A nice

[12]Another way for writing the same function is $(take\ 5) \circ (filter\ even)\ [1..]$.

[13]By the simple fact that functions are values too, i.e., first-class expressions.

demonstration of composing programs (parsers) with higher-order functions is given in [Hutton92].

Yet another type of gluing is enabled by the use of lists as a paradigmatic data structure. Many types of data structures in functional programming are modeled with lists, e.g., sequences, sets, flattened trees, etc[14]. An instructive example is the modeling of a chessboard by a list of numbers in the eight queens problem: Find a configuration of eight queens on a chessboard such that no queen threatens another [Bird & Wadler88]. The full program incrementally produces solutions with increasing board sizes [Wadler85], but we only look at the function that checks whether a new queen can safely be put on column *n* given a board configuration *p* (list of taken columns).

$$\begin{aligned} \mathit{safe}\ p\ n &= \mathit{and}\ [\mathit{not}\ (\mathit{check}\ (i,j)(m,n)) \mid (i,j) \leftarrow \mathit{zip}\ ([1..\#p],p)] \\ &\ \mathit{where}\ m = \#p+1 \end{aligned} \tag{1.9}$$

For the reason that only column positions are stored, whereas rows are implicitly encoded in list positions, *safe* has to construct one position for the new queen (column *n* & row *m*) and reconstruct one position tuple for all old queens respectively. The latter part is accomplished by (*zip*) that joins the column positions with a generated list of row positions. All thus reconstructed position tuples are checked against the new position, yielding a list of safeness indicators. This list of booleans is then reduced to a single boolean result by *and*. The use of *zip* and *and* was possible only by using a list to model column positions and using a list of safeness indicators.

Many standard functions take and produce lists and, therefore, can be often reused for purposes as above. The list effectively serves as a *lingua franca* between functions.

As a final example for the elegance of functional programming have a look at the definition of the famous quicksort algorithm.

$$\begin{aligned} \mathit{quicksort}\ [\,] &= [\,] \\ \mathit{quicksort}\ (x:xs) &= \mathit{quicksort}\ (\mathit{filter}\ (<x)\ xs) \\ &\ +\!\!+\ [x]\ +\!\!+ \\ &\ \mathit{quicksort}\ (\mathit{filter}\ (>=x)\ xs) \end{aligned}$$

The idea of the algorithm (to conquer by dividing the problem into the concatenation of two lists and a pivot element) almost springs to the eye of the reader. A typical imperative solution (using nested while-statements to pre-sort the list in-place) almost gives no clue about the design idea.

Assuming that concise programs are faster to write the compactness of functional programs translates to increased programmer productivity [Ebert87].

Despite their shortness functional programs are often said to execute inefficiently. However, experiments with a heavily optimizing compiler for the strict

[14]Lists are used as the single and universal data structure in John Backus' Fp language [Backus78]. Many functions are just there to navigate values to their right position in lists.

functional language SISAL show that functional programs can be faster than FORTRAN [Cann92] and programs that require complicated and expensive storage management in C may run faster in a ML implementation with a good garbage collector [Clinger & Hansen92].

1.3.2 Contra

Purely applicative languages are poorly applicable.
– Alan J. Perlis

The more light there is, the more shadow one might expect. Indeed, a part of the same aspects that we mentioned in favor of functional programming can be turned into counterarguments from a different perspective.

1.3.2.1 Functional decomposition

Functional decomposition works fine for the tools and materials metaphor when programming in the small. Also, a decomposition based on actions and transformations can often be more intuitive than some entity-relationship organization [Moynihan94]. On the other hand, it is difficult to assign a "main function" to any large system and any choice is likely to be invalidated some time. A functional decomposition is inherently instable, since the topmost functions are not grounded in the problem domain and are subject to change whenever new requirements occur. Additionally, often systems are expected to cope with new data which demands to accommodate all affected functions. Precisely these reasons led to the redemption of structured analysis and design by object-oriented methodologies [Meyer88].

1.3.2.2 Reduction semantics

The very same feature — renunciation of state — that gives functional programming its strength (absence of side effects) is responsible for its main weakness (lack of updates).

"How can someone program in a language that does not have a notion of state? The answer, of course, is that we cannot... [Hudak89]." – Paul Hudak

Of course, Hudak goes on explaining that functional languages treat state *explicitly* rather than *implicitly*. Passing around state for clarity and modeling references by indirection is fine but some serious problems remain.

Essential State

Felder sind physikalische Zustände des Raumes.
– Albert Einstein

We perceive the real world as consisting of objects which change over time. Real objects have identity and state. There is a great impedance mismatch between reality and software solution if we have to, say, deal with copies of a pressure tank every time pressure or temperature changes. Modeling a bank account with fair access to multiple users is a problem in functional languages [Abelson & Sussman87]. In fact, the problems to model state with a feedback model [Abelson & Sussman87] (see also figure 1.4 on page 28) and the prospect to incorporate functional programming into an object-oriented approach with a functional design pattern system caused the MUSE project to cancel the use of the functional programming language SAMPλE [Henhapl et al.91, Deegener et al.94, Jäger et al.88, Kühnapfel93, Kühnapfel & Große94].

Something fundamental and innocent looking as simple I/O has been a hard problem for functional languages. Many models, such lazy streams, continuation semantics, and systems model [Hudak & Sundaresh88, Gordon93b, Gordon93a] have been proposed to overcome the discrepancy between a referential transparent language and I/O side effects. Hudak gives examples of the above I/O styles [Hudak89] and although imperative syntax is mimicked they look very un-intuitive and cumbersome to someone used to plain I/O statements. Recently, a step forward has been made by employing so-called monads for I/O [Jones & Wadler93, Carlsson & Hallgren93]. This model has also been adopted for the latest revision for HASKELL. It remains a matter of taste whether such a treatment of I/O state is conceived as natural or unacceptable.

A related area, also suffering from lack of updates, are cyclic data structures (e.g., cyclic graphs). While these are easily and elegantly expressible in functional programming updating them is very inefficient, since there is no way around rebuilding the whole cycle[15]. One circumvention is to emulate pointers with mutable arrays (e.g., in HASKELL) [van Yzendoor95], but then we start to experience a mismatch between paradigm (imperative) and language (functional) used.

Especially reactive systems, furthermore, demand a notion of event. An image recognizing robot must stop improving its assumptions when some movement is absolutely necessary. Functional languages cannot express such time-dependent planning, since an event interrupt implies execution order, which has no meaning in a functional program [Meunier95a].

Howbeit the tasteful use of a `goto` may improve a program [Knuth74] it is not difficult to do without `goto` altogether. To renounce implicit state, however, appears much harder. Although, monads may emulate backtracking, nondeterminism, exceptions, and state, in sum —

> *“. . . it is true that programming with updates is a proven technology, and programming* entirely *without them is still ‘research’ [Appel93]*[16]*.”*
>
> – Andrew W. Appel

Monads especially do not seem to allow decoupling and structuring state in a system.

[15]Unless the whole data structure is single-threaded which can be difficult to prove.

[16]Appel already knew about Monads and cites [Wadler92].

Desirable State Arrays are often used due to efficiency considerations. Yet, functional programming languages must conceptually copy whole arrays for single updates. Only if the compiler can detect a single threaded array use it can use efficient destructive updates [Wadler90, Odersky91, Jones & Wadler93]. Monadic arrays are guaranteed to be single threaded, though.

It is very convenient to side-effect a result cache, e.g., to reduce the runtime complexity of

$$\begin{aligned} \mathit{fib}\,1 &= 0 \\ \mathit{fib}\,2 &= 1 \\ \mathit{fib}\,n &= \mathit{fib}\,(n-1) + \mathit{fib}\,(n-2) \end{aligned}$$

from $O(e^n)$ down to $O(n)$. There are purely applicative version of memoization [Keller & Sleep86], but to overcome limitations like expensive quality checks on lists more advanced approaches are necessary. Hughes' so-called *lazy memo-functions* are not syntactic sugar anymore, since an identity predicate — normally not provided by a functional language — is required [Hughes85].

> *"The interesting thing about memoization in general is that it begins to touch on some of the limitations of functional languages — in particular, the inability to side effect global objects such as caches — and solutions such as lazy memo-functions represent useful compromises. It remains to be seen whether more general solutions can be found that eliminate the need for these special-purpose features [Hudak89]."*
>
> – Paul Hudak

There are more function definitions like the fibonacci function definition above that look beautiful but are hopelessly inefficient. Often the culprit is the $+\!\!+$ operator which takes time linear to the size of its left argument. So, instead of

$$\begin{aligned} \mathit{reverse}\,[\,] &= [\,] \\ \mathit{reverse}\,(x : xs) &= (\mathit{reverse}\;xs) +\!\!+ [x] \end{aligned}$$

one should apply the accumulator technique [Burstall & Darlington77, Bird & Wadler88] and use

$$\begin{aligned} \mathit{reverse} &= \mathit{rev}\,[\,] \\ \mathit{rev}\;\mathit{acc}\,[\,] &= \mathit{acc} \\ \mathit{rev}\;\mathit{acc}\,(x : xs) &= \mathit{rev}\,(x : \mathit{acc})\;xs. \end{aligned}$$

This version of reverse is $O(n)$ instead of $O(n^2)$ but undoubtedly lost clarity as well. For the very same reason the nice formulation of displaying a tree

$$\begin{aligned} \mathit{showTree}\,(\mathit{Leaf}\;x) &= \mathit{show}\;x \\ \mathit{showTree}\,(\mathit{Branch}\;l\,r) &= {}''<'' +\!\!+ \mathit{showTree}\;l +\!\!+ {}''|'' +\!\!+ \mathit{showTree}\;r +\!\!+ {}''>'' \end{aligned}$$

should be replaced by a less nice version using *shows* [Hudak & Fasel92].

Transformations aimed at efficiency like the accumulator technique or tail-recursive functions often destroy the clarity and beauty of initial solutions[17]: One cannot really speak of declarative programming, concise programs, and mathematical specifications in view of the tweaked definition of a function to produce permutations:

$$\begin{aligned}
perms\ [\,]\ tail\ res &= tail \mathbin{+\!\!+} res \\
perms\ (x : xr)\ tail\ res &= cycle\ [\,]\ x\ xr\ tail\ res \\
\\
cycle\ left\ mid\ [\,]\ tail\ res &= perms\ left\ (mid : tail)\ res \\
cycle\ left\ mid\ right@(r : rr)\ tail\ res &= cycle\ (mid : left)\ r\ rr\ tail \\
&\ (perms\ (left \mathbin{+\!\!+} right)\ mid : tail\ res\ ) \\
\\
fastperms\ xs &= perms\ xs\ [\,]\ [\,]
\end{aligned}$$

"Functional languages etc. do of course abstract away from more of the machine-level details than do imperative ones, but real programs in any language contain a great deal of 'how'. The nearest thing I know to a declarative language in this sense is Prolog, as described in lecture 1. Unfortunately, lectures 2... n follow :-) [Sargeant96a]."

– John Sargeant

So, unfortunately a lot of the over-specification of "how" to do things that could be left out in nice formulations must be given later in case more efficient version are needed. Note that the definition of *quicksort* (see definition 1.10 on page 20) compares each element twice for creating two new input lists. A more efficient version using an auxiliary *split* function already appears less clear. Beyond this, the combination of result lists with append ($\mathbin{+\!\!+}$) is very inefficient again in time (adding an $O(n)$ factor) and cannot compete with the in-place sort of the imperative version (zero space overhead).

Beyond aesthetical considerations there appears to be a class of algorithms that are inherently imperative causing one to end up asymptotically worse without imperative features [Ponder88, Hunt & Szymanski77].

Just to achieve $O(1)$ complexity for insertion and deletion of elements in list — a triviality in imperative programming — some tricky machinery has to be employed in functional programming [Okasaki95c].

Also, some algorithms have a very intuitive imperative form. In John Horton Conway's *Game of Life* cells need to count the number of inhabited neighbor cells. One approach is to iterate all cells and determine their neighbor count by testing all adjacent cells (see functional version on the left of figure 1.3 on the next page, needing eight tests).

[17]Evil imperative programmers might remark that they do not care whether their programs are amenable to transformations, since they get acceptable efficency at once and functional programs need to be transformable because of the poor performance of initial versions.

Alternatively, one may list inhabited cells only and increase (by a side-effect) the neighbor count of all adjacent cells (see imperative version on the right of figure 1.3, needing just three[18] updates).

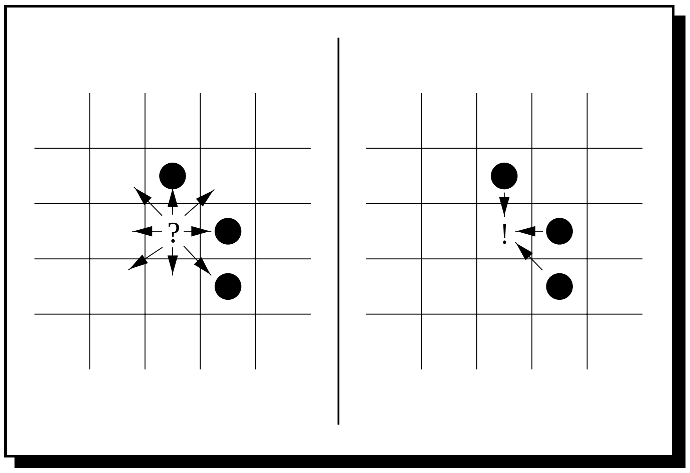


Figure 1.3: Functional and imperative LIFE

Unavoidable State Again, it is not possible to circumvent state. Ordering of actions in imperative solutions translates to ordering of data dependencies or nesting in functional solutions. Consider Joseph Weizenbaum's Eliza program. One part of it needs to replace words in order to use User input for Eliza's questions. Given the input list *inps* and the replacement list *reps* —

$$\begin{aligned} inps &= [\text{"i"},\text{"am"},\text{"i"},\text{"says"},\text{"Eliza"}] \\ reps &= [(\text{"you"},\text{"i"}),(\text{"i"},\text{"you"}),(\text{"am"},\text{"are"})] \end{aligned}$$

— you are invited to convince yourself which of the following two expressions ($substitute_1$ or $substitute_2$) does the correct substitutions:

$$\begin{aligned} replace\ org\ (from,to)\ xs &= if\ org == from\ then\ to\ else\ xs \\ replace'\ (from,to)\ org &= if\ org == from\ then\ to\ else\ org \\ substitute_1 &= map\ (\lambda w \rightarrow foldr\ (replace\ w)\ w\ reps)\ inps \\ substitute_2 &= foldr\ (\lambda p \rightarrow map\ (replace'\ p))\ inps\ reps. \end{aligned}$$

The above problem is akin to the considerations to be made when you have to determine the order of list generators in a list comprehension. Unless the correct nesting is specified the result will not be as desired.

While one of the above alternatives is wrong there is another example of two correct but nevertheless different solutions: To calculate all possible chessboard configurations of eight queens that do not threaten each other (see the corresponding '*safe*' function in definition 1.9 on page 20) both programs *queens* and *sneeuq*, with the relation

$$queens\ m = map\ reverse\ (sneeuq\ m),$$

need the same time. Notwithstanding, *sneeuq* is ten times slower in producing just one solution [Bird & Wadler88]. This effect is due to the order relevance of steps

[18]In general, the number of neighbors, which is usually low.

that produce potential solutions to be tested. An unfavorable order causes much more invalid configurations to be tested[19].

Another example how ordering corresponds to nesting are monads used in conjunction with folds [Meijer & Jeuring95]. The functions *foldr* and *foldl* replace list constructors with functions. When an associative function (together with an identity element) is used the choice of *foldl* and *foldr* does not affect the result[20] but is made according to runtime characteristics. However, when the function is, say, a state monad the result will be affected, akin to an imperative left or right traversal of a sequence with side-effects.

Finally, if one needs to model references, e.g., to represent sharing of nodes in a graph, of course aliasing may occur just as in imperative languages. In this case, the danger is much more concentrated and explicit in a functional program, but nevertheless, unavoidable.

In conclusion, treating state explicitly does not make reasoning about state easier. It only may make reasoning about programs easier, which — though of course desirable in itself — is not the same as the former.

1.3.2.3 Deceptive Clearness

Functional programs often look very nice but a deeper inspection frequently reveals unexpected behavior.

Inefficient lists Given the often weak performance of algorithms using lists it appears unfortunate that much reuse is gained by representing data structures with lists and using standard list functions for processing and interconnection (see paragraph Compositionality in section 1.3.1 on page 17). Moreover, often abstract datatypes are more appropriate for encapsulating data and providing natural operations. For instance, a representation of a list of columns for a chessboard configuration (see definition 1.9 on page 20) is fine but should be encapsulated by an abstract data type.

Machine lurks below Albeit transformations such as $(a+b)+c \Longrightarrow a+(b+c)$ are possible and valid one should not be surprised if floating point results are affected in case machine arithmetic rounding effects come into place. Also, the illusion of a mathematical world is destroyed when recursion does not terminate and, thus, causes the control stack to overflow. In fact, the problems aligned with uncontrolled user defined recursion (unproved termination, necessity for inductive proofs, unknown time and space complexity, hindrance of automatic optimization) have been a motivation for the so-called Squiggol school to allow a fixed set of recursive com-

[19]The author once wrote a backtracking program to generate a solution for the well-known solitaire game. A different ordering of north, east, south, and west moves caused a one megahertz 6502 processor to conclude in 30 seconds while an eight megahertz 68000 processor calculated for 2 days.

[20]First duality theorem of folds [Bird & Wadler88].

binators only [Bird & de Moor92, Meijer et al.91]. This research direction towards a more algebraic notion of functional programming demonstrates that

1. reasoning about general recursion is not easy and
2. there is a need to optimize the time and space efficiency of functional programs.

Pattern Matching Often, nice definitions as

$$\begin{array}{rcl} \mathit{pred}\;0 & = & 0 \\ \mathit{pred}\;(n+1) & = & n \end{array}$$

are shown but one should not be mislead to believe that the automatic inverse calculation of operands is possible in general. The $n+k$ pattern is just a special case and often one will need to decompose arguments with functions as opposed to patterns.

In lazy languages subtle effects may occur:

$$\begin{array}{rcl} f\;1\;1 & = & 1 \\ f\;2\;\_ & = & 2 \end{array}$$

behaves differently than

$$\begin{array}{rcl} g\;1\;1 & = & 1 \\ g\;\_\;2 & = & 2 \end{array}$$

because $f\;2\;\perp$[21] yields 2 but $g\;\perp\;2$ yields $\perp$ [Hudak89], i.e., contrary to intuition pattern matching depends on the ordering of arguments. The same applies to the ordering of patterns which is crucial for the correct definition of the function. By exchanging the first two patterns of *take*

$$\begin{array}{rcl} \mathit{take}\;0\;\_ & = & [\,] \\ \mathit{take}\;\_\;[\,] & = & [\,] \\ \mathit{take}\;(n+1)\;(x:\mathit{xs}) & = & x:\mathit{take}\;n\;\mathit{xs} \end{array}$$

one can tune a preference for possibly getting a result though the numeric argument (or alternatively the list argument) is not available (i.e., $\perp$). Other subtle effects of pattern and argument ordering caused by pattern matching are discussed in [Hudak89]. HASKELL uses a top-to-bottom left-to-right ordering of patterns and arguments. Note that this imposes restrictions for compilers. They are no longer free to evaluate arguments in any order.

[21]"Bottom" is used to denote "error" or non-termination.

Laziness meets pattern matching also in case of circular dependencies: HASKELL's I/O model was based on a feedback loop defined by

$$\begin{aligned} reqs &= client\ init\ resps \\ resps &= server\ reqs \end{aligned}$$

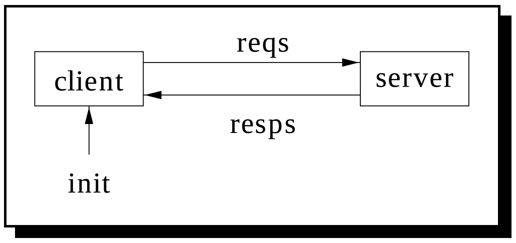


Figure 1.4: I/O feedback model

The associated function definitions are [Hudak & Fasel92]:

$$\begin{aligned} client\ init\ (resp : resps) &= init : client\ (next\ resp)\ resps \\ server\ (req : reqs) &= process\ req : server\ reqs. \end{aligned}$$

Before *client* has issued a request it already “pattern matches” for a response. Pattern matching causes evaluation and subsequently a deadlock. Note that the more conventional version without pattern matching

$$client\ init\ resps \quad = \quad init : client\ (next\ (head\ resps))\ (tail\ resps)$$

would work! For these and other cases HASKELL provides a lazy pattern matching operator.

Sometimes pattern matching works but causes unexpected inefficiencies: The definition of merge sort

$$\begin{aligned} merge\ [\,]\ ys &= [\,] \\ merge\ xs\ [\,] &= [\,] \\ merge\ (x : xs)\ (y : ys) &= x : merge\ xs\ (y : ys),\ x <= y \\ & \quad\ y : merge\ (x : xs)\ ys,\ x > y \end{aligned}$$

splits the heads from input lists only to reassemble them in two branches respectively [Buckley95]. HASKELL allows naming patterns, e.g., $merge\ xs'@(x : xs)\ ys'@(y : ys)$ in order to use xs' for a recursive call for *merge*, thereby adding another special option to pattern matching. This option introduces a further subtlety: Actually, the type (given in parentheses) of

$$\begin{aligned} data\ Sum\ a\ b &= L\ a \mid R\ b \\ copyr\ (R\ b) &= R\ b \quad (:: Sum\ a\ b \rightarrow Sum\ a\ c) \end{aligned}$$

is different to

$$copyr\ r@(R\ b) \quad = \quad r \quad (:: Sum\ a\ b \rightarrow Sum\ a\ b),$$

since the former definition amounts to a reconstruction [Okasaki95b].

An interesting collection of typical problems students encounter when learning functional programming is given in [Clack & Myers95].

2 Object-orientation

Computing is viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.
– Adele Goldberg

This chapter introduces object-orientation and defines some terms that are used in later discussions. I explain the underlying worldview (section 2.1) and the most important concepts (section 2.2 on page 31). Finally, section 2.3 on page 38 enumerates pros and cons of object-orientation.

In this dissertation I restrict myself to class based languages like C^{++}, Eiffel, and Smalltalk. There is another interesting class of so-called delegation based or prototyping languages like Self which use a dynamic type of inheritance and renounce classes. However, classless languages are not in widespread use and one of my aims is to improve the practice in object-oriented design.

2.1 Worldview

The object-oriented paradigm suggests to decompose systems in autonomous objects. Autonomous means an object

1. represents a complete entity. It is not just material or tools but both at once.

2. has a self-supporting state. It manages and keeps its state without needing support.

3. provides self-sustained operations. In case it refers to other objects it does not matter to clients.

In terms of real world modeling objects represent real world objects and capture their identity, state, and behavior. In terms of software decomposition each object can be regarded as a little computer inside the computer.

"For the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers... ? [Kay96]"
– Alan Kay

So, object-oriented systems have a less hierarchical but a more collaborative nature. Even when reducing complexity by decomposition it is not done at the expense of the power of the parts.

> *"The basic principle of recursive design is to make the parts have the same power as the whole."* – Bob Barton

In a purely object-oriented languages there are no free functions that are defined on objects. Every function must be part of a data abstraction. Hence, an object-oriented systems bears no resemblance to an Input / Processing / Output model but constitutes a network of interactive events and responses.

Objects can be thought of as abstract datatypes. This correspondence is fine with regard to real world modeling but from a software engineering perspective we have to take a closer look. Abstract datatypes define functions on constructors, i.e., provide data abstraction. Objects, distribute operations to constructors, i.e., provide procedural abstraction [Cook90]. An object operation implicitly knows the constructor it operates on, since it is tied to its definition. A function on an abstract data type has to distinguish between constructors first. That is why, it is hard to introduce new constructors to abstract datatypes (all functions must be extended) and easy to just add a new object. Conversely, it is hard to add a function to objects (all objects must be changed) and easy just to define a new function for all constructors.

Note, that most object-oriented languages allow using classes for both abstract datatypes (hide representation of implementation) and procedural abstraction (distribute constructors to subclasses and bind dynamically). In any case, objects are accessible only via the operations they provide. These are invoked by so-called message sends (or method calls).

Object-orientation not only decomposes complex systems into smaller parts corresponding to real world entities, but also captures these entities in a taxonomy of inheritance relationships. Inheritance may denote:

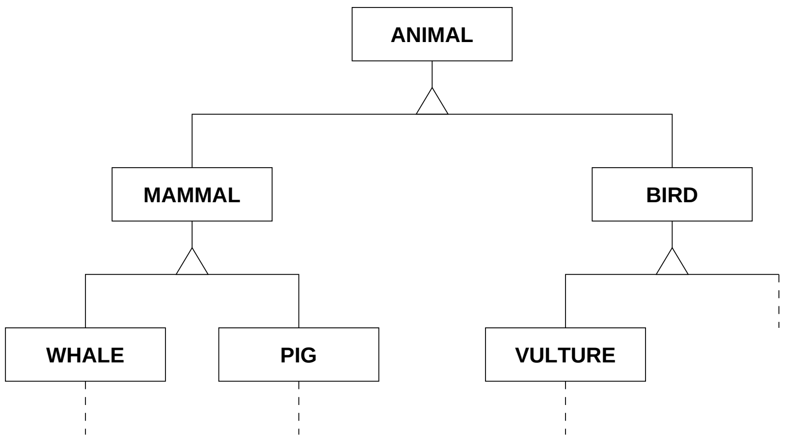


Figure 2.1: Object-oriented taxonomy

Is-a The heir is of the kind of its ancestor (also referred to as specialization inheritance). This relationship classifies entities, similar to the classification of animals in zoology[1] (e.g., **Whale** is a **Mammal**).

Subtype The heir can be used whenever the use of its ancestor is appropriate Heirs conform in interface and behavior and provide monotonic extensions only. This is also known as the *Liskov Substitution principle* [Liskov & Wing93] (e.g., a **Vulture** can be used whenever a **Bird** is expected).

Code reuse Heirs inherit code from their ancestors (also known as subclassing). Either through directly re-exporting inherited features or by their use for the definition of new features heirs may exploit ancestor code (e.g., **Pig** may reuse code templates of **Mammal**).

Ideally, these three types of inheritance coincide (see figure 2.2), but normally they are in conflict with each other. For instance, a **IntegerSet** is-a **Set** but it is not a subtype, since clients may want to insert strings. Furthermore, **Set** may inherit code from **HashArray** but neither is-a nor subtyping are accurate [LaLonde & Pugh91].

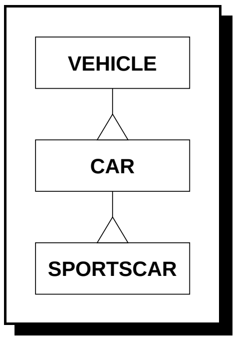


Figure 2.2: Ideal inheritance

The object-oriented worldview can be said to rule the software engineering world. Since its birth through SIMULA in 1967 it became the programming language and methodology paradigm [Rumbaugh et al.97] of choice. There was a 60% increase in SMALLTALK uses in 1993–1994 [Shan95] and today it is even used in embedded systems like oscilloscopes [Thomas95].

2.2 Concepts

The following sections are meant to briefly introduce concepts of object-oriented languages. For a more thorough discussion the reader is referred to [Wegner87, Wegner90, Beaudouin-Lafon94] for technical matters, to [Nelson91] for a clarification of terms, and to [Meyer88, Nierstrasz89, Jacobson et al.94, Booch94] for software engineering and system development relevance.

Subsequently, I will use the syntax of the object-oriented language EIFFEL in order to depict object-oriented concepts. The syntax is very clean and intuitive, but you may want to refer to a definition [Meyer92], introduction [Racko94], textbook [Rist & Terwilliger95, Thomas & Weedon95], or application development books [Walden & Nerson95, Jézéquel96]. EIFFEL's garbage collection avoids distractions from memory management and its static type system allows typing issues to be discussed. EIFFEL's clear syntax, simple semantics, and support of correctness through the use of assertions made it the premier choice for an education language for many universities throughout the world [Meyer93].

[1]With multiple inheritance one may even let whales inherit from mammals and fishes, thus, avoiding duplications caused by single inheritance.

2.2.1 Object-oriented decomposition

The idea to draw subsystem boundaries around data and associated functions is one of the most important aspect of object-orientation. Actually, it is a simple application of information hiding [Parnas72]. That way, the representation of data and the implementation of functions is hidden from other subsystems. Object-orientation supports this modularity by encapsulation (see section 2.2.2) and boosts it by allowing multiple instances of modules. The latter lifts a tool to control software complexity to a paradigm for real world modeling.

Right in the spirit of SIMULA object-oriented systems simulate real world processes. Each participant in the real world can be represented by a software artifact called object. In an object-oriented bureau we would not see functions like "*file content of incoming letter into drawer*", but objects like **Letter** (supporting retrieval of content) and **Drawer** (supporting addition of information). Clearly, objects offer services and do not prescribe the order of actions. The object-oriented secretary becomes a conductor of object capabilities as opposed to a hierarchical manipulator of materials. As a result, the design is much more stable, because even if processes in the bureau are drastically changed the basic participants (objects) are likely to stay. This promises less software maintenance in case of change but also greater potential for reuse of objects in other domains (e.g., **Letter** will be useful in other domains as well). Again, the reason for this reuse potential is the absence of a rigid top-down conceived function call structure. The bottom-up provision of general services is more suited to be reused in altered conditions.

The recursive decomposition of data works especially fine due to the concept of procedural abstraction (see section 2.2.5 on page 35). In the object-oriented bureau objects issue goals to each other. The secretary may ask a drawer to file an information. The drawer in turn forwards the goal to one of its sub-compartments and so on until the goal is achieved. The secretary does not need to know about the internal organization of drawers which also do not care what types of sub-compartments exist. Clients never care about the types of their servers. Consequently, servers may be exchanged without need to alter clients. This would not be the case if, e.g., a secretary took different actions depending on drawer types.

2.2.2 Encapsulation

We already noted above that encapsulation supports the mutual protection of independent software entities. This observation is important for programming in the large. For programming in the small it is important to recognize the support of encapsulation for data integrity. An object is responsible for its own initialization after creation. It is ought to establish a consistent state at this point. Later manipulations to the object's state may only occur through the associated procedures. These, however, are designed to keep the object's state consistent as well. It is not possible to accidentally add an element to the bottom of a **Stack** because one has access to the stack's **List** representation and choose an invalid operation. A client of the chessboard data structure from section 1.3.2 on page 21 may insert an integer out of range which does not correspond to a column. Encapsulation would prevent a chessboard to get in such an invalid state.

For the reason that procedures and functions (i.e., methods) are defined on an encapsulated object it is reasonable to allow an implicit access to the object's features (methods and attributes). Hence, object descriptions get shorter and the tight binding of features to an object is emphasized. One can think of methods to always have one[2] (the object reference) implicit argument.

The combination of encapsulation and procedural abstraction (see section 2.2.5 on page 35) allows for a nice treatment of non-free abstract data-types. For instance, a rational number object may keep its numerator and denominator always in normalized form and clients cannot be confused by comparing non-normalized instances with seemingly unequal normalized instances.

Object-oriented languages differ in their encapsulation scope. A language with class encapsulation — such as C^{++} — allows methods to access internals of entities of the same class. A **Stack** class may access the representation of a stack argument in the implementation of a pushAllFromStack method. In a language with object encapsulation — such as EIFFEL — objects are protected against each other even if they are instances of the same class. The pushAllFromStack from above would have to use the public **Stack** interface (e.g., to pop all elements one by one) only.

2.2.3 Inheritance

In comparison to languages like PASCAL or C, object-oriented languages empower the programmer much more, since it is not only possible to define new types but these can be made to appear just like built-in types[3]. Even beyond, types can be incrementally derived from other types by inheritance. An heir inherits both interface (method signatures) and code (attributes and method bodies) from its ancestor. A typical use of inheritance is to specialize a type, e.g., derive a sports-car from a car. Other uses of inheritance are discussed in section 2.3.2 on page 41. Inheritance can be use to classify both data (see figure 2.1 on page 30) and algorithms [Schmitz92]. Therefore, it can be used as an organization principle for libraries.

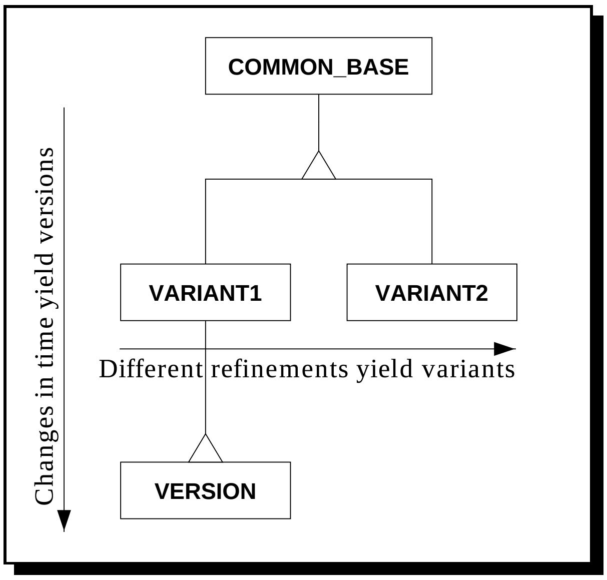

COMMON_BASE
VARIANT1
VARIANT2
VERSION
Different refinements yield variants
Changes in time yield versions

Figure 2.3: Classes as versions and variants

There is also an interesting correspondence between Inheritance and

[2]CLOS methods may break the encapsulation of more than one object.

[3]C^{++} and EIFFEL even allow user defined infix operators.

version / variant control[4] (see figure 2.3 on the preceding page). When inheritance is used to factor out code, that is, a set of subclasses uses and shares superclass code, it establishes a system of variants. Changes that should affect all classes are made in the common superclass. Changes meant for individual variants only are made to subclasses only. A subclass can be regarded as being a variant to its siblings or to be a version of its ancestor.

When object-oriented languages began to compete for market-shares their prominently advertised feature was inheritance. As a mechanism directly supporting reuse it was promoted as a unique and most important feature. The presence or absence of inheritance was used to distinguish between truly object-oriented or just object-based languages [Wegner90].

After all, inheritance turned out to be of less significance. The author rates data centered design, encapsulation, and dynamic binding to be of much more value. In fact, until today inheritance is not fully understood yet (see critique in section 2.3.2 on page 41). My view is supported by the fact that SMALLTALK-72 did not even have inheritance [Kay96], though its paragon SIMULA67 did.

2.2.4 Subtyping

We already got subtyping to know as a special kind of inheritance in section 2.1 on page 29. Yet, Subtyping has less to do with incremental class definitions but rather denotes a particular form of polymorphism called inclusion polymorphism [Cardelli & Wegner85]. Figuratively, one can assume the type **Figure** (see figure 2.4) to *include* the type **Circle**, since a **Circle** will behave exactly as a **Figure** when viewed and manipulated through a **Figure** interface. For subtyping, or the *Liskov Substitution principle* [Liskov & Wing93], to hold true it is important that not only argument types vary contravariantly and result types vary covariantly, but also the pre- and post-conditions must obey the same principle. That is, pre-conditions may be weakened and post-conditions can be be strengthened. Subtyping, hence, works especially well with a programming-by-contract model, which is supported by EIFFEL. Its language support for pre- and post-conditions promotes the verification of behavioral subtyping as well.

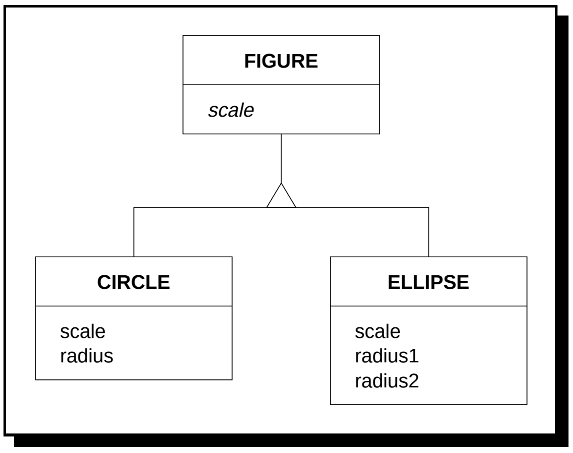


Figure 2.4: Subtyping inheritance

Inclusion polymorphism is one of the distinguished features of object-oriented languages in comparison to functional languages. Functional languages, typically offer so-called parametric polymorphism only[5]. For instance, in the type of *map*

[4]Consult [Schroeder95] for terminology, history, and classification of version controlling.

[5]Of course, subtyping is desirable for functional languages too and its integration into functional

(see typing judgment 1.7 on page 17) there is an implicit all-quantifier in front of the judgment. This type corresponds to unconstrained generic classes in object-oriented languages [Meyer92]. A variable of type **Figure**, however, is existentially quantified. The difference is that a parametric polymorphic function does the same for all instantiations of the type variable. Inclusion polymorphism just states that there is a concrete type which will behave individually.

Subtyping is of special value for the user of a library. It guarantees the substitutability of classes. A piece of code will work for all subtypes, if it works for the supertype. Library writers, on the other hand, often long for subclassing (for code reuse) and is-a relationships (for classification). For the reason of these diverging goals subtyping should not be tied to subclassing as it is the case in C^{++} or EIFFEL. Rare exceptions to this rule are POOL [America & v. Linden90] and SATHER [Murer et al.93a].

In sum, subtyping can be regarded as putting a discipline on inheritance (allowing conforming subclasses only) and polymorphism (excluding ad-hoc polymorphism, which allows no assumption about behavioral commonalities).

2.2.5 Dynamic Binding

Just as structure programming hid the functionality of goto behind if/else, while and do; OOP has hidden the functionality of indirect goto behind polymorphism.
– Robert C. Martin

Subtyping without existential qualification of type variables would be akin to HASKELL's type classes. This is a fine mechanism to make ad-hoc polymorphism less ad-hoc [Wadler & Blott89]. Contrary to an occasional misconception [Berger91] type classes do not allow for true object-oriented programming.

Most important for true object-oriented programming is the concept of dynamic binding. Subtyping just restricts dynamic binding "vertically" to the inheritance hierarchy. Now, a routine call to an object variable is not compiled as a standard subroutine call. The code to be executed is not determined before runtime. That is why, dynamic binding is often also called late binding. The code selection depends on the actual object contained in the variable at runtime. Consider the code

```
figure : FIGURE;

!CIRCLE!figure;
figure.display;
figure.typeMessage;
```

Although `figure` is of type **Figure** it refers to an object of type **Circle**. In figure 2.5 on the next page the lookup of methods is depicted.

All method searches start at **Circle**. As method display is defined by **Circle** it is used in spite the presence of a general display method in **Figure**. As innocent as this

languages is a important research topic [Läufer96].

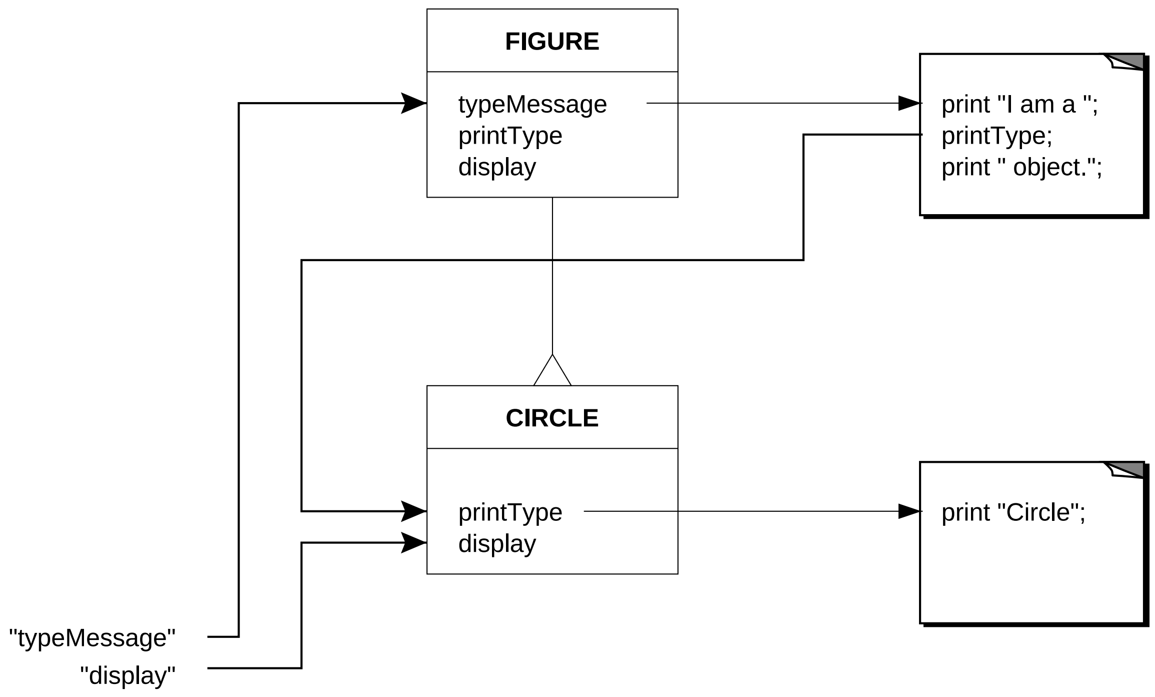


Figure 2.5: Dynamic method lookup

looks it is most important for the reuse promise of object-oriented programming, for it allows old code to use new code. For instance, a drawing editor may use an iterator to display all visible objects. If the editor uses the **Figure** interface to invoke display one can add new figure types without any need to change the iterator code. This inversion of control (don't call us — we call you), or Hollywood-principle, is typically not possible in procedural languages like C[6]. Indeed, however, it lifts reuse from plain library reuse to the reuse of design, that is, prefabricated application skeletons (called frameworks). The prefabricated code can be tailored through the use of dynamic binding.

For this to work properly it is most important that self calls (implicit calls to methods of the object itself) are also subject to dynamic binding. To see why, consider the typeMessage call in figure 2.5. First, it is redirected to **Figure** as there is no redefined version in **Circle**. The implementation of typeMessage prints a string template and calls printType to output the actual information. Now, instead of invoking the implementation in **Figure** this self-call is also late bound to the implementation in **Circle**. This leads to the desired result of printing "*I am a Circle object*". Late binding of self-calls, ergo, enable this factoring of code into code templates[7] that can be altered and refined by new classes.

Dynamic binding effectively defers behavior distinction from the main program to the datatypes. This opens up the possibility to build frameworks that do not

[6]One may exploit function pointers for this but it is not a common programming style.
[7]Method typeMessage is indeed part of the Template Method pattern [Gamma et al.94].

need to know about new types to come. Dynamic binding also makes it reasonable to operate with lists of heterogeneous types. Functional lists are homogenous, i.e., allow one element type only. One may use a sum type for homogenous lists but in order to differentiate actions according to individual element types type cases become necessary. Dynamic binding assigns the responsibility to differentiate to the elements and, therefore, avoids the type cases. Actually, the main "trick" used is the data centered approach, i.e., distributing operations to their data. Hence, case statements — otherwise contained in functions scattered around the whole program – are concentrated at data abstractions. Dynamic binding "merely" hides the indirection used to access the distributed operations. In this light it appears as language support for data centered programming which can be pursued also (by the albeit clumsy use of function pointers) in C.

Note that functional languages also allow using old code with new code. For instance, *map* can be used with any new function with appropriate type. Notwithstanding, this type of parameterization does not amount to the same flexibility, since the variable parts (e.g., function to map) must be specified in advance. An object-oriented framework, however, can be adapted in many unforeseen ways, as it is possible to override any methods — not just those intended for parameterization. For instance, **Circle** may also redefine typeMessage to print "*Circle, I am*".

Since the latter form of reuse requires knowledge about the internals of the reused classes it is called "white-box reuse". The corresponding name for composing opaque parts — just like higher-order functions — is "black-box reuse".

Akin to lazy evaluation (see section 1.2.4 on page 14) dynamic binding supports modularization. Analog to the separation of data generation and control the separation between method selection and method invocation decouples the client from the server. The client does not need to prescribe the server's action. It just declares a goal (through a message send) and the server is free to choose whatever appropriate action.

Almost all object-oriented languages use single-dispatch, i.e., dynamic binding is used for the receiver of a message send only. If dynamic binding is extended to the arguments of the message send it is called multi-dispatch. The languages CECIL [Chambers92b] and CLOS [Bobrow et al.86a] hereby allow implementations to be selected with respect to all argument types.

2.2.6 Identity

All objects are created unequal.
– me

The concept of object identity [Khoshafian & Copeland86] is easily overlooked due to the presence of more prominent concepts like inheritance, encapsulation, and polymorphism. Nevertheless it also significantly contributes to the expressiveness of the object-oriented paradigm. What does object identity mean? One way to look at it is to observe that objects are mutable. When an object changes, e.g., an element is inserted into a list, there is no new list being created. The list remains the same but has a different value. This property gives rise to easy real world modeling but also may induce problems due to unwanted aliasing. Aliasing occurs when two or more variables refer to one object and some interference takes place, e.g., a client with access to an object is not aware of other accesses and is invalidated by external changes made to its referenced object.

In contrast values (from functional programming) are immutable and lack identity [MacLennan82, Eckert & Kempe94]. It does not make sense to make a difference between two numbers of the same value or two lists with the same elements. This perspective, however, leads to the second way to look at identity: Consider personal data, such as name, age, city of birth, etc., held in records. Let us assume Ivan Smith exists twice with the same age. One Ivan is born in St. Petersburg (Russia) and the other is born in St. Petersburg (Florida). Unless we include the state of birth the two Ivan's have the same value, i.e., they are the same person in a functional language. Represented as objects, there is no problem at all. If one compares for data equality one will find out about the curiosity of the coincidental birth date. Yet, if one compares for identity it is clear that we have two distinct persons and not just one person that we somehow received twice. Objects may join their values but their identity remains [Harrison & Ossher93].

Summarizing, identity is very closely related to state, but one should emphasize its role in modeling real world objects with identity (A tank does not change only because its pressure does) and its role to distinguish between objects that happen to have the same value (Ivan is not Ivan).

2.3 Review

The following two sections give a subjective assessment of object-oriented programming from a software engineering point of view.

2.3.1 Pro

Why does the combination of the above presented concepts work well? This section examines the positive implications of the object-oriented paradigm.

2.3.1.1 Real world modeling

Whereas Turing machine behavior can be expressed by mathematical models, the observable behavior of interaction machines corresponds to that of empirical systems in the natural sciences.
– Peter Wegner

Two properties allow for real world modeling without *impedance mismatch*[8]:

[8]When a modulated current flows through a conductor which is not properly terminated some energy will be lost due to reflections. Similarly, "energy" will be lost if the modeling domain cannot adequately express the things to be modeled.

1. The emphasis on a collaborative set of self-sustained objects and
2. direct support for the notions of state and identity.

Real world systems are often composed of a collaborative set of self-sustained objects themselves and their structure provides a natural starting point for an object-oriented design. Maintaining the system, thus, will be possible without reconstructing too many transformation and abstraction steps. An insufficient understanding of the system would lead to faulty changes or bug fixes.

With regard to state support it is instructive to note that all of the arguments against reduction semantics (in section 1.3.2 on page 21) vanish. Furthermore, the usually as negative discredited implications of state support like references and aliasing are by no means just a crude copy of an underlying primitive von Neumann hardware. Consider an office software where you may embed pictures into word-processor texts. Often we want to share images between multiple texts. A correction to the image should be reflected in all uses of the image[9]. This is a truly useful use of aliasing. Also, we may have a link to our office software represented by an icon to click on in order to start the software. The link allows changing the location or version of the software without bothering the user with a new place to start from or a new icon to use. This is a truly useful use of referencing. Changes to documents are modifications to mutable objects with an identity. A model reflecting this employs a truly useful use of state. Given these useful applications of identity and state it seems worthwhile to assign them direct language support.

2.3.1.2 Maintainability

> *"Software systems are long-lived and must survive many modifications in order to prove useful over their intended life span. The primary linguistic mechanisms for managing complexity are modularity (separating a system into parts) and abstraction (hiding details that are only relevant to the internal structure of each part). A challenge for future language design is to support modularity and abstraction in a manner that allows incremental changes to be made as easily as possible. Object-oriented concepts have much to offer and are the topic of much on-going investigation [Hankin et al.97]."*
>
> – Chris Hankin et al.

Seamless development All phases of a software development (requirements analysis, analysis model, design, implementation) are based on the same foundation, hence, avoiding paradigm shifts inbetween. Phase transitions are considered seamless [Walden & Nerson95] and, ergo, backward or forward changes have less implications compared to a methodology involving paradigm shifts, i.e., redesigns.

[9]A individual change can be realized by first making a copy of the image.

Object-oriented decomposition Building up on a base of reusable self-sustained domain or business objects yields more stable systems. There is no *main function* that can be invalidated by a series of requirement changes. The introduction of new data — i.e., the "know how" of a system is applied to a new domain — is easy, for it amounts to a local addition of code only. Similarly to extensions, changes should occur more localized too. The distribution of functionality from a hierarchy of functions to the former "dead data", consequently, can be thought of as a complexity reducing mechanism. Class definitions create a small working space and make the first argument of their methods implicit. Changes, ergo, become a matter of altering a small encapsulated system fraction with short access to local data. Usually, there is no need to teach binding environments and techniques like display tables or static links anymore due to the simple structure of classes[10].

Modularity We already explained in the preceding concept sections how encapsulation (section 2.2.2 on page 32) and procedural abstraction (section 2.2.5 on page 35 and section 2.2.1 on page 32) aid the modularity of programs. Clients do not care how servers fulfill their services and can rely on consistent and self-managing objects at any time.

2.3.1.3 Beyond the imperative

Although the object-oriented paradigm is closely related to an imperative style of programming it has been argued that there are important differences [Kühne96a]. Objects successfully hide the use of state and represent explicit substates to change and to handle, hence, escape the von Neumann bottleneck. The famous critique of John Backus [Backus78] towards imperative languages like FORTRAN and ALGOL must be re-evaluated with regard to the object-oriented paradigm. Let us recall the quotation of Alan Kay on page 29: He viewed objects as a decomposition of the whole computer with equal power. Now, the whole computer and its decomposition objects do not need to be von Neumann architectures. It is possible to use objects in a referential transparent way [Meyer94a] or as nodes in a dataflow graph.

> *"What I got from Simula was that you could now replace bindings and assignment with* ***goals****. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. … It is unfortunate that much of what is called 'object-oriented programming' today is simply old style programming with fancier constructs. Many programs are loaded with 'assignment-style' operations now done by more expensive attached procedures. [Kay96]"*
>
> – Alan Kay

In particular, the improved control abstraction facilities of object-oriented languages compared to early imperative languages weaken Backus' critique. We will return to the object-oriented possibilities to escape the limitations of plain imperative languages in section 4.2 on page 57.

[10] C^{++} and JAVA allow nested classes though.

2.3.2 Contra

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.
– Rich Cook

2.3.2.1 Imperative heritage

Most object-oriented languages are based on ALGOL-like languages (C^{++}, SIMULA, EIFFEL) and some on LISP (SMALLTALK, CLOS, DYLAN). All share an imperative base trough their ancestors. While we argued in section 2.3.1.2 on page 39 that object-oriented languages reach beyond their imperative roots, they still suffer from their imperative heritage. One example are system shut-downs due to method calls on void references. This is clearly an initialization problem which in general has already been criticized by John Backus [Backus78] and which we will tackle in chapter 11 on page 191.

Aliasing Consider a program fragment using a complex number library:

```
!!a.make(2,0);
b:=a;
c:=a+b;
!!a.make(0,2);
d:=a+b;
```

We would expect d to have the value $2+2i$ but if complex numbers are implemented with reference semantics — which is quite likely to be the case due to efficiency considerations — the result will be $4i$. The reason for this is the unexpected aliasing of b with a. Though object-oriented languages typically provide support for value semantics (e.g., expanded classes in EIFFEL, part-objects in BETA, non-reference variables in C^{++}) there is no agreed system how to choose between reference and value semantics.

Another source of aliasing are interferences between parameters and results. A method does not represent a good abstraction if the effects of

```
o.m(x, y)
```

differ to

```
o.m(x, x).
```

Also,

```
aMatrix.multiply(aMatrix2)
```

possibly yields the wrong result when `aMatrix2` happens to reference `aMatrix`, since then result elements will override still to be used argument elements.

Broken encapsulation When object state is distributed over several sub-objects and these objects are accessible from outside the object the state-encapsulation of the object is broken. For instance, a careful bank customer, aiming at never having less than $2000 on his account, will be invalidated by his wife, if she has access to the account too [Duke95]. Similarly, a balanced tree can get unbalanced if one of its element is changed by a side-effect. A solution for these problems has been proposed by the use of so-called *Islands* [Hogg91].

2.3.2.2 Classes for everything

The class concept is used for representing modules, encapsulation borders, instance generation, code pools, types, visibility mechanism, and interfaces. Although the strive for unification, i.e., to capture many things with a single abstraction, is desirable, the wealth of a class' functions appears too much. There has been arguments to separate modules and classes [Szyperski92], code pools and types [Murer et al.93a], and encapsulation and classes [Limberghen & Mens94]. So-called Kinds have been proposed to aid classes with typing [Yu & Zhuang95] and "namespaces" are supposed to cluster C^{++} classes. Obviously, there will be more research necessary to remove the overloading of concepts on classes.

2.3.2.3 Inheritance for everything

A modeling tool is more than a passive medium for recording our view of reality. It shapes that view, and limits our perceptions. If a mind is committed to a certain [tool], then it will perform amazing feats of distortion to see things structured that way, and it will simply be blind to the things which don't fit that structure [Kent78].
– W. Kent

Inheritance has been the "hammer" for many problems (be they "nails" or not) for both language designers and language users. Though inheritance is primarily an incremental modification operator on classes it fulfills many purposes:

Language designers use inheritance for

Subtyping An heir is implicitly assumed to be a subtype to its ancestor and, thus, is allowed to be substituted for it.

Code reuse Code reuse happens through inheriting methods which continue to work in the heir but also by deliberately importing methods only meant to be used for the implementation of the heir.

Interfaces So-called pure, abstract, or deferred classes only specify interfaces and do not contain any code. They are used to define an interface that implementations may adhere to.

Modules The collection of mathematical constants in the EIFFEL library is an example for solely using a class as a namespace and container of data without intending to use it as a instance generator.

Mixins Languages with multiple inheritance suggest a LEGO approach to the composition of code. Each superclass mixin provides a facet of a class like persistence, observeability, etc. This style is also referred to as *facility inheritance* [Meyer94b].

Language users exploit inheritance for

Classification Libraries are sometimes organized as a taxonomy of concepts (see figure 2.1 on page 30). This helps to understand classes in terms of others. Also, algorithms may be subject to classification [Schmitz92]. Each branch in the inheritance hierarchy then represents a design decision.

Specialization Heirs narrow the scope of ancestors, e.g., a **Student** is a special **Person**, typically by adding information (e.g., student identification). Specialization may conform to subtyping (like in the **SportsCar** and **Car** example in figure 2.2 on page 31) but does not necessarily (a **3DPoint** is not a subtype of **Point** due to the covariant equality method).

Generalization Heirs cancel limitations of their ancestors. For instance an all purpose vehicle may be subclassed to obtain an all purpose vehicle that allows steering of the rear axle. This usage is typically due to unforeseen extensions and coincides with code reuse.

Versioning As explained earlier (see figure 2.3 on page 33) it is possible to use heirs as versions of their ancestors. This allows extending data records in a system while reusing all of the control code unchanged.

Parameterization Partially abstract classes are used as ancestors to concrete subclasses that specify the missing code. This type of code parameterization has been described as the Template Method pattern [Gamma et al.94].

The problem with all these uses for inheritance is that always the same rules for visibility, substitutability, and namespaces are applied. While breaking encapsulation is necessary for white-box reuse it leads to *fragile base classes* in general [Snyder86, Kühne95b]. Also,

- inheritance used for code reuse or covariant specialization should not enable substitutability [LaLonde & Pugh91].
- Inheritance used for parameterization should not break encapsulation [Kühne95b].
- Interface inheritance should not allow covariant redefinitions of arguments [Cook89b], etc.

Consult section 2.3.2.2 on page 42 for a list of suggestions to escape the "one-trick-pony" mentality towards inheritance in object-oriented languages.

Especially multiple inheritance causes additional problems like aggravating encapsulation, modularity and the open-closed principle [Carré & Geib90], which is why many languages do not provide it (e.g., SMALLTALK, BETA, JAVA).

It can be argued about whether it is positive or negative that inheritance typically is a static mechanism. There is no doubt, however, that inheritance should not be used when more dynamic solutions are appropriate. We will return to this subject in chapter 7 on page 93.

3 Calculus comparison

As we are going to subsume functional concepts with the object-oriented paradigm in chapter 4 on page 55 and aim at capturing functional techniques with object-oriented design patterns in part II starting at page 85 this chapter shall assure us about the validity of this approach.

Of course, all functional and object-oriented languages are Turing complete with regard to their computation power. There is no doubt that any of these languages can emulate one of the other. In spite of that it is quite interesting to ask whether one emulation is easier to accomplish than the other. If it is easier to capture object-oriented programming with a functional language we should write patterns facilitating object-oriented programming in a functional language. The goal of this chapter is to prove the opposite.

3.1 Language comparisons

Looking for the computational difference between two languages is like comparing apples to oranges only to find out that both are fruit.
– me

What is left if we beware of falling into to the Turing tarpit, i.e., try to find a difference in computational power that is simply not there? What other indicators for expressiveness are available? One interesting approach is to evaluate the amount of restructuring a program necessary to emulate the addition of new construct [Felleisen91]. If local changes are necessary only then the new construct can be judged to add no further expressiveness. Unfortunately, the theoretical framework used for this approach does work for conservative extensions to languages only, i.e., cannot be used to compare two fundamentally different languages.

For the purpose of this chapter my idea was to define a translation of one language into the other and vice versa and then compare the translation complexities. The rational is that a language should emulate a less expressive language with ease while a considerable amount of machinery would be necessary the other way round.

In order to compare the fundamental mechanisms instead of technicalities of specific languages it appeared most reasonable to compare calculi expressing the nature of the two paradigms respectively. The calculus representing functional programming is easily found and is referred to as the λ-calculus. You might want to refer to an introduction for its syntax and semantics [Barendregt84, Hankin94]. It is much less obvious to choose a calculus representing the object-oriented paradigm. Many approaches to model object-oriented principles with calculi use an extended version of the λ-calculus or do contain first-order functions [Abadi94, Bruce94, Cardelli84, Cook89a, Fisher & Mitchel95, Pierce & Turner94, Reddy88]. A comparison with such a calculus would be ridiculous because of the trivial emulation of the λ-calculus within it. When I made the decision which calculus to choose there where two candidates that did not model objects as records in a functional language but attempted to model object-oriented languages from first principles. One was Opus [Mens et al.94, Mens et al.95] and the other was the σ-calculus [Abadi & Cardelli94]. In fact, the latter calculus was inappropriate too for it allowed to bind the self reference of an object to an arbitrary name and, thus, enabled to access variables of arbitrary nesting depth. In a comparison carried out by Dirk Thierbach this feature was found out to be very “functional” in nature [Thierbach96] and also would have prevented a fair comparison. Thierbach also identified several weaknesses of Opus and developed an improvement called Opulus. He proved Opulus to have the Church-Rosser property and gave a comparison to both Opus and σ-calculus [Thierbach96].

3.2 Opulus

Opulus is a simple calculus that denotes

• encapsulation of multiple methods to an object by	$[E]$,
• referencing “Self” as	\$,
• sending a message N with Argument F to object E with	$E\ N : F$,
• referencing the parameter of a message send as	#,
• incremental modification (E is subclassed by F) by	$E + F$,
• declaration of attributes as	$N = E$,
• declaration of methods (with Name N and body E) as	$\lambda N = E$,
• the empty object by	$\square$,
• and finally, hiding of methods (N_1–N_k) with	$E\{N_1, \ldots, N_k\}$.

Table 3.1: Opulus syntax

Its grammar (with start-symbol “Expression”), therefore, is as given in table 3.2 on the facing page.

For a more thorough definition please consult [Thierbach96]. For our purposes we will be satisfied with just one example showing how the calculus works. Con-

Expression	::=	”[” Expression ”]” | Expression ”+” Expression |
		Expression ”{” Namelist ”}” |
		Expression Name ”:” Expression |
		Name ”=” Expression | ”λ” Name ”=” Expression |
		”#” | ”$” | ”□” |
		”(” Expression ”)”
Namelist	::=	Name ”,” Name
Name	::=	Character { Character }
Character	::=	”a” | ”b” | . . . | ”z”

Table 3.2: Opulus BNF

sider an object *POINT*:

$$POINT \equiv [x = 7 + y = 13 + \lambda setx = [\$ + x = \#]].$$

It has two attributes x and y and one method *setx* that changes the value of x by producing a new object. The new object is the old object (referenced by $) modified by an attribute x that has the value of the *setx* parameter.

The reduction occurs as follows:

$$\begin{aligned} & POINT\ setx : 42 \\ \rightarrow_\beta\ & app(POINT, setx, \square, 42) \\ \equiv\ & app(\lambda setx = [\$ + x = \#], setx, x = 7 + y = 13 + \lambda setx = (\$ + x = \#), 42) \\ \equiv\ & [x = 7 + y = 13 + \lambda setx = (\$ + x = \#) + x = 42] \\ \twoheadrightarrow_\eta\ & [x = 7 + (y = 13 + (\lambda setx = (\$ + x = \#) + x = 42))] \\ \rightarrow_\eta\ & [y = 13 + (\lambda setx = (\$ + x = \#) + x = 42))] \end{aligned}$$

The result is a new object with x possesing the value 42. It is one of the distinguishing features of OPULUS to be able to get rid of the overriden “$x = 7$” attribute by η-reduction.

3.3 Opulus within λ-calculus

A standard translation of OPULUS into the λ-calculus represents OPULUS terms in a tree structure and uses programmed reduction rules to manipulate the tree [Thierbach96]. The fixpoint combinator Y can then be employed to, e.g., repeatedly apply β-reduction.

A more efficient translation is achievable

- by recognizing that substitution in OPULUS (e.g., replacing # with the argument value) can be captured with λ-abstraction variable bindings,

- using a continuation passing style for method calling that might “backtrack” to search in A after B in the case of $(A+B)$ *message* $: \Box$,
- and by directly translating OPULUS terms into structure evaluating functions instead of separating term representation and reduction rule application [Thierbach96].

The complete (fast) translation of OPULUS to the λ-calculus is given in table 3.3 (parameter d refers to the self-reference, e to the value of an argument, n to the name of a method, and f to the continuation to be used in case a message must be redirected to the modified operand, e.g., A in $A+B$).

$$
\begin{array}{lcl}
\ll \$ \gg & \equiv & \mathit{self} \\
\ll \# \gg & \equiv & \mathit{arg} \\
\ll \Box \gg & \equiv & Y\ \lambda y.\lambda d\,e\,n\,f.y \\
\ll [A] \gg & \equiv & \lambda d\,e\,n\,f.((\lambda x.xx) \ll A \gg)\ e\,n\,f \\
 & \equiv & \lambda d.(\lambda x.xx)^1 \ll A \gg \\
\ll A+B \gg & \equiv & \lambda d\,e\,n\,f. \ll B \gg d\,e\,n\,(\ll A \gg d\,e\,n\,f) \\
\ll AM:B \gg & \equiv & \ll A \gg \ll \Box \gg \ll B \gg \ll M \gg \\
\ll \lambda N = A \gg & \equiv & \lambda d\,e\,n\,f. \ll N \gg ((\lambda \mathit{self}.\lambda \mathit{arg} \ll A \gg)\ d\,e)\ f\,n \\
\ll N = A \gg & \equiv & \lambda d\,e\,n\,f. \ll N \gg \ll A \gg f\,n \\
\ll A\{N\} \gg & \equiv & \lambda d\,e\,n\,f. \ll N \gg f\ (\ll A \gg d\,e\,n\,f)\ n
\end{array}
$$

Table 3.3: Translation of Opulus to λ-calculus

It should be noted that η-reductions are not considered anymore and that a translation for OPUS would not be possible in this manner.

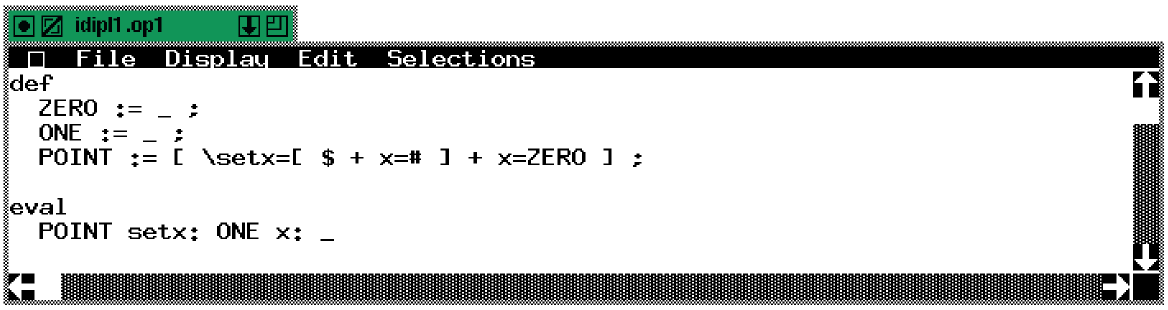
The translation appears straightforward (e.g., message send corresponds to function application) but we should pay attention to the fact that OPULUS names (such as method names) must be encoded in the λ-calculus, since it does not support a similar notion. Thierbach translated names to an encoding of numbers in the λ-calculus, because all one really needs is to compare names for equality. One can, alternatively, also think of an extended λ-calculus with δ-reductions for numerals or names. In our case, however, comparing names adds a complexity factor of $O(v)$, where v is the number of distinct names used in the OPULUS term.

Figures 3.1 and 3.2 on the facing page show an OPULUS term and its mechanically[2] derived translation to the λ-calculus respectively. Both are screenshots taken of a system generated by CENTAUR [Despeyroux88, Jager et al.91, Centaur92] from specifications written by Dirk Thierbach [Thierbach96].

The M and N macros in figure 3.2 on the next page represent the encodings of OPULUS names.

[1]An object has itself as an implicit argument.

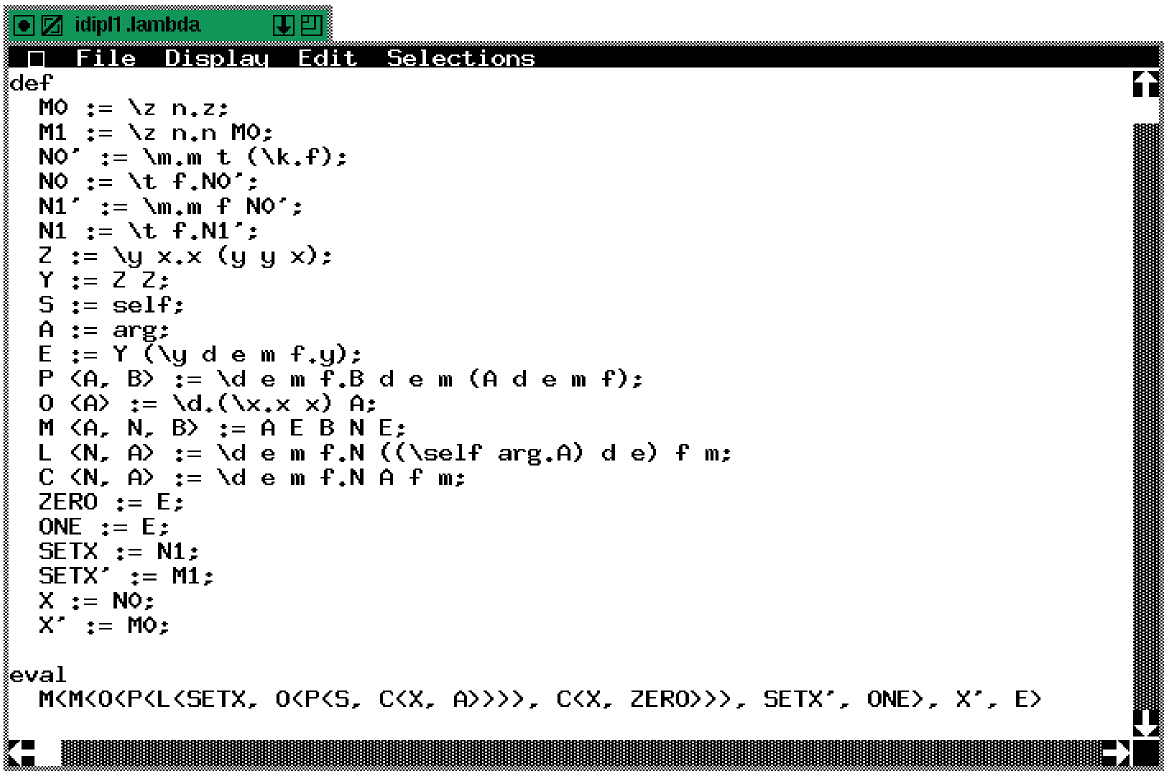
[2]Some minor modifications have been made by hand to enhance clarity.



```
def
  ZERO := _ ;
  ONE := _ ;
  POINT := [ \setx=[ $ + x=# ] + x=ZERO ] ;

eval
  POINT setx: ONE x: _
```

Figure 3.1: Screenshot: OPULUS expression



```
def
  M0 := \z n.z;
  M1 := \z n.n M0;
  N0' := \m.m t (\k.f);
  N0 := \t f.N0';
  N1' := \m.m f N0';
  N1 := \t f.N1';
  Z := \y x.x (y y x);
  Y := Z Z;
  S := self;
  A := arg;
  E := Y (\y d e m f.y);
  P <A, B> := \d e m f.B d e m (A d e m f);
  O <A> := \d.(\x.x x) A;
  M <A, N, B> := A E B N E;
  L <N, A> := \d e m f.N ((\self arg.A) d e) f m;
  C <N, A> := \d e m f.N A f m;
  ZERO := E;
  ONE := E;
  SETX := N1;
  SETX' := M1;
  X := N0;
  X' := M0;

eval
  M<M<O<P<L<SETX, O<P<S, C<X, A>>>>, C<X, ZERO>>>, SETX', ONE>, X', E>
```

Figure 3.2: Screenshot: OPULUS expression represented in the λ-calculus

3.4 λ-calculus within Opulus

Again a standard translation is easily obtained by encoding λ-terms into objects which understand application and substitution messages [Thierbach96]. The difficulties with α-conversions necessary for β-reductions are easily avoided by calculating weak-head-normal-forms only [Field & Harrison88].

A cleverer translation is obtained by exploiting the closure like nature of objects. Closures are functions that carry their variable environment with them [Field & Harrison88]. How can we make use of closures then? It is easy to translate the identity function ($\lambda x.x$) to OPULUS as

$$[\lambda app = \#].$$

Then the application to an argument

$$[\lambda app = \#]\ app : \square$$

yields the argument itself, i.e., $\square$.

However, we need a more sophisticated approach in the case of $\lambda x.\lambda y.x$. When $\lambda y.x$ is applied to a value we need to remember the value of x. This is precisely the case for a closure which can remember earlier variable bindings. Hence, the result of applying $\lambda x. \ldots$ must be a closure representing the function $\lambda y.x$ but remembering the value of x. A closure, howbeit, is most easily represented by an object storing the variable binding in its state, ergo

$$[\lambda app = [x = \# + \lambda app = (\$\, x : \square)]]$$

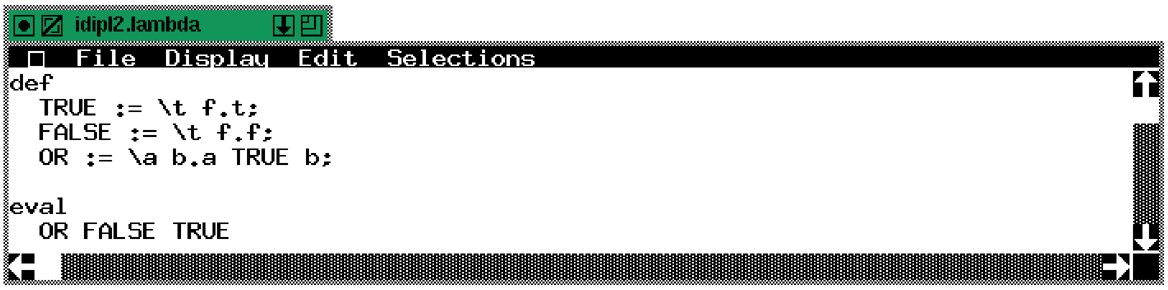
represents the function $\lambda x.\lambda y.x$ (the sub-term "$\$\, x : \square$"retrieves the function body value from the variable environment). Note that if the lambda term had been $\lambda x.\lambda y.y$ the variable access of the innermost function body translation would have been "#" instead of "$\$\, x : \square$". As a result, we need a context dependent translation of parameter access [Thierbach96]. In table 3.4 the parameter context is denoted with subscript indices.

$$\begin{aligned}
\ll \lambda X.B \gg \; &\equiv \; \lambda app = \ll B \gg_X && (3.1)\\
\ll \lambda X.B \gg_Y \; &\equiv \; [\$ + Y = \# + \lambda app = \ll B \gg_X] && (3.2)\\
\ll L\, R \gg \; &\equiv \; \ll L \gg \; app : \ll R \gg && (3.3)\\
\ll L\, R \gg_Y \; &\equiv \; \ll L \gg_Y \; app : \ll R \gg_Y && (3.4)\\
\ll X \gg_X \; &\equiv \; \# && (3.5)\\
\ll X \gg_Y \; &\equiv \; (\$\, X : \square), \quad \mathit{if}\ X \not\equiv Y && (3.6)\\
\ll X \gg \; &\equiv \; (\$\, X : \square) && (3.7)
\end{aligned}$$

Table 3.4: Translation of λ-calculus to Opulus

The subscript index is defined and respected by lambda abstraction (rules 3.1 and 3.2 of table 3.4) and distributed over application (rules 3.3 and 3.4). If the current binding variable coincides with the variable to be accessed then the parameter symbol is used (rule 3.5). Otherwise, the variable is accessed via the variable environment (rules 3.6 and 3.7).

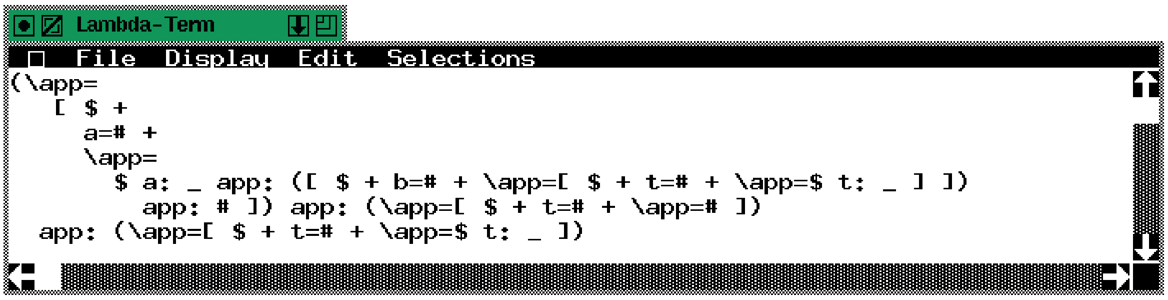
Interestingly, in contrast to the standard translation the fast translation above even allows to dispose the restriction of calculating the weak-head-normal-form only. Figure 3.3 on the next page shows a lambda expression in a CENTAUR window. Figure 3.4 on the facing page shows the corresponding OPULUS term which was derived by automatic translation.



```
def
  TRUE := \t f.t;
  FALSE := \t f.f;
  OR := \a b.a TRUE b;

eval
  OR FALSE TRUE
```

Figure 3.3: Screenshot: λ-calculus expression



```
(\app=
   [ $ +
     a=# +
     \app=
       $ a: _ app: ([ $ + b=# + \app=[ $ + t=# + \app=$ t: _ ] ])
         app: # ]) app: (\app=[ $ + t=# + \app=# ])
  app: (\app=[ $ + t=# + \app=$ t: _ ])
```

Figure 3.4: Screenshot: λ-calculus expression represented in OPULUS

3.5 Conclusion

It did not come as a surprise that either translation was possible at all but how about the results of comparing the complexity of performing one calculus in the other? Table 3.5 on the next page summarizes the results for emulating OPULUS in the λ-calculus and table 3.6 on the following page shows the results for emulating λ-calculus in OPULUS. Ranges of complexity denote the best and worst case, whereas the average case is always closer to the lower bound.

Let us first regard the standard translation which we did not explicate in the preceding sections but that we remember as using a straightforward term representation with associated reduction functions. We note that the derivation of emulating the λ-calculus in OPULUS was done within two pages [Thierbach96] resulting in three conceptually easy object definitions all implementing two messages each (substitution and β-reduction). Deriving the opposite standard translation took more than three pages and resulted in nine translation rules needing nine constructors with nine associated selection functions. In addition, this does not include the emulation of OPULUS names in the λ-calculus.

The difference in verbosity of the two emulations can be explained by two reasons:

1. The λ-calculus is somewhat weaker in its basic operations causing the need for a name comparison emulation and

2. OPULUS is a somewhat bigger calculus with a larger grammar causing the need for more (nine compared to three) emulation rules.

On the one hand the size of OPULUS may appear arbitrary given the current maturity of object-oriented calculi and one may easily excuse the λ-calculus for its efforts to emulate such a "clumsy" formalism. On the other hand, OPULUS does a very good job of capturing the basic mechanisms of class based object-oriented languages. Capturing the others calculus' richness is part of the game and the λ-calculus has obviously a harder job.

The first of the above argument also shows up in the complexity comparison: One full reduction step in OPULUS needs O(nv) steps in the λ-calculus (see table 3.5), whereas v is the number of distinct names in the OPULUS term.

	Fast translation	Standard translation
Substitution	O(1)	O(n)
Application	O(nv)	O(nv)
Redex contraction	O(nv)	O(nv)
Redex recognition	O(1)	O(1)–O(nv)
Reduction-step	O(nv)	O(nv)–O(n^2v)†

† It easy to reduce this to O(nv) by combining the functions that search a redex and perform a reduction.

Table 3.5: Complexity of Opulus in the λ-calculus

	Fast translation	Standard translation
Substitution	O(n)	O(n)
Redex contraction	O(n)	O(n)
Redex recognition	O(1)	O(1)–O(n)
Reduction-step	O(n)	O(n)

Table 3.6: Complexity of λ-calculus in Opulus

It is interesting to note that the built-in name comparison of OPULUS causes extra work when emulated in the λ-calculus. Yet, it should not be given to much importance, since the λ-calculus can easily be extended with δ-reductions that accomplish the required comparisons with O(1) complexity.

So, abstracting from the name evaluation overhead the two standard translations essentially do not differ (compare table 3.5 and table 3.6). Let us turn to the fast translations: It immediately strikes our attention that redex recognition is O(1) in either translation. This is due to the fact that redexes are directly translated into redexes of the other calculus. Besides this exception, there is the usual overhead for name comparisons in the λ-calculus. Notwithstanding, substitution in OPULUS (replacing $ and #) is just O(1) in the λ-calculus. This is achieved by using β-reduction, i.e., the application of $\lambda self.\lambda arg. \ll A \gg$ to the values of $ and #, reduces in two (constant) steps.

Substitution of λ-calculus terms in OPULUS has linear complexity, because variables have to be looked up in the variable environment. In an object-oriented programming language this step is constant too, since it simply requires an attribute access or — in the worst case — an access through an indirection table in case of an overridden attribute.

So, on the one hand substituting λ-calculus terms takes linear time in OPULUS but on the other hand note that looking up OPULUS methods takes linear time in the λ-calculus (see row "Application" in table 3.5 on the facing page). There is no equivalent to looking up method names in the λ-calculus which is why there is no such entry in table 3.6 on the preceding page. De facto, these prominent difference in complexity of emulations point out a crucial difference of the underlying paradigms: Object-orientation is characterized by the atomicity of looking up a name and performing the corresponding method. Functional programming is characterized by having access to function parameters at any place in any nesting depth.

Overall, the two calculi appear to be almost equal in expressiveness[3], since their translations involve linear complexity respectively (excluding the overhead for comparing names in the λ-calculus). Where differences are present (method lookup and variable substitution) they point out paradigm fundamentals.

Especially, the fast translations — we might also say native translations, since idiomatic constructs were used — demonstrated a very direct way of translating calculi constructs: Functions in the λ-calculus could be captured by objects operating as closures. Objects in OPULUS could be represented by λ-terms and selector functions, e.g., an object with the fields *a*, *b*, and *c* —

$$\lambda s.s\ a\ b\ c$$

— receives a message (selector) "*b*" by being applied to $\lambda x\,y\,z.y$, yielding field *b*.

Finally, one may note that fast and standard emulation of the λ-calculus are very close to each other (see table 3.6 on the facing page). This may hint to the fact that objects are a good abstraction for simulation. The direct simulation of λ-calculus was almost as efficient as the version that exploited native features of OPULUS.

Also, we may recall that an object-oriented languages would not need linear complexity to look up λ-variables in the environment. In contrast, an implementation of OPULUS in a functional programming language will still need linear complexity to look up names, unless further efforts like introducing method lookup arrays or similar machinery is introduced.

Before we derive further conclusions from the translation complexities found here, a small investigation is in order. Apparently, translation complexities depend

[3]You may want to consider [Thierbach96] for a full account including η-reductions.

on the translation strategies. What if a better scheme, say for translating OPULUS into the λ-calculus, could be found? Is it not possible that one calculus is still superior but we have not found the right translation scheme yet? The fact that both translations exhibit mostly linear characteristics gives us confidence that we found optimal translations and future translation schemes will not affect the overall result. But what about sub-linear translations? Are they possible? Consider, a translation from a RAM model with just numerical addition into one that features multiplication also. When addition loops are converted into multiplications this could result into a sub-linear translation. However, the reverse translation would obviously be super-linear. If one our translations could be optimized to a sub-linear translation, the other one could not possibly have linear characteristics right now. Proof by false assumption: Assume the forth translation to be sub-linear and the back translation to be linear. With repeated back and forth translations a program could be made infinitely faster. As the conclusion is obviously false, so must be the assumption [Thierbach97].

In summary, it is certainly not a reverse logic approach to aim at subsuming functions with objects. While the other way round is not impossible either it appears less natural. Moreover, when leaving the level of calculi, it appears more reasonable to subsume reduction semantics with state than to “ruin” the basic assumption of referential transparency with the introduction of stateful objects. Note, however, the striking correspondence of some well-known combinators with imperative statements: $S \equiv$ “`:=`”, $K \equiv$ “`const`”, $I \equiv$ “`goto`”, and $CB \equiv$ “`;`” [Hudak89], suggesting how to emulate imperative features in a declarative language. I will continue the discussion about the conflict between state and referential transparency in section 4.1.1 on the next page and section 4.2.1 on page 57.

4 Conflict & Cohabitance

When two worlds collide the question is whether they cancel out or enlighten each other.
– me

Now that we know the foundations of both functional and object-oriented programming and convinced ourselves that subsuming functional programming with an object-oriented language works fine at the calculus level it is time to refocus our overall goal. Part II starting at page 85 is going to present functional concepts made amenable to object-oriented design by capturing them in design patterns. But which functional concepts are appropriate? This chapter discusses functional and object-oriented concepts that seem to be at odds with each other (section 4.1) and shows a path of possible integration (section 4.2 on page 57) partly reconciling conflicts discovered in the following section.

4.1 Conflict

Although, the preceding chapter gave us confidence that an integration of functional concepts into an object-oriented language is viable it might be the case that incommensurable properties prevent an integration at the programming language level. Indeed, the following sections reveal some immediate oppositions and redundancies.

4.1.1 Oppositions

This section discusses diametral oppositions of the functional and the object-oriented paradigm.

4.1.1.1 Semantic model

As elaborated in section 1.2.2 on page 11 functional programming is founded on reduction semantics, i.e., excludes the presence of side-effects. Object-orientation, however, relies on stateful objects (see section 2.2.6 on page 37). This is a serious conflict. Abandoning either reduction semantics or stateful objects seems to destroy one of the paradigm foundation piles respectively. Actually, the integration of functional and object-oriented features amounts to combining both declarative and algorithmic language paradigms (see figure 4.1 on the next page).

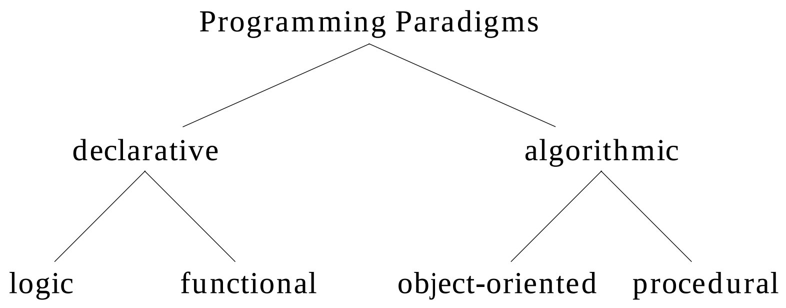


Figure 4.1: Classification of programming paradigms

4.1.1.2 Decomposition

There is a fundamental dichotomy between the decomposition strategies in functional and object-oriented programming on two levels: On a large scale we must decide whether to decompose a software system into functions (section 1.2.1 on page 10) or into objects (section 2.2.1 on page 32) [Meyer88]. On a small scale we have to choose between data abstraction (section 1.2.1 on page 10) or procedural abstraction (section 2.2.1 on page 32) [Cook90]. Either we package data constructors and use functions that dispatch on those or we package functions and distribute them to their respective constructors.

4.1.1.3 Evaluation

In section 1.2.2 on page 11 and section 1.2.4 on page 14 we have learned that normal-order reduction or lazy evaluation has desirable properties. This is the case as long as side-effects are not allowed. Side-effects do not fit with lazy evaluation with its unintuitive and data dependent evaluation order. For this reason, functional languages with side-effects (e.g., ML or UFO) use eager evaluation. Then, side-effects of the function arguments will appear prior to that of the function itself.

It has also been said that laziness can conflict with dynamic binding. In order to dynamically bind on an argument it needs to be evaluated anyway without any option of delay [Sargeant95].

4.1.1.4 Encapsulation

Data abstraction, i.e., exposing data constructors to functions of an abstract datatype works well with pattern matching (section 1.2.5 on page 16) but is at odds with encapsulation (section 2.2.2 on page 32). This is not an issue of object encapsulation but has been recognized in the functional programming community as well [Wadler87]. A function that uses pattern matching for a datatype is exposed to its representation — as it accesses its constructors — and is subject to change whenever the datatype has to be changed. There has been a number of proposals how to avoid this drawback, each trying to improve on the weaknesses of the former [Wadler87, Thompson89, Burton & Cameron94, Gostanza et al.96].

4.1.2 Redundancy

It is obvious that opposing concepts exclude their integration. All the same, it is of no use to integrate concepts that are redundant to each other. Concepts must complement each other otherwise a software solution will become an incoherent mix of styles which is hard to understand but does not justify its diversity with corresponding properties.

4.1.2.1 Parameterization

Higher-order functions (see section 1.2.3 on page 12) and inheritance (see section 2.2.3 on page 33) can both be used for behavior parameterization [Kühne95b]. The Template Method pattern uses subclassing to achieve behavior parameterization akin to higher-order functions. Unless the two mechanism do not yield software solutions with different properties and no further complement of one to the other can be found, one should be dismissed.

4.1.2.2 Dispatch

Pattern matching (section 1.2.5 on page 16) and dynamic binding (section 2.2.5 on page 35) can both be used to decompose a function into partial definitions and to select the appropriate portion when executing the function. Both mechanisms divide a function according to the constructors it operates on. While pattern matching keeps the patterns at one place, dynamic binding distributes it to the constructors. It should be clear whether two code selection mechanisms are needed.

4.2 Cohabitance

If we literally agreed to all the arguments made in the previous sections it would be time to abandon any paradigm integration at all. Luckily, many points made above that at first appear excluding are in fact complementing. In the following, I propose how to subsume and integrate functional concepts into those of the object-oriented paradigm. The last section goes beyond feasibility and presents synergistic effects between the two paradigms.

4.2.1 Subsumption

Subsuming functional concepts into object-oriented ones means to find ways to express them without changing an object-oriented language.

4.2.1.1 Pure functions

Even without motivation from functional languages it has been suggested to use side-effect free functions only [Meyer88]. The so-called command-query separation principle prescribes to use side-effects for commands only. For instance the code,

```
a:=io.nextChar;   should be written as   a:=io.lastChar;
b:=io.nextChar;                          io.nextChar;
                                         b:=io.lastChar;
                                         io.nextChar;
```

because the latter version clearly differentiates between reading input (query) and advancing the input stream (command). Thus, it is possible to assign a character from the input stream to two variables without accidently advancing the input stream in between.

Concerning the side-effects of functions one might make a difference between abstract state and concrete state of objects. The latter may change without affecting the former. For instance, a complex number may have two states of representation: A Polar coordinates state for fast multiplication and a Cartesian coordinates state for fast addition. A complex number object may autonomously switch between these concrete states without affecting the abstract state, i.e., its complex number value. Also, some side-effects that do not affect the behavior of a system can be allowed too. The protocoling of executed commands [Dosch95] is a typical example of harmless side-effects.

As a result, the command-query separation principle reconciles state with reduction semantics and also state with lazy evaluation. The reduction semantics of functional languages is embedded into an imperative environment by restricting functions to query functions only. Especially, with the interpretation of objects being sub-states of a system [Kühne96a] the configuration of side-effect free functions operating on system states is close to the design of John Backus' AST[1] system [Backus78] with its *main-computations* between system states.

Furthermore, when functions do not cause state changes and also are not influenced by state changes (see section 4.2.2.2 on the facing page) they may be executed in any order. This opens up the possibility of lazy evaluation again.

4.2.1.2 Dispatch

We have seen that pattern matching is responsible for many subtleties (section 1.3.2 on page 21) and problems (section 4.1.2.2 on the preceding page) without a real impact on software engineering properties (section 1.2.5 on page 16). We therefore decide to abandon pattern matching in favor of dynamic binding which is capable of achieving the same but in addition provides further opportunities (see section 2.2.5 on page 35).

I admit it is unfortunate to loose the nice syntax of pattern matching and to be forced to work with several classes (representing constructors) in order to deal with one function. Notwithstanding, the question of how to present and manipulate a function seems to be more a matter of tools rather than language [Kühne96b]. Although functions are distributed over objects they can be presented to the programmer as a single definition with an appropriate tool (see the epilogue on page 261 for a further discussion).

[1]Applicative-State-Transition

4.2.2 Integration

In contrast to subsumption, integration demands for adaptions of the host language in order to encompass functional concepts.

4.2.2.1 Evaluation

We already regained functions for lazy evaluation (section 4.2.1.1 on page 57) but may also want to have lazy object state semantics too. Otherwise, internal object state might be computed without being requested at all. One approach to achieve lazy effects relies on a refined state monad concept [Launchbury93, Launchbury & Jones94]. Another opportunity is to use models from concurrent object-oriented languages and to propose lazy message mailboxes for objects. In the context of this dissertation we will be satisfied with lazy functions only, though.

A short note is in order regarding the remark that dynamic binding does not cohabit with lazy evaluation (see section 4.1.1.3 on page 56). First, dynamic binding only requires to know the type the receiver which amounts to a partial evaluation only. Second, function application is strict in its first argument anyway. One might evaluate arguments first but eventually the function to be applied must be evaluated to a function abstraction. Therefore, dynamic binding poses no additional requirements that would render lazy evaluation useless.

4.2.2.2 Closures

Section 3.4 on page 49 demonstrated how to use objects as closures in order to implement functions with lexical scoping. Closures capture the values of variables at their declaration and / or application environment as opposed to the point of execution. That is why, one can safely use closures in conjunction with lazy evaluation (see section 4.2.1.1 on page 57). Once applied to values closures do not depend on state changes anymore.

Why do closures need to be integrated rather than subsumed? Most object-oriented languages do not allow free functions[2] and require class definition overhead for closures which especially gets worse when currying should be supported. Moreover, few object-oriented languages (e.g., SMALLTALK and JAVA) provide mechanisms for anonymous closures (e.g., blocks [Goldberg & Robson83] and inner classes [Sun97] respectively).

4.2.3 Synergy

Synergy occurs when two or more concepts complement each other in a way that results in properties that cannot be described by the sum of the separated concepts.

[2] C^{++} being an exception but then its functions are too weak to support closures.

4.2.3.1 Parameterization

Section 4.1.2 on page 57 postulated to either surrender higher-order functions or inheritance unless a good reason not to do so can be found. This section argues that both concepts are necessary and even amplify each other. We will proceed with an example involving a cook who accepts recipes in order to prepare dishes. Written mathematically: $\mathit{Cook} \oplus \mathit{Recipe} \Rightarrow \mathit{Dish}$. We will try several instantiations for the $\oplus$-operator, i.e.,

$\oplus =$ object responsibility

$\oplus =$ single-, multiple-, repeated-inheritance

$\oplus =$ parameterization

We can evaluate the resulting solutions by asking a set of questions:

Category	*Question*
SIMPLICITY	What is the number of classes and the nature of class relations needed?
FLEXIBILITY	How flexible are we in adding new cooks and recipes? Is it possible to adapt cooks?
SCALABILITY	How does the model scale up when we add new cooks and recipes?
SELECTABILITY	What support do we have in selecting cooks and recipes?
REUSABILITY	Can we reuse cooks and recipes for other purposes?
ENCAPSULATION	Which degree of privacy is maintained in a cook and recipe combination?

Table 4.1: Criteria to evaluate cook and recipe combination

We will now try each of the above alternatives for the $\oplus$ combinator. We assess the resulting solutions by referring to the above criteria by a SMALL CAPS typesetting.

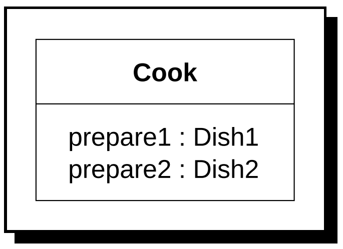


Figure 4.2: Cooking with object responsibility

Object responsibility The first natural approach is to consider recipes as the cook's knowledge and, thus, assign them to a cook as methods (see figure 4.2). This is the most SIMPLE model but it does not offer much FLEXIBILITY: If the Cook does not know how to prepare our favorite dish, we are lost. There is no way to extend the available recipes except by changing cook itself. Subclassing **Cook** for cooks with more recipes is akin to the alternative discussed in the next paragraph. Another drawback is that we cannot REUSE the recipes for other purposes, e.g., nutrition analysis. Finally, we have to SELECT recipes by name (**prepare1**, **prepare2**, ...), i.e., we cannot exploit dynamic binding for recipe selection. Consequently, code that demands a Cook to prepare a dish is sensitive to the addition of new dishes.

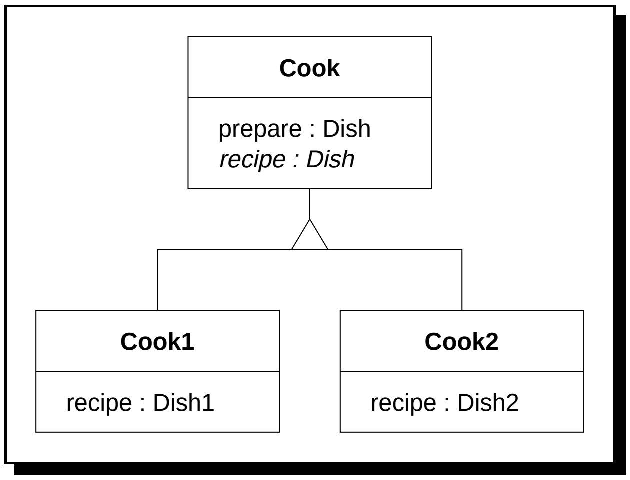


Figure 4.3: Cooking with single inheritance

Single inheritance Creating a cook subclass for each new recipe allows to add recipes without changing existing cooks (see figure 4.3; prepare is a template method using recipe as a hook method, hence this design follows the design pattern Template Method [Gamma et al.94]). However, the price to pay for this FLEXIBILITY is a lot of traffic in the class name space for all these specialized cooks especially when we consider multiple cooks (e.g., French and Italian chefs de cuisine). Nevertheless, the SELECTABILITY problem is coincidentally solved too. Now we can say prepare to **Cook** and depending on the actual subclass (**Cook1** or **Cook2**) we get the corresponding dish. Code can rely on the abstract recipe method and is not invalidated by the addition of new recipes. Unfortunately, we still cannot REUSE recipes for other purposes.

Multiple inheritance The only way to make the recipes reusable is to make them entities (classes) of their own. Using inheritance this leads to multiple inheritance (see figure 4.4). Now we can REUSE recipes, SELECT recipes with dynamic binding, and are FLEXIBLE in adding new recipes. But the model gets more COMPLEX, because of the inheritance relations. SCALABILITY suffers by the fact that each new recipe demands for an additional combinator class. A new cook multiplies the inheritance relationships.

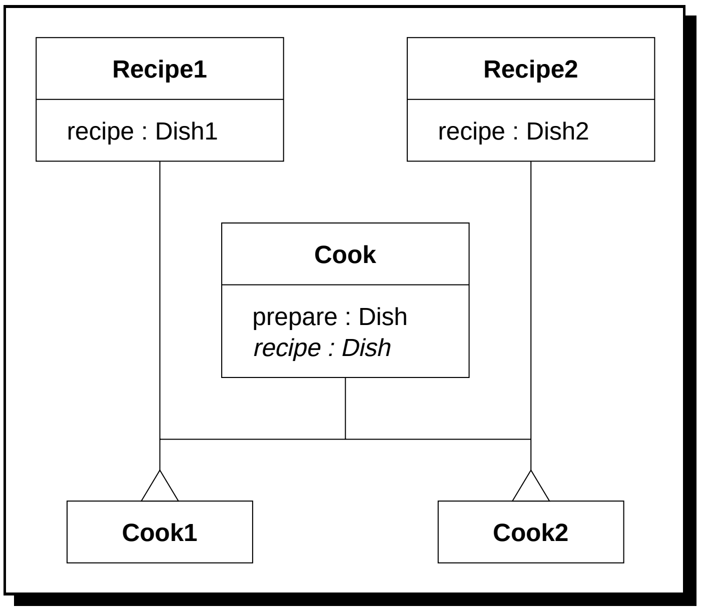


Figure 4.4: Cooking with multiple inheritance

Finally we experience a loss of ENCAPSULATION: Not only cook and recipes break the encapsulation of each other (as already the case with single inheritance) but the combinator classes break the encapsulation of both cook and recipes. A change to the implementation of **Recipe2** may produce a conflict with the implementation of **Cook** (e.g., through the use of a common variable name).

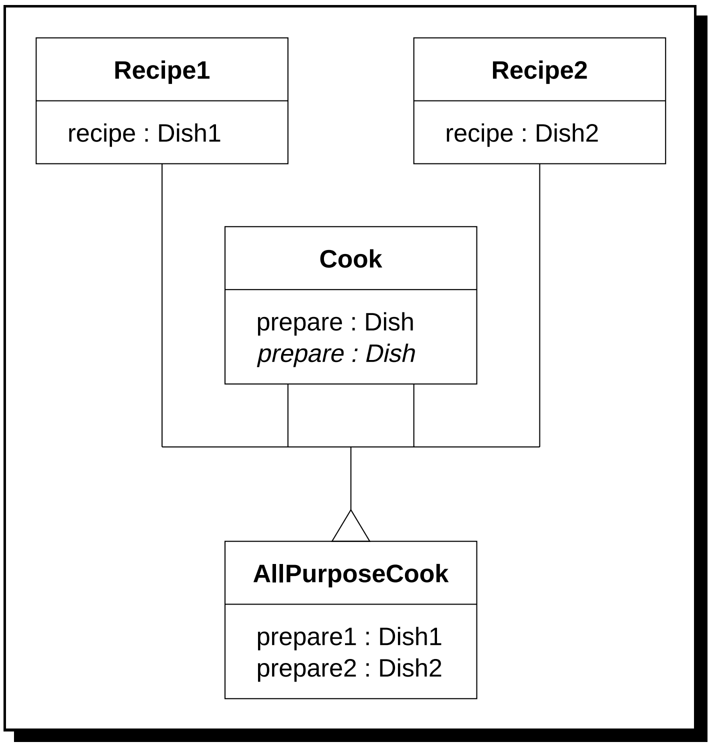


Figure 4.5: Cooking with repeated inheritance

Repeated inheritance The final try involving inheritance aims at saving combinator classes, hence, regaining SIMPLICITY and SCALABILITY. The idea is to fold all combinator classes from the solution above to a single combinator class (see figure 4.5). This combinator class (**AllPurposeCook**) repeatedly inherits **Cook** and distributes each inheritance path and recipe to a unique recipe name [Meyer92]. We gained a little SIMPLICITY but did not really gain on SCALABILITY, since for many recipes we get big combinator classes with a lot of renaming needed inside. Moreover, adding a Cook means adding a new big combinator. As a serious drawback we lost on SELECTABILITY. We cannot exploit late binding for choosing cooks or recipes anymore. Finally, ENCAPSULATION problems represent the worst of all solutions.

Parameterization Clearly, inheritance does not provide a satisfactory answer to the combination of cooks with recipes. Now let us take a fresh look at the problem and regard cooks as a higher-order function parameterized with recipes. Now **Cook** "takes-a[3]" **Recipe** (see figure 4.6 on the next page). This is a client/server relationship, hence, ENCAPSULATION is provided. The **Cook** asks recipes to apply themselves and depending on the actual **Recipe1** the corresponding dish is prepared. Ergo, SELECTABILITY is also enabled. All components are REUSABLE and the model SCALES up without problems. The model also is fairly SIMPLE especially since one can forget about the (abstract) inheritance relationship as a model user. It only needs to be considered when adding new recipes which must simply contain an inherit clause in their class definition.

[3]"uses" or accepts as a parameter.

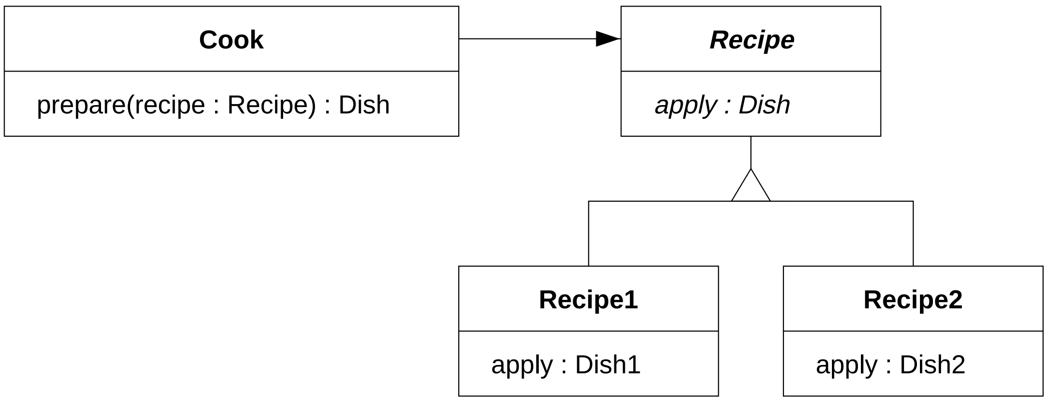


Figure 4.6: Cooking with parameterization

Now is the above model the solution to all problems? Unfortunately not, as it is not as FLEXIBLE as the inheritance models. Why is this so? Remember, the criterion for FLEXIBILITY (see table 4.1 on page 60) also asked for the ADAPTABILITY of cooks: Let us assume we receive an exquisite recipe requiring a special procedure of evaporation. Our standard cook is not capable of that so we need an extended version. The inheritance solutions have no difficulties with that requirement, for they can add the extension in the subclass that combines cook and recipe. In a pure parameterization model a redesign would require to change the parameter of **Cook** to something like **CookAdapter**, provide a dummy adapter for old recipes, and develop a special adapter — containing the cooking extension — for the new recipe.

Parameterization and Inheritance To reconcile FLEXIBILITY with parameterization one can simply combine inheritance and parameterization in their appropriate places (see figure 4.7).

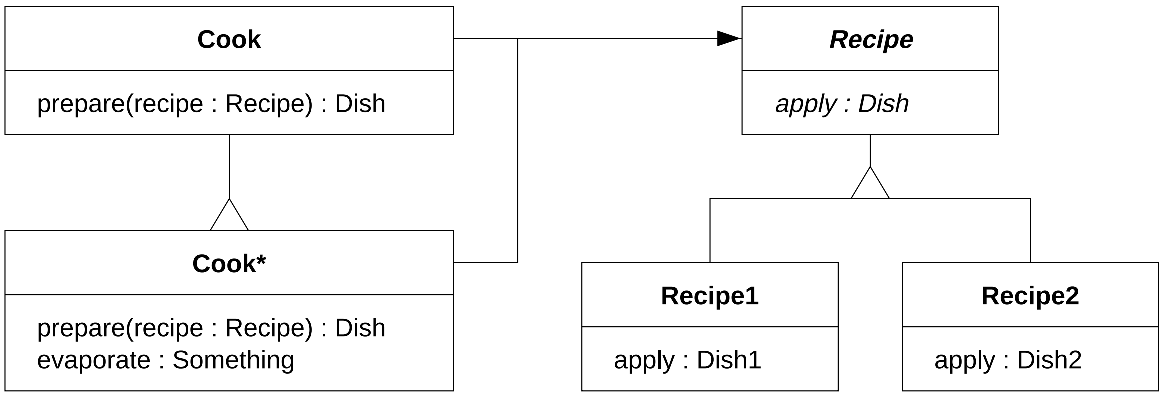


Figure 4.7: Cooking with synergy

In this setup both parameterization and inheritance contribute to a solution that yields optimal criteria evaluation except for a little loss of ENCAPSULATION between the cook classes. Yet, this appears inevitable considering the amount of FLEXIBILITY possible with regard to a cook adaption. Note that using a pure parameterization model it is not an option to simply add a new cook to the system since that would not allow subtyping between cooks, that is, we cannot reuse code that assumes the existence of one cook type only.

What are the lessons to be learned from this excursion? Parameterization is superior to inheritance to model behavior variability. Although, it is not true — as often claimed [Pree94] — that inheritance allows for static combinations only, it involves severe encapsulation problems between the entities to combine. Note that it is easy to apply the arguments for cooks and recipes to other domains. For instance, let cook be an iterator and recipes iteration actions. Indeed, even the repeated inheritance solution has been suggested for combining iterators with several iteration actions [Meyer94b]. Parameterization achieves its complexity and encapsulation advantages through the principle of black-box reuse [Johnson & Foote88, Griss95], that is, to reuse components as is without further modification.

Nonetheless, parameterization alone can reduce flexibility. The strength of inheritance, i.e., to adapt a class in unforeseen ways, is also called "white-box reuse". White-box reuse breaches encapsulation and requires internal knowledge of the reused component but is a powerful supplement to black-box reuse due to its flexibility.

Of course, inheritance has many other useful purposes we did not mention here (see section 2.3.2.3 on page 42) that justifies its coexistence with parameterization. Parameterization in turn has more to offer than demonstrated here. This discussion is continued in chapter 7 on page 93 mentioning, among other issues, why restricted flexibility can also be of advantage.

4.2.3.2 Decomposition

Data abstraction We already mentioned the fundamental dichotomy between a functional and an object-oriented decomposition (see section 4.1.1.2 on page 56). Today there is no doubt about the advantages of structuring a software system according to the business objects it handles. Enormous grow rates for object-oriented systems, languages and publications, give testimony to the historical win of the object-oriented paradigm since the mid eighties over structured design and analysis [Quibeldey-Cirkel94, Shan95]. Apparently, there is no need for functions operating on data, i.e., data abstraction, since everything can be — with advantage — expressed with data annotated with functions, i.e., procedural abstraction.

Unfortunately, it is not as easy like this. The decision for data abstraction of procedural abstraction is an inevitable trade-off. It just happens to be the case that choosing procedural abstraction more often results in better future maintainability. Why is this? Consider table 4.2 on the next page.

The first row contains data constructors while the first column contains operations. The intersection cells contain operation implementations for the correspond-

	Line	*Rectangle*	*Circle*
move(x, y)	move two points	move four points	move center
scale(f)	scale one point	scale one point	scale radius
display	draw line	draw four lines	draw circle

Table 4.2: Decomposition matrix

ing data constructor. If we slice table 4.2 into columns we obtain an object-oriented decomposition. Each constructor packages its operation implementations. If we slice table 4.2 into rows we obtain a functional decomposition. Each function packages the implementations for all constructors.

What happens when we add a new constructor (e.g., *Rhombus*) to table 4.2 representing our our software system? This is the case where functional decomposition requires to modify all functions in order to account for the new constructor (see table 4.3). It does not matter whether we need to add a new function pattern or have to extent a type switch. The crucial point is that all functions must be changed and that they might be spread over the whole software system hiding in unexpected regions. This is the case where the object-oriented decomposition excels. All that needs to be done is to add another object to the system supporting all required operations.

		Adding	
		constructor	function
Decomposition	Functional	open all functions	local function addition
	Object-oriented	local object addition	open all objects

Table 4.3: Sensitivity of decomposition types

Now, what happens when we add a new function (e.g., *rotate(a)*) to table 4.2? The object-oriented decomposition requires to open all objects and add the operation implementation (see table 4.3 on the preceding page). At least, no client code has to be modified. Still, we must perform changes at distributed locations and possibly re-testing (→ regression test) of objects becomes necessary. Furthermore, operations will accumulate at object interfaces which become bloated sooner or later. Is *display* an operation an object should support? Or should *display* be a function of an external view component whose exchange would allow for eased management of changing presentation style? The addition of a function is the case where functional decomposition excels. Only a local addition of a function covering all constructors is necessary.

Two reasons speak for preferring the object-oriented decomposition. First, it is more likely to expect the addition of constructor. A software system can be regarded as a competence provider. Quite often one wants to expand the competence to more data (e.g., handle more graphical objects) rather than expanding the competences (adding yet another user interaction). Second, extending objects with operations (monotonously) does not affect clients. Changing data in a functional decomposition, on the contrary, involves changing client code at many different places.

Nevertheless, there are opportunities where a functional decomposition pays off. For instance, you might want to experiment with language tools working on an abstract syntax representation. Then the language will not change, that is, the constructors are fixed, but often new operations will be defined on the abstract syntax tree. Here, the functional decomposition is better suited as it, in addition, keeps individual operations — which are the focus in this scenario — at a single place rather than distributing them over abstract syntax nodes. This discussion is continued in chapter 12 on page 201 that combines functional decomposition with local semantic functions.

In summary, there are reasons for both object-oriented and functional decomposition. One accounts for data- and the other for operational extensions. It is my understanding of Ivar Jacobson's *control objects* [Jacobson et al.94] that these account for operational extensions. Jacobson also features *entity objects* which could be — using a dogmatic object-oriented view — extended to support any required functionality. However, a division of functionality into intrinsic object properties (e.g., a circle's center and radius) and extrinsic properties (e.g., displaying a circle or calculating π by stochastically dropping a needle onto a circle) results in a system that features *hot spots* for extending both data and operations.

Part III beginning at page 233 will discuss a third option to interpret table 4.2 on the preceding page, elegantly escaping the dichotomy.

Structured programming While it appears worthwhile to retain functions — outside of class definitions — there is further motivation caused by the need to structure functions. It has been argued that functional decomposition techniques are still useful for the decomposition of methods [Henderson & Constantin91]. Furthermore, functional decomposition techniques should also be applied to extract methods from objects. Consider the simulation of planes. During take-off a lot of management is needed to control landing gear, throttle, vertical rudder, etc. This is best managed by a dedicated takeOff function that abstracts from individual actions required. In case we also want to handle helicopters we may simply alter takeOff adding rotor adjustments etc. Other code remains unchanged since it relies on a take-off abstraction. With this view, takeOff is a *business-transaction* [Coplien92] that orchestrates multiple objects to achieve an operational effect. It is not a direct property or capability of an object. It is true that it is perfectly natural to create an abstraction **FlyingObject** that supports takeOff as a method. In this case object and function abstraction just happily coincide. But what about checkPlane? Is it a feature of planes or of service personnel? Is the presentation (view) of an object its responsibility or an external function?

Is it the responsibility of a Tetris piece or the shaft to detect collisions? In the latter example with either choice a representation dependency would be created (see upper half of figure 4.8). Representation changes to the class containing the collision method would affect the method's implementation and interface changes to the other class would require to adapt the former. Ergo, a natural solution would be a free collision function belonging to neither **Piece** nor **Shaft** (see lower half of figure 4.8). In that case changes to piece or shaft representation do not affect the collision function and interface changes are not propagated beyond the collision function. In either case a potential ripple effect of changes through the system is prevented. Object-oriented purists might invent a **Part** object abstraction that would feature a collision method and let **Piece**, **Shaft**, or both inherit from it. Just in order to avoid a free function (**Collision**) the resulting design would posses the natural coincidence of object and function abstraction as in the takeOff case above and, furthermore, would have worse properties than that of figure 4.8. If **Part** is used for other function abstractions and even holds data then a restructuring of one function abstraction affects the others and may require the data hierarchy to be changed. This should not happen and would be the result of forcing function abstractions into a pure object-oriented view.

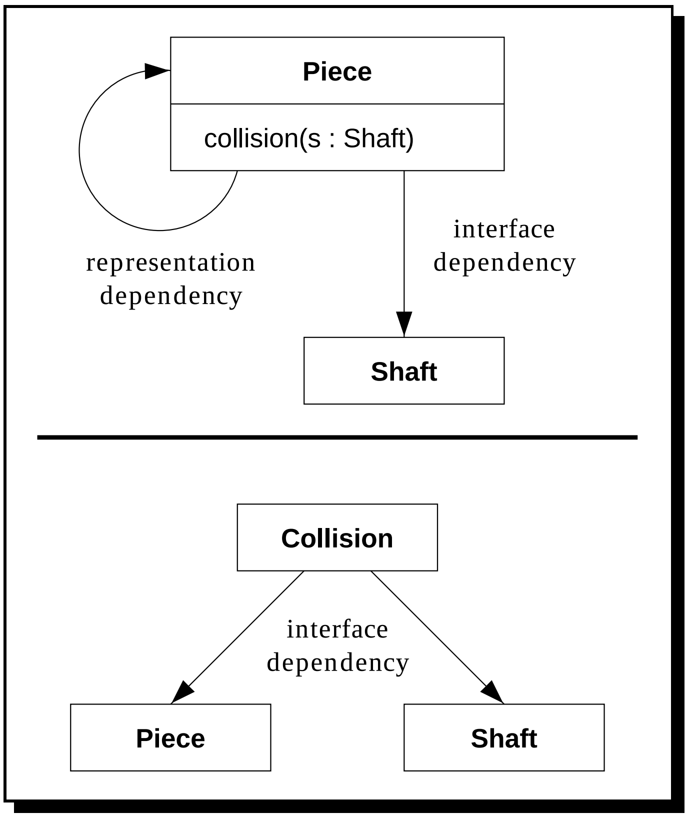


Figure 4.8: Restructured dependencies

4.3 Conclusion

Although a fruitful integration of functional concepts into object-oriented programming appeared to be impossible first (section 4.1 on page 55) I demonstrated the most concepts to be complementary rather then excluding (section 4.2 on page 57).

We retained stateful objects for real world modeling without impedance mismatch, efficiency, and for maintenence reasons.

> *"Pure languages ease change by making manifest the data upon which each operation depends. But, sometimes, a seemingly small change may require a program in a pure language to be extensively restructured, when judicious use of an impure feature may obtain the same effect by altering a mere handful of lines. [Wadler92]"* – Philip Wadler

Nevertheless, the virtues of referential transparency are still available due to the command-query separation principle.

We have seen that a restriction to a pure functional or object-oriented decomposition leads to unsatisfactory results in either case. Both object-oriented or functional decomposition can be appropriate depending whether the objects or the functions (business-transactions) are the more stable concept. After an initial repulsion of any function which is not a object method it is time to re-integrate free functions into object-oriented design again. A move towards this direction are so-called command objects, e.g., used for application callback functions [Meyer88, Gamma et al.94]. Still, these are not acknowledged as functions but as representations of actions to be undone or protocoled [Meyer88].

Regarding the separation between subsumption (section 4.2.1 on page 57) and integration (section 4.2.2 on page 59) it should be noted that the borderline between is somewhat arbitrary. One may also claim that objects subsume functions without further integration needed or that side-effect free functions and value semantics need further integration. With the assessment that stateful programming subsumes functional programming and objects subsume values I meant the opportunity to partly renounce side-effects by programmer discipline. Of course, it is acknowledged that language support for controlling side-effects is highly desirable (see section 14.5 on page 243).

In conclusion, it appears quite reasonable and promising to integrate two seemingly contradicting paradigms. However, someone using such a dual-paradigm approach needs guidance *when* to choose *which* alternative. Figuratively speaking, when to choose the declarative or the procedural branch of figure 4.1 on page 56. It is the intent of the pattern system presented in part II starting at page 85 to provide a base of arguments for a decision.

5 Design Patterns

Once is an event, twice is an incident, thrice it's a pattern.
– Jerry Weinberg

The notion of a software design pattern is not formally defined. More than two years after design patterns have been popularized in 1994 there are still argues about their correct definition and their meaning for software development [Gabriel96]. In order to define the role of design patterns in the context of this dissertation I provide an introductory definition starting with examples from everyday life (section 5.1). After considering the origins of patterns (section 5.2 on page 74) I mention some of the promises to be expected from design patterns (section 5.3 on page 76). This chapter concludes with an explanation of the design pattern format (section 5.4 on page 76) and a description of the diagram notation (section 5.5 on page 79) both to be used in part II beginning at page 85.

5.1 Definition

Patterns are ubiquitous. Most often they are not recognized and less often they are captured by a literate form, but still they exist. This is true of the everyday meaning of the word "pattern", but also, and more importantly, it is also true for a special meaning first described by the architect Christopher Alexander:

> *"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [Alexander79]."*
> – Christopher Alexander

However, it is the nature of patterns to recur, so let us simplify the above to:

> *A pattern is a recurring solution to a problem in a context.*

That is why patterns are ubiquitous. People, but also the mechanisms of nature, repeatedly apply the same solutions to problems that occur in a certain context.

5.1.1 A gentle introduction

Let us assume the context of writing a piece of music for the mass market. We are confronted with conflicting forces:

- *Easy listening.* For quick reception and lasting pleasure we need a catching theme that is often repeated.
- *Boring monotony.* Simple repetition of the same stuff will turn people off and we will loose their interest.

A solution to these forces that has been applied again and again by composers all over the world is to use two basic themes that are repeated in turn and throw in a special theme near the end of the song. If we name the themes A, B, and C, the successful composition pattern is depicted by figure 5.1.

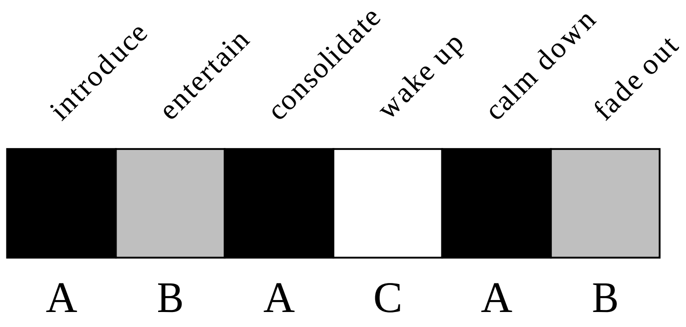


Figure 5.1: A composer's pattern

Using this pattern has several consequences:

- *Familiarity.* The listener is smoothly introduced into our musical ideas and can learn the themes due to their repetition. After the exiting change with special theme "C" we calm him down again.
- *Keep awake.* At the point when the listener might think there is no new stuff to discover we surprise him with the more exiting "C" theme.
- *Regularity.* If we build all our songs according to this knitting pattern the listener will recognize it in the long run and be bored in spite of the surprising variation of repetition.

This description does not aim at completeness. It merely attempts to explain the pattern concept with everyday notions. If we travel a bit further down this road we can see how patterns may collaborate with each other. Let us imagine we composed a song according to the "ABACAB" pattern but aim at a very dramatic coda. We may use another item from a composer's toolbox that can tentatively be called "One Note Up". It is another pattern resolving a different set of forces:

- *Keep attention.* Listeners accustomed to the "ABACAB" pattern may loose their interest towards the end of the song.
- *Do not confuse.* Introducing yet another theme will overstrain the listener.

We resolve the forces by repeating the last theme again but this time with all notes increased by one note. As consequences we have:

- *Cosy feeling.* The listener recognizes the variation to the theme and is not required to learn a new one.

- *Boredom prevented.* The added note creates a dramatic effect that keeps the listener tuned.

Note, how the solution of the first pattern provides the context for the second pattern. This is how true *pattern languages* work. We can think of single patterns as elements of a grammar that, by its production rules, defines a language which tell us how to weave patterns together. In contrast, a set of patterns that provides useful collaborations but does not provide guidance how to choose patterns from a lower level when higher level decisions have been made is called a *pattern system*.

Also, we have seen how the names of patterns like "ABACAB" and "One Note Up" establish a vocabulary to discuss the design of songs. As language can lever our level of thinking we may now design songs on a higher level. In fact, patterns work as chunks and instead of handling only three simple things at once we may deal with the same amount of high level chunks.

Let us recapitulate by assuring that "pattern" is indeed a good name for the above described notion. Table 5.1 features everyday meanings on the left hand side and their interpretation in our context on the right hand side:

Facet[1]	*Meaning*
pattern recognition; inducing attention by repeated occurrence.	Patterns are not invented, they are discovered.
form or model proposed for imitation.	Resolving forces in a successful way is worthwhile to be copied.
knitting pattern; form or style in literary or musical composition.	A pattern description suggests to arrange existing participants in a certain style rather than prescribing the participants.
dress pattern; a construction plan.	Pattern descriptions also tell you how to arrive at the solution described.
configuration; like frost- or a wallpaper patterns.	The participants in a pattern solution form a structure by their relations.
behavior pattern; complex of individual or group characteristics.	Each pattern participant plays a characteristic role in the solution.

Table 5.1: Facets of "pattern"

With regard to patterns we may have several level of consciousness.

1. We are not aware of a recurring solution.
2. We know some “tricks” that jump to our help in some situations, but we cannot fully capture what the real magic is, neither can we determine the character of the applicable situations.
3. We exactly know when we can apply a recurring solution.
4. We know when to apply the solution and are aware of all consequences involved.
5. We are able to linearize our knowledge into a literate form such that others can learn from our experience.

The last level of course refers to design pattern descriptions. Considerable insight is necessary to write a comprehensible pattern description and the result can be considered to be novel work, although the main content on the contrary must be well-known to work. Of course, it is possible to use a pattern description style (see, e.g., section 5.4 on page 76) to document original solutions, i.e., that did not yet recur. It has been suggested to call these “Proto-Patterns”.

5.1.2 Software Patterns

Patterns are useful in the context of software development, since software engineering is not a science yet. There is no straight way of creating an optimal system and probably never will be. This is the case for patterns. They are useful when something creative has to be build and no hard and fast rules apply. Patterns are to be applied by humans not by (today’s) computers.

> *“...software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention Christopher Alexander: Notes on the Synthesis of Form [Alexander64].”*
>
> – Peter Naur

> *“Subsystems created by the composition of objects or interaction machines do not conform to any accepted notion of structure and are very hard to characterize, though they do determine subsystems that exhibit reusable regularities of interface behavior. The term pattern is used to denote reusable regularities of behavior exhibited by interactive subsystems created by the composition of interaction primitives. [Wegner95].”* – Peter Wegner

[1]These are partly taken from a dictionary [Harkavy & et al.94].

“A pattern is a piece of literature that describes a design problem and a general solution for the problem in a particular context.” – James O. Coplien

As yet, no methodology (e.g., [Booch94]) offers a means to describe designs as appropriately as patterns can do[2].

“The problem specification and results for sorting can be formally specified, while the problem class and effects of interactive patterns cannot generally be formalized and require a sometimes complex qualitative description. [Wegner95]”

– Peter Wegner

Patterns tie a design solution to its driving forces and its related consequences. Why is it important to capture the forces too? First, this helps to decide whether a pattern is applicable or not.

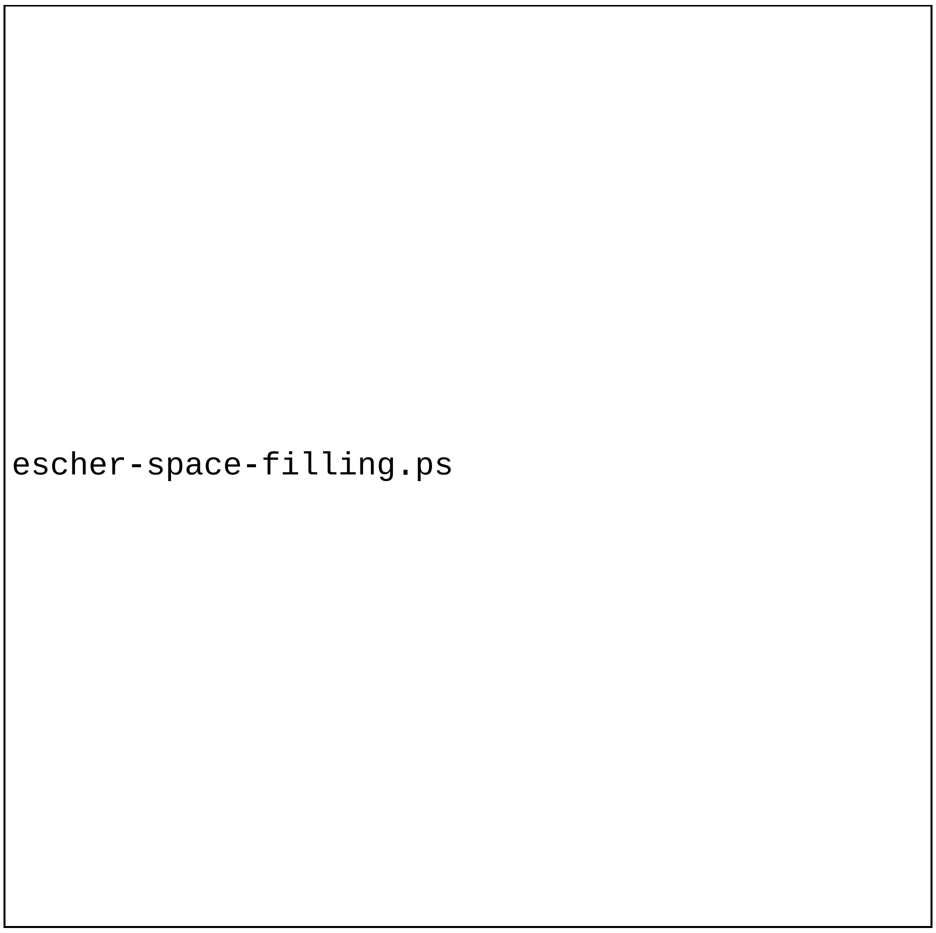


Figure 5.2: Space filling pattern

[2]The upcoming UML Method integrates patterns, though [Rumbaugh et al.97].

Second, when you look, for instance, at a space filling pattern by the Dutch artist M.C. Escher the complexity of its derivation is completely hidden in the beauty of its solution. However, if you try a variation to the basic filling pattern you might be able to fill some space, only to find out it does not work in general. Similarly, it has been observed that month of work flow into the change of a system only to find out that there was a particular good reason for the old solution, which has to be restored then.

In addition to presenting a concrete solution, rather than a set of first principles (like e.g., aim at low coupling and high cohesion), patterns also tie the solution to a concrete example.

> *"If a prior experience is understood only in terms of the generalization or principle behind the case, we don't have as many places to put the new case in memory. We can tell people abstract rules of thumb which we have derived from prior experiences, but it is very difficult for other people to learn from these. We have difficulty remembering such abstractions, but we can more easily remember a good story. Stories give life to past experience. Stories make the events in memory memorable to others and to ourselves. This is one of the reasons why people like to tell stories [Shank90]."* – Roger C. Shank

Design expertise is not a collection of abstract rules, because rules, though useful for evaluations, do not generate solutions. Design experts know many working solutions and know when to apply them:

> *"...wisdom is often ascribed to those who can tell just the right story at the right moment and who often have a large number of stories to tell [Shank90]."*
> – Roger C. Shank

5.2 History

It is hard to determine who described the notion of a pattern first. Because of its fundamental and universal meaning for the human mind it should not come as a surprise to discover works as old as Liu Hsieh's (465–522) "The Literary Mind and the Carving of Dragons" (subtitle: A study of thought and pattern in Chinese literature). He discusses forty nine different patterns in Chinese literary classics in the style of a cookbook [Freddo96].

The first to popularize patterns, however, was Christopher Alexander. He captured successful architectural solutions in pattern descriptions that are supposed to allow even non-architects to create their own individual architectural solutions (e.g., rooms, houses, gardens) [Alexander et al.77, Alexander79]. His patterns form a language, i.e., they provide a top-down guidance from the arrangements of cities down to areas, houses, and the shaping of rooms. Alexander's work also caused the software community to pay attention to patterns. His first influence already took place at the 1968 NATO conference[3] in Garmisch when Naur, referring to Alexander's Ph.D. thesis [Alexander64], mentioned the connection between architecture

[3]Coincidentally, Friedrich L. Bauer coined the term "Software Engineering" at this conference.

and software construction [Naur69] (see section 5.1.2 on page 72). Alexander's thesis is even said to have been the basis for the structured design concepts of coupling and cohesion [Ballard95].

In retrospect, Robert Floyd mentioned pattern-like ideas in his 1978 Turing Award lecture, where he proposed to teach concepts or paradigms like "prompt-read-check-echo" and "generate-filter-accumulate", rather than concrete programming languages [Floyd87].

But officially, it took the software community nineteen (19!) years to rediscover patterns when Ward Cunningham and KentBeck, inspired by Alexander's work, in 1987 decided to let future users of a system design it by supplying them with a small pattern language for developing user interfaces with SMALLTALK. They reported the results at OOPSLA '87 [Meyrowitz87].

Independently, Erich Gamma was dealing with ET++ [Weinand & Gamma94] and realized the importance of recurring design structures in his Ph.D. thesis about ET++ [Gamma91].

Ralph E. Johnson used patterns to document a framework in 1992 [Johnson92] and also James O. Coplien's C^{++} idioms[4] contributed to the pattern literature [Coplien92]. Further publications on patterns followed [Gamma et al.93, Beck & Johnson94]. A landmark was established with the first design pattern catalog book by the GOF[5] in 1994 [Gamma et al.94]. In August 1994, the first conference on patterns were held at the Allerton Park estate near Monticello, Illinois. Many followed since then and are sure to follow in the future.

Curiously, the foundation of object-orientation was perceived by Alan Kay through a typical pattern discovery. First, in 1961 he noticed a particularly clever solution to store and retrieve data for the Burroughs 220 machine [Kay96]. The scheme provided a flexible file format by using three file sections: The first was an indirection table to procedures (residing in the second part) that knew how to read and write the data of arbitrary size and format in the third pard. This scheme corresponds to objects that "know" how to store and restore themselves. The next occurrence for Kay was the Program Reference Table of the Burroughs B5000. It similarly allowed a procedural access to modules. The third encounter in 1966 was caused by the Sketchpad[6] system. It used embedded pointers to procedures to achieve an indirection in calls to procedures. Finally, Kay was asked to fix an ALGOL compiler, which has been doctored to compile SIMULA. When, during his compiler studies, he noticed the similarity between Sketchpad and SIMULA objects and discovered the independent nature and message sending activities of SIMULA instances the "pattern" object-orientation came alive in his mind.

[4]Idioms can be considered to be language-specific patterns that are used to circumvent limitations in — and / or to provide extra functionality beyond — an implementation language.

[5]Gang of Four: Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides; authors of the first design pattern catalog book [Gamma et al.94].

[6]An early interactive computer graphics system.

5.3 Promise

Design patterns recommend themselves by many good reasons (see table 5.2).

Category	*Description*
Reusable Software	Software systems built with patterns are easier to maintain, since patterns, among other aspects, often capture the key solutions that make software extensible.
Reusable Design	Patterns can be used as building blocks for system designs [Beck & Johnson94]. There is no need to invent these micro-architectures again and again.
Documentation	Augmenting a design documentation with pattern names and assigning pattern roles to system components instantly communicates a chosen design decision to a pattern connoisseur[7] [Johnson92].
Communication	Pattern names establish a vocabulary that enables to discuss about designs at a higher level.
Teaching	As recorded experience patterns can be used for teaching successful designs. Patterns are a way to hand down culture from generation to generation [Coplien96b].
Language Design	If a recurring solution can be recognized as a work-around due to an inappropriate implementation language, it should be taken in consideration for the design of new languages [Baumgartner et al.96, Seiter et al.96].

Table 5.2: Promise of design patterns

The last category in table 5.2 is not usually promoted as a benefit of patterns, yet plays an important role for this dissertation: Part III starting at page 233 discusses the impact on language design caused by the pattern system presented in part II beginning at page 85.

5.4 Form

All design pattern descriptions of part II use a slightly modified GOF pattern template [Gamma et al.94]. Besides minor changes to style and the division of the motivation into a problem and solution section I added categories to section "Related Patterns" in order to clarify the nature of each relation. The following template description has been adapted from the original GOF template description:

[7]A scientific proof of this statement has been made with a controlled experiment showing the recognition of patterns to result in faster and better changes to a program [Prechel et al.97].

Figure 5.3: Software system of interwoven patterns.

Pattern Name

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will be used in design discussions.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates —

Problem

— a design problem —

Solution

— and how the class and object structures in the pattern solve the problem.

The scenario will help you understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

- *An applicability bullet.* An applicable situation.

Structure

A diagram showing relationships between participating classes and / or objects.

Participants

The classes and / or objects participating in the design pattern and their responsibilities.

- **Participant Name**
  - Responsibility for what.

Collaborations

An interaction diagram and verbal description explaining how the participants collaborate to carry out their responsibilities.

- Collaboration.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

- *An consequence bullet.* Description of consequence.

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

- *An implementation bullet.* Description of implementation aspect.

Sample Code

Code fragments that illustrate how you might implement the pattern in EIFFEL.

Known Uses

Examples of the pattern found in real systems or languages.

Related Patterns

What design patterns are closely related to this one?

Categorization

Patterns with important similarities and differences.

Collaboration

Patterns that allow useful collaborations.

Implementation

Patterns likely to be used for implementation.

5.5 Diagram notation

Following the GOF pattern template, all interaction-, class- and object diagrams use an extended OMT diagram notation [Rumbaugh91]. In the following figures shadowed ellipses are used to separate notational elements from their description.

5.5.1 Class diagram notation

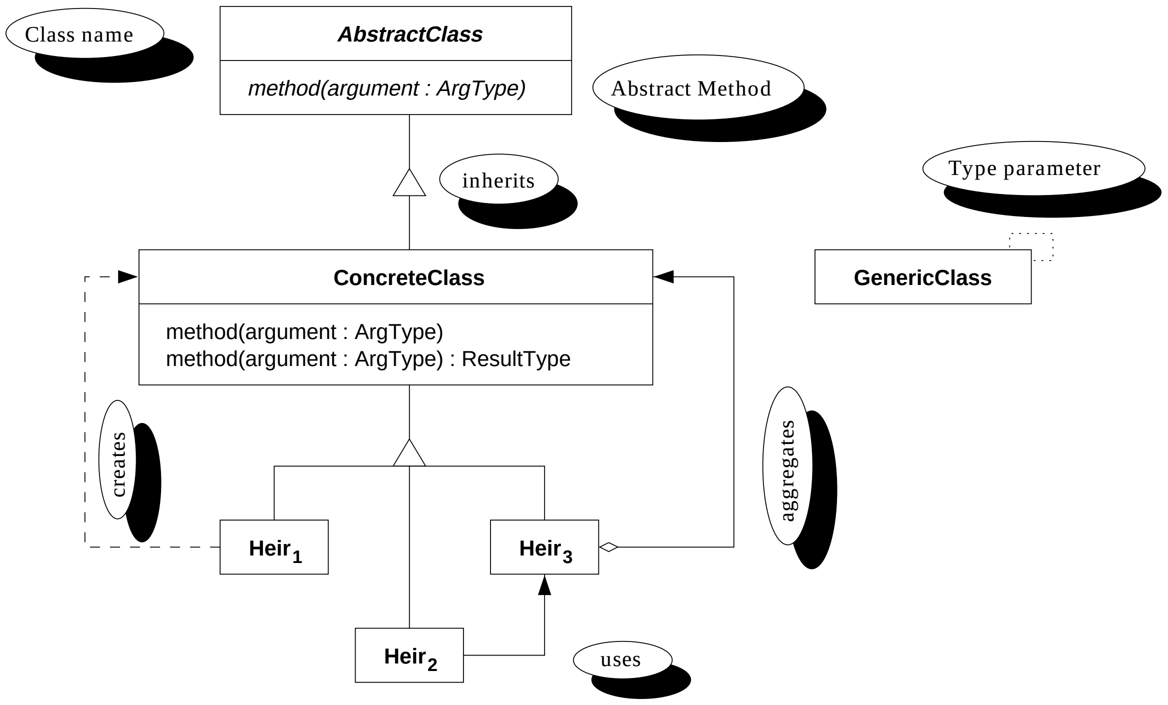


Figure 5.4: OMT Notation: Class diagram

Aggregation, as usual, means that a class has an attribute of the pointed-to class type. The uses relation denotes the pointed-to class to be a server and thus includes aggregation and parameter passing.

5.5.2 Object diagram notation

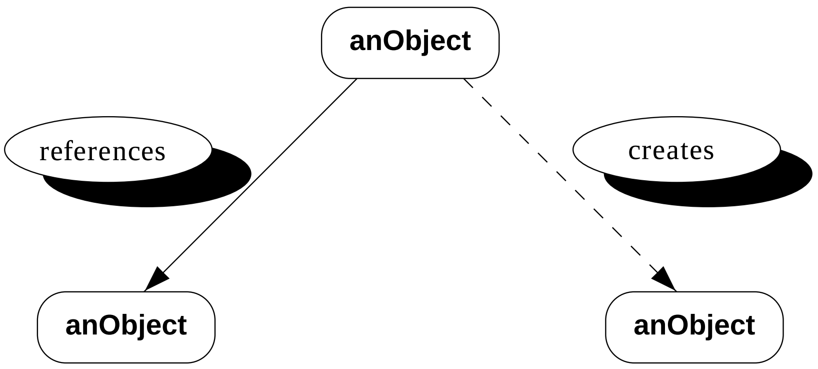


Figure 5.5: OMT Notation: Object diagram

5.5.3 Interaction diagram notation

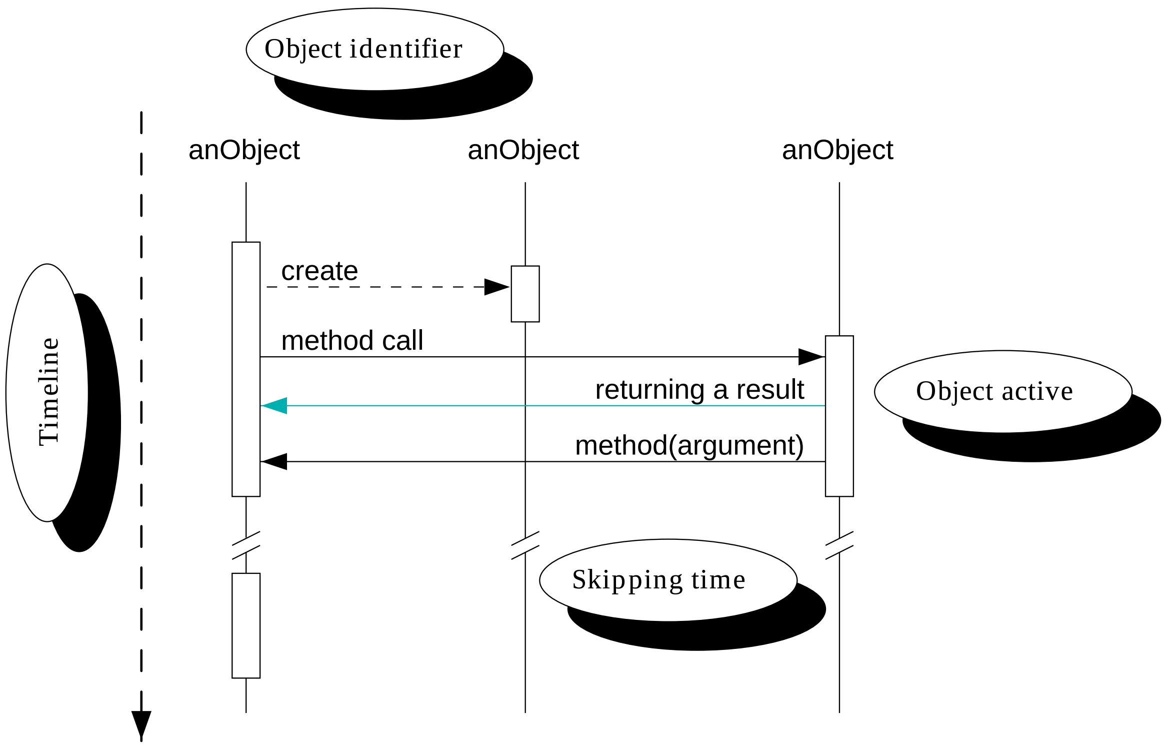


Figure 5.6: OMT Notation: Interaction diagram

Part II

Pattern System

6 Catalog

Design patterns are points in a multidimensional design space
that denote configurations of abstract design rules that actually work.
– me

Chapter 3 on page 45 discussed the compatibility and utility of several functional concepts for object-oriented design. Now we select the concepts to be presented as design patterns. First, I briefly discuss the tension between Patterns and Proto-Patterns. Second, I motivate the particular selection of concepts made and use so-called pattlets to generalize on each resulting pattern's solution or supporting idea. Table 6.3 on page 92 lists all patterns with page index and intent description. Finally, I motivate the use of EIFFEL for sample code. Chapter 13 on page 221 elaborates on the possible interactions between patterns of the system.

6.1 Proto-Patterns

Before I am going to present a system of patterns I should elucidate whether it should better be called a system of Proto-Patterns. Section 5.1.1 on page 69 introduced the term "Proto-Pattern" to denote patterns that lack the empirical evidence of successful application. After all, I aim at enriching object-oriented culture with concepts from a different paradigm. Indeed, for most of the presented patterns and their variants I cannot fulfill the rule of three, that is, name three occurrences in software systems. Does that make the patterns less useful? My answer is no! First, it is problematic to define the quality of something by its quantitative occurrence. According to this measure, FORTRAN and COBOL would probably be the best programming languages. Neither does the widespread use of a design solution imply its quality. Second, the patterns presented are proven designs though not in object-oriented systems. Nevertheless, they record successful designs from functional programming.

Apparently, the rule of three is a good idea to motivate pattern authors to do their homework but should not be taken too literally.

"And occasionally, we do not start from concrete observation at all, but build up the invariant by purely abstract reasoning.

For of course, the discovery of patterns is not always historical. Some of the examples I have given might make it seem as though the only way to find patterns is to get them from observation. This would imply that it was impossible to find patterns which do not already exist in the world already: and would therefore imply a claustrophobic conservatism; since no patterns which do not already exist could ever be discovered. The real situation is quite different. A pattern is a discovery in the sense that it is a discovery of relationship between context, forces and relationships in space, which holds absolutely. This discovery can be made on a purely theoretical level [Alexander79]."

– Christopher Alexander

Anyway, the presented patterns capture the software engineering properties of functional concepts and techniques and introduce object-oriented designers to some remarkable solutions.

6.2 Pattern System

The subsequent sections motivate each pattern (see table 6.3 on page 92) in the context of the discussion in chapter 4 on page 55. The actual patterns are presented in the following chapters using the template description of section 5.4 on page 76. Any collaboration between patterns within the presented system will be covered in chapter 13 on page 221.

Subsequently, I use so-called pattlets to express the general idea, wisdom behind, or observations supporting a pattern. Pattlets recommend themselves to express a practical advise without the overhead of a full blown pattern description. For instance:

Pattlet I

Describe a very general pattern extremely poignant and concise with a pattlet.

◇

A pattlet just very briefly expresses a statement that has a very general applicability and would need specializations at least in the form of multiple examples or even dedicated patterns in order to be a comprehensible pattern description.

6.2.1 Function Object

One of the most important concepts in functional programming are higher-order functions [Hughes87]. Coincidentally, parameterization was shown to cohabit with inheritance even synergistically (see section 4.2.3.1 on page 60):

Pattlet II

For mature domains use black-box reuse.

◇

This statement refers to the fact that object composition, i.e., function parameterization is superior to white-box reuse [Johnson & Foote88, Griss95] but the hot-spots of the domain have to be known. In other words, use white-box reuse until you found out the right spots to parameterize. Then, black-box reuse is more safe, dynamic, and comfortable.

A Function Object [Kühne94, Kühne95b, Kühne95a, Kühne96c, Kühne97] is foremost a method object (akin to the Strategy pattern [Gamma et al.94]) but — as a variant — also subsumes a message object (Command pattern [Gamma et al.94]). A method object "objectifies" [Zimmer94] a method, thus, lifting it to first class status. A message object reifies a message, hence, making it amenable to persistence, logging, undoing, and most importantly allows to postpone it. The latter property is ideal for implementing callbacks [Meyer88, Gamma et al.94].

Beyond this, Function Object adopts the functional technique of currying (see section 1.2.3 on page 12). Hence, it expands on Strategy by capturing real closures and extends Command applications with partial parameterization.

6.2.2 Lazy Object

The second most important concept supporting modularity — next to higher-order functions — from functional programming is lazy evaluation [Hughes87]. It is worthwhile attempting to achieve the same decoupling between generators and consumers, to aim at a similar independence of irrelevant dependencies —

Pattlet III

The early bird gets the worm but the lazy lion gets the flesh.

◇

— and to capture infinite data structures.

Pattlet IV

Capture infinity by creating a cycle.

◇

Lazy Object handles both aspects of call-by-need semantics:

1. Latest possible evaluation and
2. at most once evaluation of expressions.

It also leads to the particular useful notion of lazy streams, which can be used to support iteration and conversion of collections.

6.2.3 Value Object

Section 4.2.1.1 on page 57 argued in favor of pure, i.e., side-effect free functions. A deliberate renunciation of state in programming was chosen to reduce software complexity due to state interactions.

Pattlet V

Expand your freedom by restricting yourself.

◊

Beyond that, we might want to exclude certain procedures from changing otherwise mutable objects. The desirable properties of values from functional programming motivate the need for declaring immutable access paths to objects. Furthermore, some objects should behave like values, that is, should have copy semantics.

6.2.4 Transfold

Whereas in object-oriented programming iterators are very popular (see Iterator pattern [Gamma et al.94]) functional programmers use so-called mappers.

Pattlet VI

For safety and reuse: Don't call us we call you.

◊

Mappers (e.g., *map* and *fold*) accept a function and apply it to all elements of a collection. Transfold, therefore, is an application of the functional principle to use general purpose library functions which can be specialized by using higher-order functionality. Transfold is shown to be specializeable to a plethora of operations.

Pattlet VII

Provide the most powerful concept and specialize it to solve concrete problems.

◊

Although, it is acknowledged that functional mappers are easier and safer to use than conventional iterators, it is commonly assumed that

- they are not applicable in languages without built-in support for closures and
- they are less flexible than iterators.

Transfold invalidates both arguments by generalized folding with function objects.

6.2.5 Void Value

Void Value [Kühne96b] is inspired by pattern matching. Yet, instead of establishing a similar mechanism I observe that pattern matching is simply another form of case statements. One of the fundamentals of object-oriented design, however, is

Pattlet VIII

Replace case statements with dynamic binding.

◊

Although this guideline can also be taken too far (see the Translator pattern on page 201 for a discussion when external dispatching is superior) it is at the heart of object-oriented design to avoid change sensitive client code. Surprisingly, object-oriented languages force their users to write case statements all the time, namely in the form of conditionals testing for object references to be uninitialized (Nil[1]) or initialized (non-Nil) [Kühne96b]. This is caused by the fact that empty or non-initialized data is often represented as a Nil value. Dispatching on Nil does not work since it is not a type but an exceptional reference state. Luckily, another pattlet comes to our rescue:

Pattlet IX

Dispatch on values by making them types.

◇

Void Value shows how to represent datatype constructors as subtypes in order to avoid client code checking for uninitialized data. Coincidentally, the values as types metaphor shows the object-oriented way of pattern matching. Instead of representing, e.g., the empty or non-empty state of a list as the list's state one can define **NilList** and **ConsList** as subclasses of **List** and distribute any function's definition respectively.

Curiously, for Void Value the functional concept was not the prototype, but pointed out a weak and inconsistent spot in the object-oriented framework.

6.2.6 Translator

Translator [Kühne98] uses external functions to avoid element interfaces from becoming bloated and being subject to changes. It, accordingly, prefers a functional over an object-oriented decomposition (see section 4.2.3.2 on page 64). Actually, it employs the Generic Function Object pattern on page 109 to implement free multi-dispatching functions. Translator is inspired by denotational semantics descriptions [Schmidt86] and facilitates incremental evaluation.

Pattlet X

Incremental evaluation calls for homomorphic translations.

◇

Homomorphic translations are well-known in functional programming (e.g., by the so-called fold operators [Bird86, Gibbons & Jones93]) and Translator relies on them for local function descriptions and incremental evaluation.

Translator also uses an intermediate data structure which is in close correspondence to the explicit data structures used in functional programming to facilitate proofs and clarify design [King & Launchbury93]. John Backus' FP language com-

[1]Nil has many different names such as Null, None, and Void.

pletely relied on such intermediate structures which were processed by the available set of fixed functional forms [Backus78].

Summarizing, Translator joins the four aspects of

- homomorphic translations,
- separation of translation and semantics,
- potential incremental translations, and
- external polymorphism.

and thus describes a general approach for homomorphic translations in an object-oriented language.

In conclusion, I certainly do not claim to have presented all patterns existing in functional programming. Functional programming is full of interesting techniques that are worthwhile to be documented and are not presented here. Nevertheless, the patterns chosen for this thesis provide a complete coverage of concepts supported by functional languages. See section 4 on page 55 for the discussion about concepts worthwhile to be covered and reasons for rejecting others.

Most importantly, higher order functions and lazy evaluation are captured by Function Object and Lazy Object. The virtues of immutability are discussed by Value Object. With Transfold we leave the realm of language concepts and enter language usage. Using folds on collections stems from functional programming folklore rather than being a particular language concept[2]. Also, Void Value does not capture pattern matching in full but draws an important lesson from this functional language concept. Finally, Translator builds on Function Object and the idea of Transfold — to replace constructors with functions — to externalize functionality for homomorphic interpretations. While less obvious then folding, recursive interpretations and explicit intermediate data structures are certainly reoccurring concepts in functional programming. Coincidentally, Transfold is an ideal example to showcase the virtues of higher-order functions and lazy evaluation contributing to a design solution that resolves surprisingly many issues at once.

The presented patterns cover a spectrum from functional language concepts to typical uses of functional languages as depicted in table 6.1.

Language concept			⇐ ◊ ⇒		Language usage
Function Object	Lazy Object	Value Object	Transfold	Void Value	Translator
Functional programming			⇐ ◊ ⇒		Object-Orientation

Table 6.1: Functional pattern spectrum

[2]Higher order functions certainly suggest to use internal iteration, though.

6.3 Why Eiffel?

I choose EIFFEL as the language to illustrate pattern issues with sample code, since

- it is well-known,
- uses garbage collection,
- has a clean-syntax,
- and is purely object-oriented.

A language like C^{++} would force the examples to include code for memory management, distracting from the prior goals. EIFFEL's syntax is sufficiently easy to read in order to allow C^{++}- or SMALLTALK programmers to benefit from the examples. BETA has a much less clear syntax and is also already influenced by functional ideas which makes it less clear to demonstrate the emulations needed for pure object-oriented languages.

EIFFEL scores well above a number of other well-known general purposes languages with 21 source statements per function point [Jones96] (see table 6.2).

Language	*Level*	*Average source statements per function point*
ADA 95	6.50	49
CLOS	15.00	21
C^{++}	6.00	53
EIFFEL	15.00	21
HASKELL	8.50	38
JAVA	6.00	53
SMALLTALK	15.00	21

Table 6.2: Language levels

JAVA would have been a very good candidate too. I expect a correction to the language level given in table 6.2. Unfortunately, until today JAVA lacks support for generic datatypes. Any generic treatment of datatypes means to create superfluous subclasses or spurious cast statements. Given this limitation, EIFFEL was the best choice to express pattern goals with the least distraction from language limitations.

Name (page)	*Intent*
Function Object (93)	Encapsulate a function with an object. This is useful for parameterization of algorithms, partial parameterization of functions, delayed calculation, lifting methods to first-class citizens, and for separating functions from data.
Lazy Object (115)	Defer calculations to the latest possible point in time. This is useful for resolving dependencies, calculation complexity reduction, increased modularity and infinite data structures.
Value Object (149)	Use immutable objects with generator operations for protection against side-effects and aliasing.
Transfold (163)	Process the elements of one or more aggregate objects without exposing their representation and without writing explicit loops.
Void Value (191)	Raise Nil to a first-class value. This allows to treat void and non-void data uniformly, is a way to provide default behavior, facilitates the definition of recursive methods, and enables to deal with error situations more gracefully.
Translator (201)	Add semantics to structures with heterogeneous elements without changing the elements. Separate interpretations from each other and use local interpretations that allow for incremental reevaluation.

Table 6.3: Functional pattern system

7 Function Object

The secret of dealing successfully with a child is not to be its parent.
– Mell Lazarus

7.1 Intent

Encapsulate a function with an object. This is useful for parameterization of algorithms, partial parameterization of functions, delayed calculation, lifting methods to first-class citizens, and for separating functions from data.

7.2 Also Known As

Lexical Closure [Breuel88], Functor [Coplien92], Agent [Hillegass93], Agent-Object [Kühne94], Functionoid [Coleman et al.94], Functoid [Läufer95], Function-Object [Stepanov & Lee94, Kühne95b, Kühne97].

7.3 Motivation

Almost any software system makes use of behavior parameterization in one or the other way. Iterators are a good example. The iteration algorithm is constant whereas an iteration action or function varies. A special kind of iteration is the has (member test) function of a collection. Consider a collection of books. Member testing should cover testing for a particular book title, book author, book type, etc.

7.3.1 Problem

We may implement the predicate in the collection's elements. This works nicely if

- the element's natural interface contains the predicate.
- the predicate implementation may vary with the element type, but we do not want to have a selection of several predicates for one element type.

Composing traversal algorithm and predicate as above uses dynamic binding of the element's predicate method. For instance, it is acceptable for a **SortedList** of

Comparable elements to defer the compare predicate to the elements [Meyer94b]. As it is the sole purpose of such a collection to transpose an element property (e.g., ordering relation) to the entire collection, it is convenient that the element type automatically provides a **fixed** predicate. The collection of books, however, should be sortable according to various criteria, like author, title, and date. In order to be able to change the sorting criterion at runtime, we do not want to encumber the traversal algorithm with a switch statement that would select the appropriate book compare method.

One way to achieve this is to use an *external* iterator. Then the varying predicate (compare title, compare author, etc.) can be combined with the traversal algorithm by placing the predicate in an explicit loop that advances the external iterator one by one. As a result, the number of explicit loops corresponds to the number of member test predicates uses.

However, there are good reasons to write such a loop only once (see “Write a Loop Once” [Martin94], the discussion in the Iterator pattern [Gamma et al.94] and chapter 10 on page 163). Consequently, we use an *internal* iterator. Given a predicate, it returns true if any of the books fits the predicate. Here is how the predicate is “given” to the internal iterator conventionally: The actual member test method is a Template Method [Gamma et al.94], which depends on an abstract predicate. The implementation for the abstract predicate, and thus the specific member test operation, is given in descendants [Madsen et al.93, Meyer94b, Martin94]. Selection of the member tests is done by selecting the appropriate descendant. So, traversal algorithm and functions in general are combined through dynamic binding of the abstract function method. Note that this forces us to place the member test method outside the collection of books (e.g., at iteration objects) since we do not want to create book collection subclasses but member test variations only. Further disadvantages aligned with the above application of an object-oriented design, using inheritance and dynamic binding are:

Static combination. All possible combinations of iteration schemes and functions are fixed at compile time. Neither is it possible to create a new function at runtime.

Combinatorial explosion. Sometimes it is useful to select not just one, but a combination of functions or tests and functions. With subclassing, it is not feasible to provide any independent combination, since it leads to an exponentially growing number of subclasses.

Subclass proliferation. Each new function demands a new **Iterator** subclass. The class name space is cluttered by many concrete **Iterator** subclasses. We may use repeated inheritance to combine all functions in one subclass [Meyer92], but this makes things worse. First, it is non-local design to lump all functions in one class. Second, we have to apply heavy renaming for iteration schemes and functions in the subclass; any combination of iteration scheme and function must be given a distinct name. Third, we lose the ability to use dynamic binding for the selection of a function. Since all functions belong to one class, we no longer can use concrete **Iterator** instances to select the actual combination of iteration and function.

Awkward reuse. Reusing the functions for other iteration schemes or different purposes is practically impossible if they are defined in **Iterator** subclasses. The solution is to extract the functions in classes of their own. But now multiple inheritance is necessary in order to inherit from **Iterator and** to inherit from a particular function. At least multiple tests or functions can be "mixed-in", but scope resolution is needed, and each function combination results in a combinator subclass. Note that some languages, such as SMALLTALK and JAVA do not even allow multiple inheritance.

Poor encapsulation. Composing an iteration scheme and a function with inheritance joins the name spaces of both. In fact, the multiple inheritance solution causes iterator, function, and combinator class to share the same name-space. Implementation changes to either of the classes can easily invalidate the other. An interface between super- and subclasses, as the private parts in C^{++} [Ellis & Stroustrup90], alleviates the problem considerably. As an evidence for the relevance of this problem consider the programming of a dialog in JAVA. If you use an is_valid flag to record the validity of the dialog fields you will not get a dialog display although the program compiles fine. It probably takes a while till you find out about a flag is_valid in class **Component** (you happen to indirectly inherit from) which is used for recording whether a window has been displayed already [Maughan96].

Unrestricted flexibility. Creating a designated class for the combination of an iteration scheme and a function opens up the possibility of overriding the iteration scheme for particular actions. Explicitly counting the elements in a collection could be replaced by just returning the value of an attribute count. Unfortunately, this advantage for the designer of a library is a disadvantage for the user of a library. The user may rely on properties of the original iteration scheme. If the iteration function not only counts the elements, but in addition produces some side-effect, the side-effects will not be executed in the optimized version described above. Or, referring to the cook and recipe example of section 4.2.3.1 on page 60, an adapted cook may accompany its cooking with singing which is not always expected and desirable.

Also, it is possible that the optimized version does not fully conform to the original version, not to speak of plain errors. An original robust iteration scheme [Coleman et al.94], that first evaluates all tests and then applies all functions, could be replaced by a standard scheme, that evaluates test and functions on each element separately. The two versions will behave differently when iteration functions side-effect neighboring elements.

Pre- and postconditions [Meyer88] can help to enforce behavioral identity between iteration schemes, but problems like the above are hard to cover and checking contracts at runtime boils down to testing, as opposed to rigorous proofs.

Identity changes. In order to change the iteration function a different iterator instance must be used. While one would seldom need to rely on an unchanging iterator instance, this property is inhibiting in other settings of parameterization.

For instance, it might be mandatory to keep the same instance of a cook, while being able to process different recipes.

7.3.2 Solution

The best way to get rid of the above disadvantages is to objectify predicates with the Function Object pattern. Combining traversal algorithm and function with Function Object works through literally "giving" an objectified function to an internal iterator. In our example, the member test method accepts a test predicate to be used during iteration:

```
has(predicate : Function[Book, Boolean]) : Boolean is
do
  from  i:=1;
  until i>count or Result
  loop
    if predicate @ (books.item(i)) then
      Result:=true;
    end;
    i:=i+1;
  end;
end
```

Here, the collection of books simply consists of an array of books that is traversed with an integer loop. Note how EIFFEL allows using a nice function application syntax for passing the current book to the predicate. The operator "@" is defined as an infix operator in the common interface for all function objects:

```
deferred class Function[In, Out]
feature
  infix  "@" (v : In) : Out is deferred end;
end
```

The @-operator is defined to take a generic argument type **In** and to return a generic result type **Out**. The member test method from above instantiates these to **Book** and **Boolean** respectively.

A predicate to check for a bible instance is:

```
class IsBible
inherit Function[Book, Boolean]
feature
  infix  "@" (b : Book) : Boolean is
  do
    Result:=(b.title.is_equal("Bible"));
  end;
end
```

Now member testing looks like:

```
containsBible:=library.has(isBible));
```

Checking for a book with a certain title can be achieved by passing a predicate that receives the book title through its constructor:

```
!!checkTitle.make("Moby Dick");
library.has(checkTitle);
```

The code for **CheckTitle** is straightforward:

```
class CheckTitle
inherit Function[Book, Boolean]
creation make
feature
  title : String;

  make(b : Book) is
  do
    title:=clone(b.title);
  end;

  infix  "@" (b : Book) : Boolean is
  do
    Result:=(b.title.is_equal(title));
  end;
end
```

Yet, there is another exciting way to achieve the same thing. Suppose the existence of a predicate that compares the titles of two books, e.g., for sorting books with respect to titles. Although the member test method expects a predicate with one book parameter only we can make the two argument compare function fit by passing a book with the title to be looked for in advance:

```
library.has(titleCompare @ mobyBook);
```

Predicate `titleCompare` has two parameters. Passing mobyBook results in a predicate with one parameter that perfectly fits as an argument to the `has` method. Class `titleCompare` is easily implemented like this:

```
class TitleCompare
inherit Function[Book, Function[Book, Boolean]]
feature
  infix  "@" (b : Book) : CheckTitle is
  do
    !!Result.make(b);
  end;
end
```

Thanks to the generic type parameters of function objects the compiler can check for correct function application and will reject wrong uses concerning number or type of arguments.

Finally, we may combine multiple search criteria like title comparison and date checking by composing predicates with a composite function object:

```
library.has(and @ pred1 @ pred2);
```

Function object and takes a variable number of predicates as arguments, applies each of them to the book it receives, and returns true if all predicates hold.

7.4 Applicability

- *Parameterization.* Function objects are a good candidate whenever general behavior can be adapted to special behavior:

  Dynamics. In addition to runtime selection of existing function objects, new function objects can also be created at runtime. A user may dynamically compose a multi-media function object from text-, graphic-, and sound-producing function objects.

  Orthogonality. Having more than one behavior parameter creates the problem of handling all possible combinations of the individual cases. Function objects can freely be mixed without interfering and without combinator classes.

  Reuse. Function objects can be used by any adaptable algorithm that knows their interface. Even if the algorithm was not designed to supply a function with mandatory arguments, it is often possible to supply them to the function in advance. Consider an error reporter, parameterized by output format functions, only intended for generating text messages. We can upgrade the reporter to create a graphical alert-box by passing a function object that already received information about box-size, colors, etc.

  Identity. When the behavior of an object should change while keeping its identity, function objects can be used as behavior parameters to the object. In contrast, encoding behavior in subclasses calls for something like SMALLTALK's "become:" in order to achieve the same effect.

- *Business transactions.* Often the *functions* are the stable concepts of a system and represent good maintainance spots, in order to cope with changing functionality. Instead of being a well-defined operation on one single object, transactions are *"an orchestration of objects working together toward a common goal"* [Coplien92]. When transactions do not naturally fit into existing data abstractions, Function Object can lift them to first-class status while providing a uniform function application interface. Functions may even form a hierarchy, just like algorithms, for classification and reuse purposes [Schmitz92].

- *Monolithic algorithms.* Just like Strategy [Gamma et al.94] Function Object can be used to define localized algorithms for data structures. When a data structure is stable, but the operations on it often change, it is not a good idea to use the standard object-oriented method to distribute the operations over the object types involved. For instance, each change or addition of an algorithm on abstract syntax trees (such as typeCheck, compile) demands a change in all node-object types. In addition, it is not possible to exchange the operations at runtime.

  If we turn the algorithm into a Function Object, we must use a generic Function Object (see section 7.10 on page 107) to dispatch on the node types, but in analogy to the Strategy pattern [Gamma et al.94],

  - the algorithm logic is localized,
  - a function's state can accumulate results, and
  - we can dynamically choose an algorithm.

  See section 7.12 on page 111 for a comparison of Command, State, Strategy, and Function Object.

- *Small interfaces.* When an object potentially supports many extrinsic operations (e.g., CAD-objects may support different viewing methods, cost- and stability calculations, etc.), but its interface preferably should contain the basic, intrinsic functions only (e.g., geometric data), then the functionality can be implemented in function objects that take the object as an argument. The MVC-framework is an example for separating views from data structures though presentation might be considered to be the model's responsibility at first glance.

- *Method simplification.* If a large method, containing many lines of code, cannot be split into smaller, more simple methods, because the code heavily uses temporary variables for communication, then the method can be transformed into a function object. The main transformation is to replace the temporary variables with function object attributes. As a result, the method can be split up into more manageable sub-methods, without passing parameters between inter-method (but still intra-object) invocations, since communication still can take place via function object attributes. The main computation method simply puts the pieces together, itself being as clear as documentation [Beck96].

- *Symmetric functions.* Some functions do not need to privilege (break the encapsulation of) one of their arguments, by making it the receiver of a method. For instance, checking whether a Tetris piece hits something in the shaft can equally be implemented by **Piece** or **Shaft**. We are even forced to introduce a free function if the source code of neither class is available. Likewise, basic types are often not extendible. Note that the free function replaces a mutual dependency between two classes by two classes depending on the free function. If the two classes belong to separate large clusters, we decoupled the clusters with regard to change propagation and recompilation.

- *Delayed calculation.* Function objects postpone the calculation of their result until it is actually needed. When a function object is passed as a parameter but the receiving method does not make use of it, it will not be evaluated. If the result is never needed, this pays off in run time efficiency. Function Object effectively supports lazy evaluation and thus can be used to implement infinite data structures and supports modularization by decoupling data generation from data consumption [Hughes87].

  Trying to achieve this with methods establishes a function object one way or the other. We may delay method invocation by passing the object that supports the method. Then we use the object as a closure with the disadvantage of non-uniform method names. When we allow the client to call the method, although the client might not really need the result, the method must produce an object which stands for the calculation of the result. This is precisely the case of returning a Function Object.

Do not use Function unless you have a concrete reason. There is a time and space penalty in creating and calling a function object, instead of just invoking a method. Also, there is an initial effort to write an extra function class. Functions with many parameters require just as many class definitions in order to exploit partial parameterization. Finally, sometimes *unrestricted flexibility* as mentioned in section 7.3.1 on page 93 is clearly desirable. Functions that vary with the implementation of their argument should be members of this abstraction and not free function objects. Also, see the remarks concerning *flexibility* and *efficiency* in section 7.8 on page 102.

7.5 Structure

See figure 7.1 on the next page.

7.6 Participants

- **Function**
  - declares an interface for function application.
- **ConcreteFunction** (e.g., isBible)
  - implements a function.
  - collects argument values until evaluation.

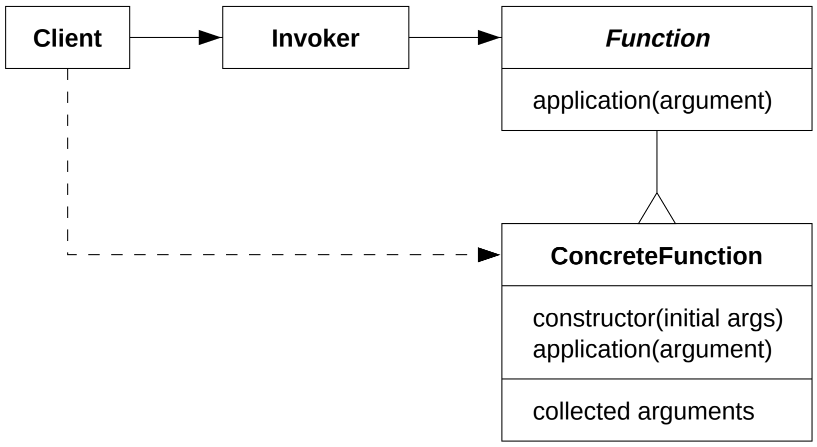


Figure 7.1: Function Object structure diagram

- **Client**
  - creates or uses a **ConcreteFunction**.
  - possibly applies it to arguments.
  - calls **Invoker** with a **ConcreteFunction**.
- **Invoker** (e.g., Iterator)
  - applies **ConcreteFunction** to arguments
  - returns a final result to its **Client**.

7.7 Collaborations

- A client creates a function and possibly supplies arguments.
- An invoker takes the function as a parameter.
- The invoker applies the function to arguments.
- The client receives a result from the invoker.

Client and invoker do not necessarily have to be different objects. A client may evaluate a function object itself if no further arguments are needed or it does not need an invoker to determine the point of evaluation. If an invoker does not supply further argument but uses a dummy argument only to evaluate the function object than we have an application of the Command pattern.

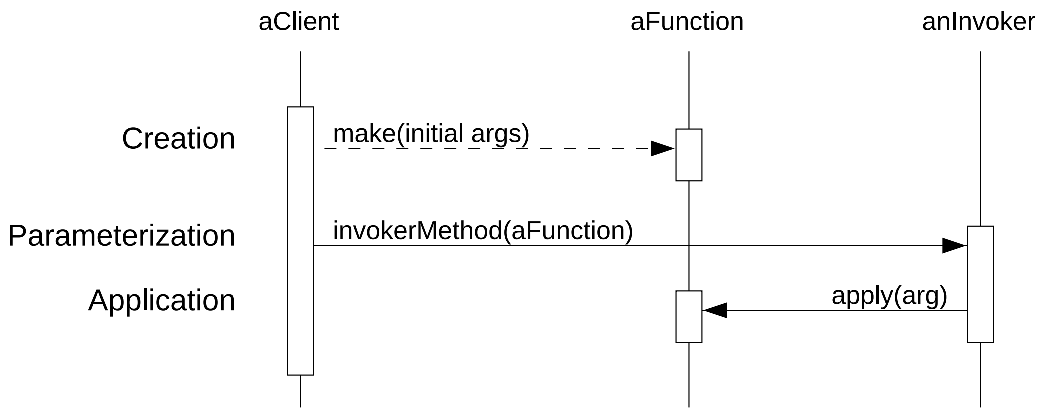


Figure 7.2: Function Object interaction diagram

7.8 Consequences

- *Abstraction.* Function objects abstract from function pointers and in particular from pointers to methods [Coplien92]. Instead of the C^{++} code:

  ```
  aFilter.*(aFilter.current)(t);
  ```

  we can write

  ```
  aFilter(t);
  ```

- *Simplicity.* The use of function objects does not introduce inheritance relationships and does not create spurious combinator classes.

- *Explicitness.* The code `cook.prepare(fish)` is easy to understand. When recipes are wired into **Cook** subclasses, `cook.prepare` depends on the actual **Cook** type. Clever variable names (e.g., `fishCook.prepare`) often are not an option, e.g., `cook.prepare(fish)`, followed by `cook.prepare(desert)`.

  Note that the client must know the function object. A variability on the implementation — rather than the behavior — of **Cook** is better treated with a Bridge [Gamma et al.94]. Clients of **Cook**s should not know about implementation alternatives, unless they explicitly want to decide on time and space behavior.

- *Compositionality.* In analogy to Macro-Command [Gamma et al.94], function objects can be dynamically composed to form a sequence of functions by forwarding intermediate results to subsequent function objects. Composite function objects may also apply several component function objects in parallel. Composite function object variants differ in the way they produce a final output from the single results.

- *Uniform invocation.* Imposing a function object's interface on related operations allows uniformly invoking them. Instead of switching to different method names (like compTitle and compAuthor), we evaluate an abstract function object and rely on dynamic binding [Meyer88, Gamma et al.94]. Consequently, we can add new operations, without changing the caller (e.g., event handler).

  Whenever specific evaluation names (e.g., evaluate, solve, find) are considered important, then they can be provided as aliases.

- *Encapsulation.* As function objects establish client relationships only, they are protected from implementation changes to algorithms that use them. Likewise, the implementation of function objects can change without invalidating the algorithms. Hence, Function Object allows *black-box reuse* [Johnson & Foote88] and helps to advance reuse by inheritance to reuse by composition [Johnson94].

- *Security.* A client of an adaptable algorithm can be sure to use a fixed algorithm semantics. It is impossible to be given an optimized version which does not fully comply to the original semantics (see 7.3.1 on page 93, *Unrestricted flexibility*).

- *Flexibility.*

  + A statement like `iterator.do(f)` is polymorphic in three ways:
    1. **Iterator** may internally reference any data structure that conforms to a particular interface.
    2. The actual instance of **Iterator** determines the iteration strategy (e.g., pre- or post-order traversal on trees).
    3. The actual instance of `f` determines the iteration function.

  – It is not possible to automatically optimize algorithms for specific functions. Nevertheless, one more level of indirection can explicitly construct optimized combinations of functions and algorithms.

  – As discussed in section 4.2.3.1 on page 60 black-box composition does not allow adapting, e.g., cooks for new recipes. Often, it is nonetheless possible to use new recipes with unaltered cooks by partial application of recipes. If a new class of recipes needs the room temperature as an argument (which cooks do not supply) it is possible to curry the recipe with the room temperature in advance. Thus, the new recipe class looks just like the old ones to cooks (see how keyword parameters boost this option in section 7.10 on page 107).

  – When a function has been extracted from a data structure, it is no longer possible to simply redefine it in future derivations. One way to account for this is to make the extracted function a generic Function Object (see section 7.10 on page 107) that will discriminate between data structure variations.

- *Separation.* The ability to carry data (e.g., alert box size) enables function objects to operate on data from outside an (e.g., error-reporter) algorithm as well as (e.g., error text) data from inside the algorithm. The data from outside the algorithm can be local to the algorithm caller. There is no need to communicate via global data. Since function objects can combine local data spaces, they allow decoupling data spaces from each other while still supporting communication.

- *Reuse.* Adaptable algorithms become more reusable because they do not need to know about additional parameters for functions.

  Moreover, function objects are multi-purpose:

  + Functions can easily be used for different iteration strategies. They are are not bound to a particular adaptable algorithm, e.g., comparing book-titles is useful for sorting and for membership testing in collections.

  + One function with n parameters actually represents n functions and one value. The first function has n parameters. The second, created by applying the first to an argument, has $n-1$ parameters, and so on, until the last function is applied to an argument and produces the result[1].

    An example from physics shows the useful functions which can be created from the gravitational force function [Leavens94]:

$$\begin{aligned} GravityLaw\ m_1\ r\ m_2 &= \frac{G\ m_1\ m_2}{r^2} \\ force_{earth} &= GravityLaw\ mass_{earth} \\ force_{surface} &= force_{earth}\ radius_{earth} \\ force_{my} &= force_{surface}\ mass_{my} \end{aligned}$$

- *Iteration.* Function Object suggests the use of internal, rather than external, iteratorsiterator. Internal iterators avoid explicit state and re-occurring explicit control loops. Often external iterators are promoted to be more flexible. It is said to be practically impossible to compare two data structures with an internal iterator [Gamma et al.94]. However, the Transfold pattern on page 163 simply extends an iterator to accept not just one, but n data structures. A transfold-method accesses the first, second, etc., elements of all data structures simultaneously.

  Function Object allows making iteration a method of data structures since it does not demand for subclassing the data structure. This facilitates the use of iterators and allows redefining iteration algorithms for special data structures. Moreover, the data structure (e.g., **Dictionary**) then does not need to export methods (e.g., first, next) in order to allow iterators to access its elements.

[1]Which could again be a function.

- *Efficiency.*

  + A function object may calculate partial results from arguments and pass these to a result function. Hence, the partial result is computed only once, no matter how many times the resulting function object will be applied to different arguments in the future, e.g.,

  $$force_{surface} = force_{earth}\, costlyCalculation$$

  and then

  $$\begin{aligned} force_{my} &= force_{surface}\, mass_{my} \\ force_{your} &= force_{surface}\, mass_{your}. \end{aligned}$$

  – Passing client parameters to function objects can be more inefficient than to, e.g., directly access internal attributes of an iterator superclass. In principle this could be tackled by compiler optimizations [Sestoft88].

  – Care should be taken not to unnecessarily keep references to unevaluated calculations, i.e., function objects. Otherwise, the occupied space cannot be reclaimed.

  – Finally, function objects access the public interface of their servers only. This is why *Symmetric functions* (see section 7.4 on page 98) enforce the encapsulation of their arguments. This represents positive decoupling, but can be more inefficient than unrestricted access. However, selective export (EIFFEL) or *friends* (C^{++}), allow trading in efficiency for safety.

  – It is not possible to apply partial evaluation techniques to closures [Davies & Pfenning96]. Hence, function objects typically cannot be evaluated before runtime. However, this is just a price to pay for flexibility. Dynamic binding introduces runtime overhead exactly for the same reason.

7.9 Implementation

- *Creation.* Quite naturally, function objects must be created before they can be used. A typical code fragment shows the overhead in notation to create the object:

```
  ...
local
  times : Times[Integer];
do
  !!times;

  Result:=times @ 6 @ 7;
end;
```

The fragment coincidentally also illustrates the danger in forgetting the creation instruction, causing a runtime exception. Fortunately, EIFFEL enables the useful idiom of declaring a variable of expanded type, i.e., rendering the creation instruction unnecessary. With —

```
...
local
  times : expanded Times[Integer];
do
  Result:=times @ 6 @ 7;
end;
```

— the declaration already takes care of providing `times` with a value. No sharing of function objects can occur since the first application of `times` creates a new instance anyway (see 7.12.3 on page 113, *Prototype*).

- *Call-by-value.* Function objects must copy their arguments. Otherwise, their behavior will depend on side-effects on their arguments. In general, this would produce unpredictable results. In some cases, however, it may be desirable. The function object then plays the role of a future variable, which is passed to an invoker before all data needed to compute the result is available. Long after the function object has been passed to the invoker, it can be supplied with the necessary data by producing side-effects on arguments.

- *Initial arguments.* Real closures (e.g., SMALLTALK blocks) implicitly bind variables, which are not declared as parameters, in their creation environment. This is not possible with function objects. One way out is to treat initial arguments and standard arguments uniformly through lambda lifting [Field & Harrison88]: Initial arguments are passed as arguments to the function object in its creation environment.

  Alternatively, it is possible to pass initial arguments through the function object constructor (see **CheckTitle** in section 7.3.2 on page 96). This saves the intermediate classes needed to implement the partial application to initial arguments. Of course, both variants do not exclude each other.

  Note, however, that explicit binding is equivalent to implicit binding. This follows from the reverse application of β-reduction. Yet, the implicit variable binding of real closures forces them to use the the same variables names as its creation environment. In contrast, a function object can be used in various environments without the need to make initial argument names match the environment.

- *Delayed calculation.* Commonly a function is evaluated after it has received its last argument. Yet, function object application and evaluation can be separated by corresponding methods for application and evaluation. As a result, the client, triggering evaluation, does not have to know about the last argument. Also, the supplier of the last argument does not need to enforce the calculation of the result, which is crucial for lazy evaluation. In order to enable automatic evaluation on full argument supply while still supporting the above separation, it is possible to evaluate on the last parameter and use a kind of dummy parameter (called unit in ML [Wikström87]).

So function objects that separate their evaluation from the supplement of the last parameter (e.g., Call-back functions) are defined with a dummy parameter, which is supplied by the evaluator.

- *Interface width.* As already mentioned in section 7.8 on page 102 function objects may force their argument classes to provide access to otherwise non-public features. If a particular function object is closely coupled to a specific object type, then declaring the function object as the object's friend, i.e., using selective export, allows efficient access to internal data nevertheless. As a result the object can keep its public interface to other clients to a minimum.

- *Partial parameterization.* Two extremes to implement partial parameterization exist. The less verbose is to always keep the same instance of function object and assign incoming arguments to corresponding internal attributes. This will cause trouble when the same function object is used by several clients. As the application of a function object does not create a new instance, the clients will get confused at the shared state. Furthermore, static typing becomes impossible. If each application of a function object produces a new instance of a different type, then static typing is enabled again and no unwanted sharing of function object state can occur. Unfortunately, this forces us to write at least $n-1$ classes for a function object with n parameters.

  Note, that it is not necessary to provide application methods, that allow passing multiple arguments at once. This would only produce even more implementation overhead and can easily replaced by passing the arguments one by one.

- *Covariance.* In principle, EIFFEL allows using a non-generic function interface since the application method can be covariantly redefined. A general ancestor would have the type **Any** to **Any** and descendents could narrow input and output types. However, we recommend to use a generic interface as presented in order to avoid complications[2] through the combination of polymorphism and covariant redefinition, i.e., *catcalls* [Meyer96].

7.10 Function Object variants

This section shortly touches upon important extensions to Function Object.

- *Keyword Parameters.* In addition to standard application, function objects may provide keyword parameters. Accordingly, parameters can be passed in any

[2]Until today no EIFFEL compiler implements system-level validity checking, which is necessary to catch type errors resulting from mixing argument covariance with polymorphism.

order. This makes sense, in order to create useful abstractions. If the definition of the gravitational force in the example of section 7.8 on page 102 had been

$$GravityLaw\, m_1\, m_2\, r = \frac{G\, m_1\, m_2}{r^2},$$

(note the different order of parameters) we cannot define:

$$force_{surface} = GravityLaw\, mass_{earth}\, radius_{earth}.^3$$

With keyword parameters, notwithstanding, we can write:

```
force:=gravityLaw.m1(earthMass).r(earthRadius);
```

Keyword parameters effectively extend currying to partial application. Beyond the reusability aspect above, however, they also enhance the clarity of programs: Consider the initialization of a node with a label and an identification. The code

```
node.make(a, b);
```

does not tell us what parameter plays which role. The meaning is hidden in the position of the arguments. Surely, we should use more expressive variable names but these should reflect the meaning in their environment, e.g.:

```
node.make(name, counter);
```

In other words, we use a name and a counter to initialize a node but still are confused about the argument meanings. Only the version using keyword parameters —

```
node.label(name).id(counter);
```

— documents the meaning of arguments without compromising the documentation of the environment variable meanings. In another case a node is possibly created using a number string and a random number which should be visible in the initialization statement. Keyword parameters respect that there is both an inner and an outer meaning for parameters and that both meanings are important to document.

- *Default Parameters.* A keyword parameter does not need to be mandatory. Recipes may be designed to work for four persons by default but allow a optional adaption. So, both

```
cook.prepare(dinner);
```

and

[3] Functional programmers know this problem and use the *flip* function to exchange the order of (howbeit only) two arguments.

```
cook.prepare(dinner.people(2));
```

are valid. Default parameters in general have been described as the design pattern Convenience Methods [Hirschfeld96]. Unix shell commands are an example for the application of both default- and keyword parameters.

- *Imperative result.* It is possible to use the internal state of a function object to calculate one or multiple results (add accumulated results to **ConcreteFunction** in the structure diagram of section 7.5 on page 100). For instance, one function object may count and simultaneously sum up the integers in a set during a single traversal. The set iteration client must request the results from the function object through an extended interface (add an arrow getResult from aClient to aFunction in the diagram of section 7.7 on page 101). Note that imperative function objects may produce any result from a standard traversal algorithm. The latter does not need any adaption concerning type or whatsoever.

- *Procedure Object.* If we allow function objects to have side effects on their arguments we arrive at the Command pattern extended with parameters and result value. Procedure Object makes methods amenable to persistent command logging, command histories for undoing, network distribution of commands, etc. Like Command, Procedure Object may feature an undo method, which uses information in the procedure object's state to undo operations [Meyer88, Gamma et al.94]. Note how easy it is to compose a procedure object, to be used as an iteration action, with a function object predicate that acts as a sentinel for the action. As a result, special iterations as "do_if" [Meyer94b] can be replaced with a standard iteration.

- *Multi-dispatch.* Sometimes an operation depends on more than one argument type. For instance, adding two numbers works differently for various pairs of integers, reals, and complex numbers. Simulating multi-dispatch with standard single-dispatch [Ingalls86] results in many additional methods (like addInteger, addReal). The dispatching code is thus distributed over all involved classes. If, as in the above example, the operation must cope with a symmetric type relation (e.g., real+int & int+real), **each** class has to know all other argument types.

  A generic[4] function object removes the dispatching code from the argument types and concentrates it in one place. It uses runtime type identification to select the correct code for a given combination of argument types. As such, it is not simply an *overloaded* Function Object, which would statically resolve the types.

  Note that nested type switches can be avoided with partial parameterization: Upon receipt of an argument, a generic function object uses one type switch

[4]Named after CLOS' [DeMichiel & Gabriel87] generic functions.

statement to create a corresponding new generic function object that will handle the rest of the arguments.

Unfortunately, the necessary switch statements on argument types are sensitive to the introduction of new types[5]. Yet, in the case of single-dispatch simulation, new dispatching methods (e.g., addComplex) are necessary as well.

The goals of the Visitor pattern [Gamma et al.94] can be achieved with a combination of generic Function Object and any iteration mechanism. A generic function object chooses the appropriate code for each combination of operation and element type. Once the generic Function Object has done the dispatch, the exact element type is known and access to the full interface is possible. Between invocations, function objects can hold intermediate results, e.g., variable environments for a type-checking algorithm on abstract syntax nodes.

A generic function object may even be realized as a type dispatcher, parameterized with a set of function objects that actually perform an operation. This allows reuse of the dispatching part for various operations.

Visiting data structures with Function Object is acyclic w.r.t. data dependencies [Martin97] and does not force the visited classes to know about visitors [Nordberg96].

Finally, a generic function object is not restricted to dispatch on types only. It may also take the value of arguments into consideration. Consequently, one may represent musical notes and quarter notes by the same class. The corresponding objects will differ in a value, e.g., of attribute duration. Nevertheless, it is still possible to use a generic function to dispatch on this note representation.

7.11 Known Uses

Apart from the uncountable uses of Function Object in functional programming and SCHEME [Abelson & Sussman87], there are many truly object-oriented uses: SMALLTALK [Goldberg & Robson83] features *blocks* as true closures with implicit binding of free variables. SATHER provides *bound routines* [Omohundro94]. ALGOL 68's thunks for call-by-name parameter passing are also closures but bind free variables dynamically in the calling environment. JAVA 1.1 features "inner classes" which are essentially (anonymous) closures used, e.g., for callbacks [Sun97]. The Eiffel Booch Components uses Function Object for searching, sorting, transforming and filtering of containers [Hillegass93]. The Standard Template Library, which was adopted as part of the standard C^{++} library, uses Function Object to inline operations for arithmetic, logic, and comparison [Stepanov & Lee94].

[5] A more flexible approach is to use dynamically extendible dictionaries that associate types with code.

7.12 Related Patterns

7.12.1 Categorization

Objectifier: A function object, like Objectifier, does not represent a concrete object from the real world [Zimmer94], though one can reasonably take business-transactions for real. Function Object is very similar to Objectifier, in that it objectifies behavior and takes parameters during initialization and call. Per contra, clients "have-an" objectifier, while clients "take-a" function object. The latter is a *uses*, not a *has-a* relationship.

Command: A procedure object which does not take any arguments after creation and produces side-effects only boils down to the Command pattern [Gamma et al.94]. One key aspect of Command is to decouple an invoker from a target object. Function objects typically do not delegate functionality. Rather than delegating behavior to server objects they implement it themselves. So, function objects normally do not work with side-effects, but return their computation as an argument-application result. Nevertheless, function objects also can be used for client/server separation, i.e., as Call-back functions. In addition to Command, invokers are then able to pass additional information to function objects by supplying arguments.

State,
Strategy: Function Object, State, and Strategy [Gamma et al.94] are concerned with encapsulating behavior. A decision between them can be based on concerns such as:

- Who is responsible for changing the variable part of an algorithm?

  The State pattern manages the change of variability autonomously. Function objects are explicitly chosen by the client. Strategies are chosen by the client also, but independently of operation requests. Requesting an operation can rely on a Strategy choice made some time earlier.

- Is it feasible to impose the same interface on all variations?

  If the available Strategies range from simple to complex, the abstract Strategy must support the maximum parameter interface [Gamma et al.94]. Function Object avoids this by partial parameterization.

- Does the combination of common and variable part constitute a useful concept?

  The State pattern conceptually represents a monolithic finite state machine, so the combination of standard- and state-dependent behavior makes sense indeed. Strategies are a *permanent* part of general behavior and thus provide default behavior. Here, the combination acts as a built-in bookkeeping for the selection of the variable part. Function objects, however, take part in the "takes-a" relation. A function object and its receiver are only *temporarily* combined in order to accomplish a task. Function objects, nevertheless, can have a long lifetime, such as being memorized in attributes or representing unevaluated data.

Visitor: Data structures need to know about Visitors because they have to provide an accept method [Gamma et al.94]. Sometimes this is undesirable because of the so-introduced mutual dependency between data structures and Visitors [Martin97]. When the data structure is not available as source code, it is even impossible to add the accept method. A combination of Iterator and generic Function Object (see also the Transfold pattern in chapter 10 on page 163) avoids these drawbacks, while providing the same functionality as Visitor:

- It frees the data structures from needing to provide the operations themselves.
- It differentiates between types in the data structure. The generic function object chooses the appropriate code for each combination of operation and element type.
- It allows heterogeneous interfaces on the data elements. Once the generic function object has done the dispatch, the exact element type is known and access to the full interface is possible.
- It concentrates operations at one place and provides a local state for them. Between invocations, function objects can hold intermediate results, e.g., variable environments for a type-checking algorithm on abstract syntax nodes.

Of course, the type switches are sensitive to the addition of new structure objects. However, if the structure is unstable, Visitor is not recommended either [Gamma et al.94].

Lazy Object: An unevaluated, i.e., still parameter awaiting function object is a lazy object (see chapter 8 on page 115). It represents a suspended calculation and may, therefore, be used for delaying calculations.

7.12.2 Collaboration

Iterator: Function objects allow the use of data from inside (elements) and outside the collection (previous arguments). There is no collaboration between Command and Iterator, since Command does not take arguments.

Adapter: Function Object extends the use of *Parameterized adapters* as described in the implementation section of the Adapter pattern [Gamma et al.94] from SMALLTALK to any object-oriented language.

Chain of Responsibility: Pairs of test- (check responsibility) and action function objects can be put into a Chain of Responsibility in order to separate responsibility checks from the execution of tasks. Function Object allows replacing the inheritance relationship between Links and Handlers [Gamma et al.94] with object-composition.

Observer: Instead of receiving a notification message from a subject an observer may register call-back procedure objects at the subject that will perform necessary updates. This scheme adds another variant to the push- and pull- models of the Observer pattern.

Void Value: A void value (see chapter 11 on page 191) may define the default function to be used for initializing a parameterized structure.

Void Value,
Value Object: A function object does not have to copy void value (see chapter 11 on page 191) or value object (see chapter 9 on page 149) arguments because they ensure immutability anyway.

7.12.3 Implementation

Composite: Standard and composed function objects can be uniformly accessed with the Composite pattern [Gamma et al.94]. A composite function forwards arguments to its component functions. A tuple-Composite applies all functions in parallel to the same argument and thus produces multiple results. Several reduce functions (e.g., and of section 7.3.2 on page 96) may somehow fold all results into one output.

A pipeline-Composite applies each function to the result of its predecessor and thus forms a calculation pipeline.

Prototype: Often it is useful to distribute the accumulated state of a function object to different clients. For instance, a command for deleting text can capture the information whether to ask for confirmation or not. However, when placed on a history list for undoing, different commands must maintain different pointers to the deleted text. Consequently, Prototype can be used to clone pre-configured function objects which should not share their state any further. With this regard, any partially parameterized function object can be interpreted as a prototype for further applications (see example about the costly calculation in section 7.8 on page 102, *Efficiency*).

Chain of Responsibility: Generic Function Object can employ a Chain of Responsibility for argument type discrimination. Chain members check whether they can handle the actual argument type. This enables a highly dynamic exchange of the dispatch strategy.

8 Lazy Object

Ungeduld hat häufig Schuld.
– Wilhelm Busch

8.1 Intent

Defer calculations to the latest possible point in time. This is useful for resolving dependencies, calculation complexity reduction, increased modularity and infinite data structures.

8.2 Also Known As

Stream [Ritchie84, Abelson & Sussman87, Edwards95], Pipes and Filters Architecture [Meunier95b, Buschmann et al.96], Pipeline [Posnak et al.96, Shaw96].

8.3 Motivation

Typically, the specification of a solution to a problem inevitably also makes a statement about the order of solution steps to be applied[1]. In this context, it does not matter whether order is expressed as a sequence of events in time or as a chain of data dependencies. The crucial point is that we welcome to be able to order sub-solutions in order to decompose problems into easier sub-problems. But sometimes overstating the order of events yields complex systems.

8.3.1 Problem

Listening to music at home usually amounts to the problem of converting a pit structure on a flat disc to an alternation in air pressure. In this example we concentrate on the part to be accomplished by the compact disc player (see figure 8.1 on the next page).

We may think of the surface-reading-unit (SRU) as some electronic circuit that controls the spinning speed of the disc, tracks the laser, and produces a digital bit

[1]Note, however, that especially the logical paradigm allows being very unspecific about order.

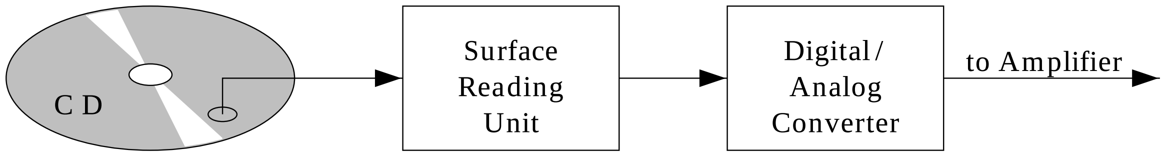


Figure 8.1: Schematic CD-player

stream. These bits are actively fed into the digital-analog-converter (DAC). Let us assume the DAC has a small buffer which it permanently consumes and resorts to during reading problems. Hence, the SRU has to watch the buffer for underflow — causing speed up of reading — and for overflow — resulting in slowing down reading. In this scenario, both SRU and DAC modules are closely coupled. The SRU pushes bit per bit into the DAC but always requests the DAC for its buffer condition. Any alternative SRU to be used in the future must know about the buffer query interface of the DAC module. Vice versa, a future DAC replacement must accept bits through a protocol as specified by the current SRU. This coupling limits the freedom in designing future implementations.

Furthermore, it will not easily be possible to insert bit processing modules inbetween SRU and DAC (see figure 8.2). Provided we still want to keep the bit buffer in the DAC module, the error correction unit (EC) must fake a buffer protocol for the surface reading unit and the DAC buffer condition must be fed back through all components.

8.3.2 Solution

Instead of handshaking the bits through a "push and request" procedure call protocol, we better recognize the data flow architecture of the schematic compact disc player. We should observe that a unit either produces (SRU), transforms (EC and DF), or consumes (DAC) a stream of bits (see figure 8.2).

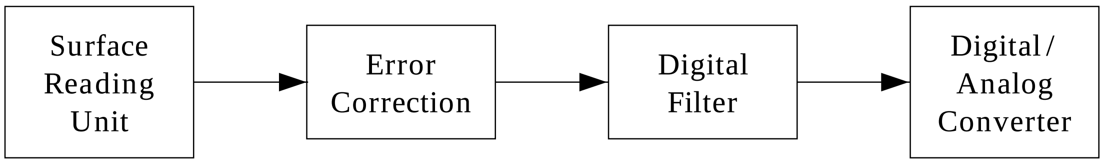


Figure 8.2: Plug-in modules in a bitstream architecture

With this view, it is most natural to let the DAC request further bits from its preceding unit whenever its (self managed) buffer runs out of data. Driven by the demand of the DAC each unit generates bits, possibly by pulling further data from its preceding unit. In this scenario the modules are much less coupled since buffer management is solely accomplished by the DAC itself. The interface between the modules consists of the stable notion of a bitstream. Ergo, alternate implementations or future extensions (like a digital deemphasize unit) can be easily exchanged, shuffled, and inserted. Data sources and transformers are lazy in that they output data only if required by a data sink. Compare this to the model before, where the SRU pushed data to the DAC while managing the DAC's buffer load.

A lazy data source does not need to know about the existence and nature of a receiving unit at all. Moreover, the compact disc's control panel now needs to communicate with the DAC only. Instead of directing a press of the stop button to the SRU it now informs the DAC. Control panel and DAC communicated before also (e.g., for digital volume control), therefore, the cohesion between control panel and DAC increased while the coupling of control panel to other units decreased. With the data driven approach, the architecture can now be divided into layers that depend on each other in one direction only.

We were able to decouple a data source and a data sink since we abandoned the idea of explicitely handshaking each single piece of information and adopted the notion of a stream, representing all information that is still to come at once. John Hughes uses a similar example of separating an algorithm to compute approximations to numerical solutions from the controlling issues like relative or absolute precision reached [Hughes87]. By representing successing approximations as a stream between generator and controlling module he was also able to achieve a higher degree of modularization.

Without lazy evaluation we are often forced to either sacrifice modularity or efficiency. Suppose, we want to feed a server object with an argument whose evaluation requires considerable effort:

```
server.update(costlyCalculation(a, b));
```

The server does not always need the argument to perform its `update` operation. With a lazy calculation everything is fine. With a standard calculation that is always performed, however, we either accept unnecessary calculations or defer the calculation to the server:

```
server.calculateAndUpdate(a, b);
```

While the efficency issue is apparently solved since the server can decide to do the calculation or not, modularity was severely hurt.

1. The `server` now contains a calculation that is probably alien to the abstraction it previously represented.

2. The `server` now depends on two argument types that it did not need to know before.

It is amazing that the initial version above manages to combine both modularity (separation of concerns) and efficiency (avoiding unnecessary calculations).

A lazy attitude is of special importance when a (possibly infinite) stream is generated by referring to itself. A popular example are the so-called hamming numbers. They are defined as the sorted list of all numbers with prime factors two, three, and five [Bird & Wadler88]. A functional program to generate all hamming numbers — as given by definition 8.1 — clearly shows that hamming numbers can be generated by depending on themselves.

$$\begin{aligned}
\mathit{hamming} &= 1 : \mathit{merge}\ (\mathit{map}\ (*\,2)\ \mathit{hamming}) \\
&\quad\ (\mathit{merge}\ (\mathit{map}\ (*\,3)\ \mathit{hamming})\ (\mathit{map}\ (*\,5)\ \mathit{hamming})) \\
\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) &= x : \mathit{merge}\ \mathit{xs}\ \mathit{ys}\ ,\ x = y \\
\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) &= x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys}),\ x < y \\
\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) &= y : \mathit{merge}\ (x : \mathit{xs}\ )\ \mathit{ys}),\ y < x
\end{aligned} \tag{8.1}$$

Figure 8.3 shows the unfolding of function calls into a sequence of numbers.

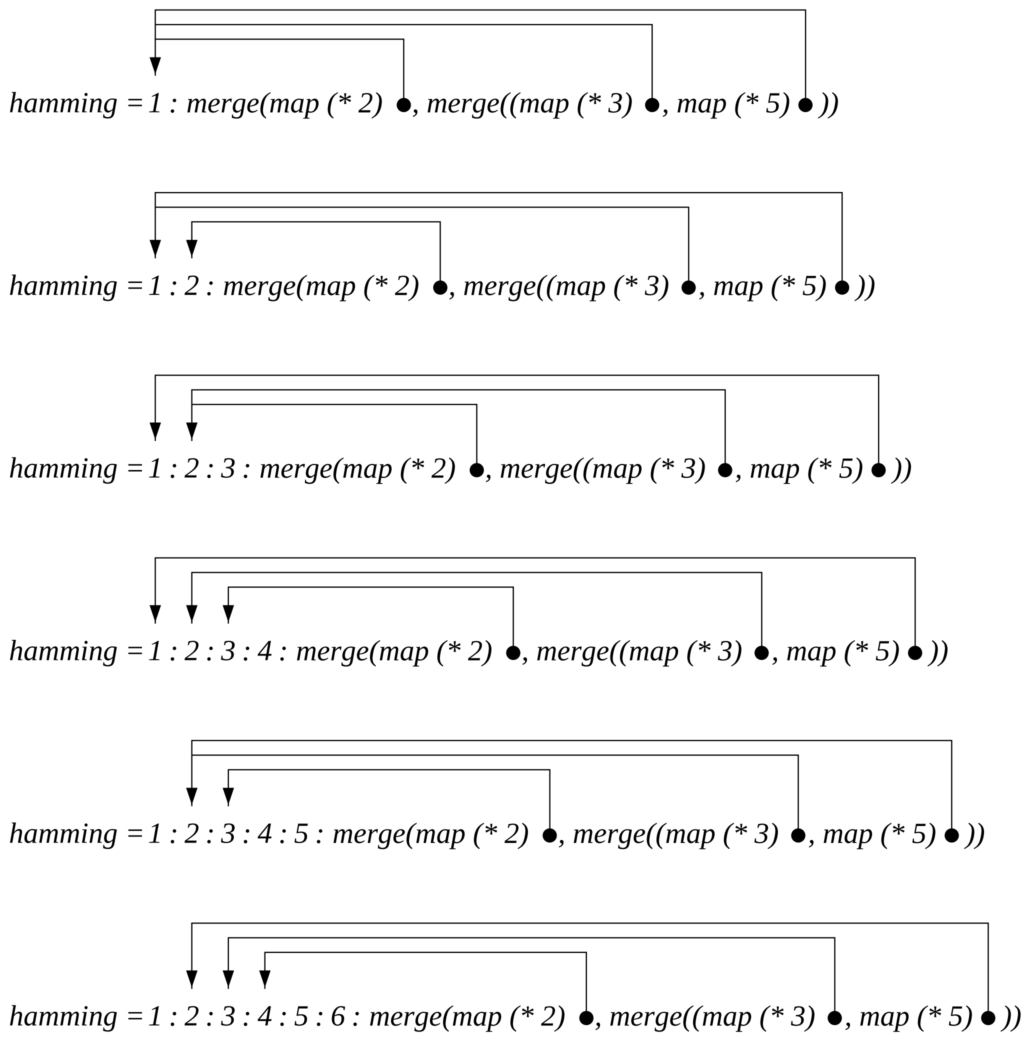

hamming = 1 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))
hamming = 1 : 2 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))
hamming = 1 : 2 : 3 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))
hamming = 1 : 2 : 3 : 4 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))
hamming = 1 : 2 : 3 : 4 : 5 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))
hamming = 1 : 2 : 3 : 4 : 5 : 6 : merge(map (* 2) ●, merge((map (* 3) ●, map (* 5) ●))

Figure 8.3: Generating hamming numbers

Each of the three prime factor multipliers (*map* invocations) maintains its own access point to the already generated list of numbers. Note that the recursive calls of *map* to the remaining hamming numbers must be suspended until a sufficient amount of numbers has been generated. If recursive calls referred to function calls again, rather than access already generated concrete numbers, nothing at all would be generated due to an infinite recursion on function calls. This observation also emphasizes that in order to make evaluation really lazy — as opposed to just non-strict — we also need to cache generated results. Any element should be calculated only once, no matter how often accessed. In an object-oriented implementation this memoization effect is achieved by maintaining caches and / or replacing suspension objects with value objects.

While streams have extensively been used as an example for lazy objects, this should not create the impression that Lazy Object is about streams only. Simply put, streams are just lazy lists. All the same, we may have lazy trees or whatever type of lazy structure. A lazy tree could be the result of a depth first search traversal of a graph. Such a tree would traverse only as much of the graph as needed for the tree part actually accessed [King & Launchbury93]. A lazy object is not even required to represent a collection. Any operation that can reasonably be delayed is a candidate to be become a method of a lazy object.

8.4 Applicability

- *Unknown demands.* Often, the amount of certain data needed is not known before that data is actually accessed. We might, for instance, repeatedly access fibonacci numbers in order to achieve an evenly workload distribution for the parallel evaluation of polynomials. An upper bound is difficult to estimate and we do not want to waste time and space for providing fibonacci numbers that will never be accessed. The solution is to generate them lazily, i.e., calculate them on demand. Note that although we sometimes might be in a position to provide enough space for some data to be calculated in advance, nevertheless infinite data (like fibonacci numbers) would force us to fix an artificial upper bound.

- *Up-front declaration.* The freedom of not needing to care about unnecessary evaluations allows the adoption of a declarative style of programming. It is possible to decompose a routine's arguments or make recursive calls to parts of them without concern whether the results will be needed [Jeschke95]. This style is clearer if the routine needs to multiply access the results. Otherwise, the repeated decomposition code might even require a distributed cache management, if it is not known which branch — of several — will cause the first evaluation and calculations should be made at most once.

- *Dependency specification.* Execution of lazy functions follows dynamic data dependencies. As a result, even statically circular definitions can be specified, without regard for execution order. For instance, a machine code generator needs to determine the destination address for a forward jump. At the time the forward jump is generated, the destination address is not known. A lazy function does not care because it will not evaluate until the destination address is actually inspected (e.g., by accessing the code after its generation). At this time, however, the forward jump can be completed since code generation either already proceeded beyond the destination address or it will be forced to do so by the inspection of the forward address.

Similarly, one may use an implicit evaluation order for the generation of a wavefront in a table. All that is needed are the constant values for the table borders and the main equation, e.g., $a_{ij} = a_{i-1j} + a_{i-1j-1} + a_{ij-1}$ [Hudak89].

- *Partial evaluation.* When a composed operation, such as retrieving all interface information from a programming language library, is evaluated lazily, this may amount to a partial execution of the operation only. When a compiler — for a given source code — does not need to look at the whole library information, because only a part of the library was used, then it will explore only as much interface information as needed [Hudak & Sundaresh88]. Unneeded information will not be retrieved from disc nor extracted from archives.

- *Module decoupling.* When modules should exhibit loose coupling but, nevertheless, an interleaved inter module execution is desirable, then use a stream to connect the modules. For instance, a parser can be connected to a scanner through a stream of tokens. Parser and scanner will alternate their execution, driven by the parser's demands.

- *Co-routines.* Co-routines basically represent suspended behavior. The same effect can be achieved by using a stream generating all co-routine states in succession. As with co-routines, clients do not need to care about state management and behavior scheduling which all happens behind the scenes.

Do not apply Lazy Object, in case of

- *Strict ordering.* If one needs to rely on a fixed evaluation order (e.g., involving side-effects), a lazy evaluation strategy is inappropriate. Lazy evaluations are data driven and may vary their execution order from application to application.

- *Limited memory.* Suspensions of function calls may claim substantial amounts of heap memory. In the unlikely case (since object-oriented programming normally implies heap oriented memory use, anyway) you need an emphasis on a stack based memory allocation, you will not be able to draw from the benefits of Lazy Object.

- *Time constraints.* Lazy evaluation may cause unexpected and unevenly distributed claims of computing resources. If you need to depend on a calculation complexity that is easy to predict and fixed with regard to an algorithm — rather than depending on an algorithm's usage — you should prefer eager evaluation.

8.5 Structure

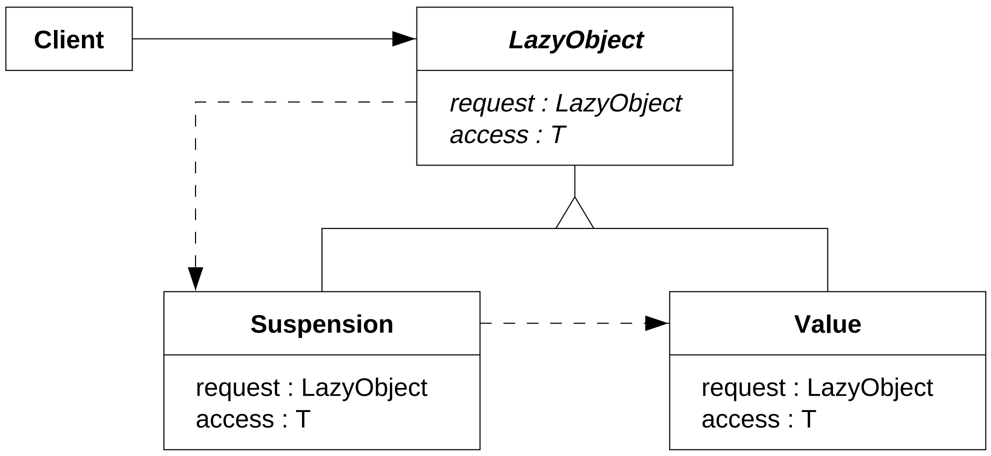


Figure 8.4: Structure diagram

8.6 Participants

- **Client**
  - issues a request to **LazyObject** (e.g., calling `tail` on a stream of hamming numbers).
  - accesses **Suspension** and later **Value** for a value (e.g., calling `item` on a stream of hamming numbers).
- **LazyObject**
  - provides an interface for requests and accesses.
  - creates and returns a **Suspension** on request.
- **Suspension** (e.g., representing a stream's tail)
  - implements **LazyObject**'s interface.
  - represents a suspended calculation.
  - calculates a result when accessed.
  - creates a **Value** for further accesses.
- **Value** (e.g., a cached hamming number)
  - implements **LazyObject**'s interface.
  - caches the result calculated by **Suspension**.

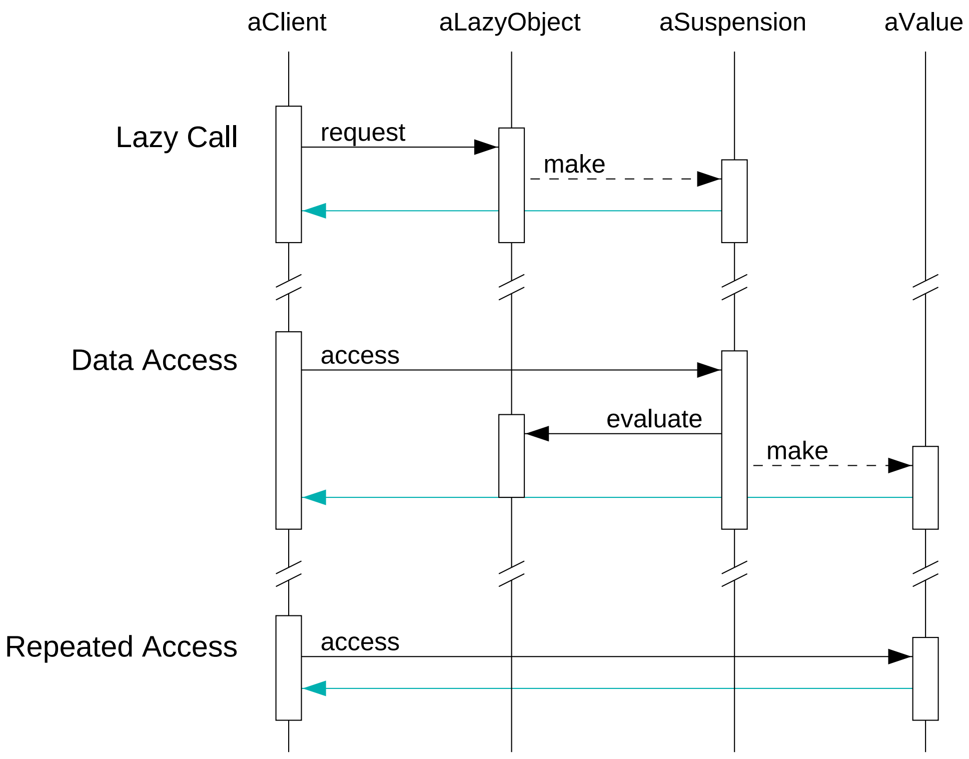


Figure 8.5: Interaction diagram

8.7 Collaborations

- A client issues a request to a lazy object.
- The lazy object creates a suspension and returns it to the client.
- Some time later the client accesses the suspension.
- The suspension calculates a result (possibly accessing the lazy object), returns it to the client, and creates a value for future accesses[2].
- Future client accesses are made to the value that caches the calculation result.

8.8 Consequences

- *Initialization.* The best place for initialization values (e.g., return 1 as the radius of a fresh circle) is the data abstraction they belong to [Auer94]. A getter-routine can calculate an initial value whenever it is called the first time. Hence, initial values will not be calculated unless they are actually needed. Also, setter-routines may provide space for received arguments, only when actually asked to do so.

[2]For instance, a stream element will convert its successor to a value when it is safe to do so.

- *Data generation.* Lazy generation of data structures does not require artificial upper bounds in order to restrict time and space investments. The calculate by need approach allows specifying *all* as the upper bound even in the case of infinite data structures.

- *Data-flow architecture.* Streams emphasize data flow rather than control flow. A system based on streams is determined by the types of modules and the interconnections (streams) between the modules [Manolescu97]. Delayed evaluation and streams nicely match because delayed evaluation inherently provides the intertwined processing of data, required for interconnected streams. Note that connected streams do not necessarily operate in a synchronized fashion. A pipeline with a Huffman encoding[3] inbetween (e.g., to simulate a distorted transmission via a serial connection), will do much more processing at the two ends of the bitstream due to the distribution of one element into several bit elements representing its Huffman encoding. Lazy evaluation, however, will make this inner stream automatically run faster than the outermost ends.

  A chain of components, such as $chain = double \circ filter(< 81) \circ square$, can be assembled without worrying about the size of lists to be processed or specifying loops with exit conditions [Braine95]. One may also reason about such chains and, for instance, simplify the above to $chain = double \circ square \circ filter(< 9)$[4], in order to save operations.

- *Modularity.* The ability to separate what is computed from how much of it is computed is a powerful aid to writing modular programs [Hughes87]. The control over *data consumption* can be separated from the issues involved with *data generation*. A lazy stream allows separating iteration loops (deciding on halting and possibly using other streams) and iterations (including element generation and iteration strategy). See pattern Transfold on page 163.

  Modules can be decoupled but not necessarily implying a loss of efficiency. On the contrary, through partial evaluation, efficiency can actually be gained. Lazily connected modules never cause each other to perform beyond the absolute minimum to produce the results of the driving module.

- *Partial evaluation.* With respect to streams it is worthwhile noting that one may evaluate the spine of stream without causing evaluation of the underlying stream elements. For instance, the call `sieve @ (nats.tail)` does not evaluate the first natural, i.e., “1” since it directly proceeds beyond to the second member “2”. This can be useful when advancing the state is easy but the actual calculation of elements is costly and not all elements are required.

- *Information hiding.* A lazy object hides its delayed internals from clients. A client is never requested to treat lazy object specially, e.g., by passing a

[3]A variable-length code, discovered by David Huffman.

[4]Provided processed numbers are positive only.

dummy parameter for evaluation[5]. Not every lazy computation can be expressed like this, though. In certain cases it might be necessary to use lazy function objects, e.g., as elements of a stream. For instance, laziness may be achieved between a program and an operating system that interchange requests and responses (see figure 1.4 on page 28), only if a response is structured. An atomic response, e.g., an integer, must be encapsulated in a function since it can not be calculated until a request has been processed.

Streams also hide their element generation strategy from clients. They may

1. directly represent stream elements (e.g., a chain of **StreamVal**s).
2. precompute and buffer groups of elements.
3. compute elements on demand.

All these strategies can be mixed in stream combinations. This is possible since through their access restriction — one may only access the head of a stream — two dual views on streams are possible without conflict [Edwards95]. On the one hand, a stream appears as a channel for transmitting data. On the other hand, a stream can be regarded as the entirety of all elements to come.

Summarizing, streams are an excellent example for interface reuse.

- *Value semantics.* Lazy results are actually lazy *values*. Repeated evaluation will not yield different (state dependent) results. Hence, Lazy Object nicely uses state (for memoizing results) in a referentially transparent manner. One may use this kind of *clean state* to model efficient non-single-threaded amortized data structures [Okasaki95c], without the need to resort to side-effects.

- *Iteration.* The close resemblance of a **Stream** interface (see section 8.10 on page 130) to that of an **Iterator** [Gamma et al.94] suggests to use streams as an intermediate language between collections and iterators. Then, streams would provide a linear access to both finite and infinite data collections. Clients, in general, do not have to care whether streams are finite or not with the exception that operations on infinite streams — such as naïvely trying to detect "42" in a list of primes — may not terminate.

- *Inter-collection language.* Actually, streams are a generalization of collection iterators since they can be generated by far more alternatives. Moreover, streams can be used to generate collections. Consequently, streams are an intermediate language for the conversion of data structures into each other (see pattern Transfold on page 163).

- *Persistence.* Streamable collections could also easily made persistent without requiring additional mechanisms, such as Serializer [Riehle et al.97].

[5]As it is the case with the explicit delay styles using `unit` and `delay` of ML and SCHEME.

- *Stream configuration.* The ease of exchanging stream components facilitates end user programming [Manolescu97]. Instead of *recoding* an application by a programmer, such a flexible system allows *reconfiguration* by the user.

- *Parallel execution.*

  + A pipeline of interconnected streams can be processed in parallel since it represents a pure dataflow architecture [Buschmann et al.96].

  − Eager processing can execute operations in parallel to make their results available prior to their usage. Lazy evaluation is directed by data dependencies and, thus, normally does not allow for exploiting peaks of computing resources for future use.

- *Workload distribution.* The increased modularization achieved by streams can be exploited to distribute development efforts nicely into autonomous groups [Manolescu97].

- *Execution profile.* Lazy operations can lead to a peak in computation performance. Insertions to a lazily sorted list nicely require constant time only. Yet, the first reading access will have to resolve all necessary sorting (which may or may not mean to sort the whole list). This behavior — akin to non-incrementally garbage collected applications — should be kept in mind if a more evenly distribution of work is aspired.

  Sometimes, however, it can be desirable to postpone operations. For instance, system start up time should be kept minimal while more elaborate system initialization will happen as required [Wampler94].

- *Time efficiency.*

  + The computational complexity of a lazy algorithm is determined by its usage rather than its intrinsic properties. It is possible to use an $O(n^2)$ sorting algorithm (insertion-sort) to obtain the minimal element from a collection in $O(n)$ time. Access to the first element of the list will not cause the costly sorting of the rest of the list.

    A collection which is traversed with a predicate and a lazy and operator will not be fully visited, if predicate application on one of the elements yields false.

  + Speed is not only to be gained by avoiding unnecessary calculations but also by avoiding to calculate values more than once. For instance, a fibonacci number stream, modeled as the addition of two fibonacci streams ($fibs = 0 : 1 : streamadd\ fibs\ tl(fibs)$), exhibits linear[6] time complexity through the inherent caching of stream values. With this respect,

[6]As opposed to standard exponential time behavior.

Lazy Object is in close relationship to dynamic programming[7]. For example, to calculate all squared natural numbers one can choose an incremental approach by deriving each new value from its predecessor: With $pred = n^2$, it follows that $(n+1)^2 = pred + 2n + 1$, avoiding the costly squaring operation for each element. Speed penalties introduced by closure creations [Allison92] can be more than compensated by this incremental computation option.

+ It is efficient to pass lazy objects (like streams) around since only references will be moved. There is no need to pass lazy objects with copy semantics since they are immutable anyway.

+ It is often possible to avoid repeated traversals of a structure if results are lazily constructed. In order to replace all values in a tree with the minimum value one typically needs to traverse the tree twice. Once, for determining the minimum value, and twice for replacing all values with the minimum. With a lazily computed minimum value, however, one can replace all tree values with a value yet to be computed [Bird84]. The same traversal which is used to replace the values is also used to build a hierarchical function structure of `min` functions over the whole tree. When the tree is accessed afterwards, the minimum function will be evaluated *once* and the obtained value will be used for each tree value accessed.

– When streams are designed to be interchangeable and therefore are required to share a common element format, this may impose conversion overhead. For instance, UNIX commands need to parse ASCII[8] representations of numbers, calculate, and write out an ASCII representation again [Buschmann et al.96]. Object-oriented streams, however, open up the possibility to defer conversions to the elements themselves. Then, special stream element subtypes could avoid any conversions at all.

– Access to lazy values is indirect. Typically, method dispatch redirects calls to either existing values or function calls. Alternatively, flags have to be checked whether cached values already exist.

• *Space efficiency.*

+ Streams can help to restrict the amount of memory needed. Consider a data source that is transformed into another structure that requires exponential space. If this structure is to be processed by a function that extracts a linearly sized result, it is beneficial to model the structure generating process as a stream. Then the extraction function is applied to the stream, resulting in the same linearly sized result, without causing a transient exponential space requirement. Thus, useful intermediate

[7]Also called memoing or tabulation [Abelson & Sussman87].

[8]American Standard Code for Information Interchange.

structures [King & Launchbury93] do not imply infeasible space overhead.

+ Since lazy objects are immutable they can be shared by multiple clients without requiring duplication in order to avoid interferences.

– As can be seen in section 8.10 on page 130 (figures 8.12 on page 138, 8.13 on page 139, 8.14 on page 141 and 8.15 on page 142) the representation of a stream with objects can be storage intensive (see section 8.9 for alleviations). Also, the space required by stream suspensions might be overkill (see next bullet *Closure overhead*). Care should be taken not to unnecessarily keep references to stream suspensions. This would prevent their early removal by garbage collection.

  Although object overhead encumbers garbage collection, clever memory management can make lazy languages outperform eager and even imperative languages [Kozato & Otto93].

- *Closure overhead.* Sometimes, more overhead through closure management is introduced, than is gained by computational savings. Accessing the second element of two lazily added lists $[1,2,3,4]+[5,6,7,8]$, results in, $(1+5):8:([3,4]+[7,8])$, creating two suspensions for one standard addition and a deferred list addition respectively. Especially, when all elements of a structure will eventually be evaluated anyway, it can be worthwhile to avoid the creation of space and time (through creation and management) consuming suspensions.

- *Event handling.* A pull driven stream model as presented here cannot handle prioritized or asynchronous events. It is, however, possible to either use push driven streams [Buschmann et al.96] or to apply an Out-of-band and In-band partition [Manolescu97]. The latter approach divides an interactive application into user interaction (events) and data (stream) processing parts.

8.9 Implementation

On a first reading the reader might want to skip this section. After examination of the sample code in section 8.10 the following implementations details are easier to deal with.

- *Stream functions.* It is very interesting to note that the value, step, and eos interface of **ContFunc** (see section 8.10 on page 130) is justified by a categorical interpretation of streams as coinductive types [Jacobs & Rutten97]. Here, streams over a type A are modeled as a state of type B (state of a **ContFunc** object) and two functions: Function $h : B \to A$ produces a new stream element (value) and function $t : B \to B$ produces a new state (step).

  Then, an anamorphism on h and t (a stream generating function) is defined to be the unique function $f : B \to Str\,A$ such that diagram 8.6 on the next page

$$\begin{tikzcd}
A \times B \arrow[d, "id_A \times f"'] & B \arrow[l, "\langle h{,}t \rangle"'] \arrow[d, "f"] \\
A \times Str\ A & Str\ A \arrow[l, "\langle head{,}\ tail \rangle"]
\end{tikzcd}$$

Figure 8.6: Definition of an Anamorphism

commutes. A stream generator f, hence, needs to build a list cell (eventually **StreamValue**) from calling an element generator h (value) and itself composed with a state transformer t (step): $f = Cons \circ \langle h, f \circ t \rangle$ [Pardo98b].

This view concerns infinite streams only. Introducing finite streams involves consideration of a function that determines the stream's end (eos of **ContFunc** and last of Stream, respectively). For a further discussion the reader is referred to [Meijer et al.91].

- *Avoiding tail suspensions.* Some stream generating functions may create a successing stream state without waiting for arguments to become available. For example, it is not necessary to suspend a tail call to a **StreamFunc** which refers to a **NatFunc** (see section 8.10 on page 130). One can save the creation of a **StreamTail** and its later removal by introducing an optimized stream function (**StreamFuncOpt**) that redefines the tail method of **StreamSuspension** and the forceTail method of **StreamFunc**:

```
class StreamFuncOpt[G]
inherit StreamFunc[G] redefine tail, forceTail end
creation make
feature
  tail : Stream[G] is
  do
    if not tailIsValue then
      if tailCache/=Void then
        if tailCache.isNotSuspended then
          tailCache:=tailCache.toValue;
          tailIsValue:=True;
        end;
      elseif not last then
          tailCache:=forcedTail;
      end;
    end;
```

```
      Result:=tailCache;
    end;

  feature {None}
    forceTail : Stream[G] is
    do
      !StreamFuncOpt[G]!Result.make(contFunc.step);
    end;
  end
```

Instead of creating a stream tail suspension, `tail` directly creates a stream function successor by calling `forceTail`.

A client can make advantage of this optimized scheme by declaring, e.g.,

```
    nats : StreamFuncOpt[Integer];
```

Creation and usage of the optimized stream is exactly as before.

Clients can be made unaware of the existence of two different **StreamFunc** by either going via an application syntax function like **Pipe** (see section 8.10 on the next page) or deferring creation of stream functions to continuation functions.

- *Class versus object adapter.* In order to improve clarity of presentation I used the object version of Adapter [Gamma et al.94] for class **StreamFunc** (see section 8.10 on the following page). What it essentially does is to translate a service from continuation functions (value, step, eos) to the requirements of **StreamSuspension** (forceItem, forceTail, last).

  Alternatively, one can use the class adaptor version for **StreamFunc**. There would be several stream function heirs to **StreamSuspension** that would also inherit from an individual continuation function respectively. On the one hand this would less cleanly separate stream functionality (e.g., forcing) from continuation functionality (e.g., argument processing). On the other hand, a considerable amount of object creation and storage would be saved. In figure 8.15 on page 142 all pairs of **StreamFunc** and **ContFunc** objects would collapse into single objects.

  On may even think about dropping any adaption at all and make continuation functions direct heirs to **StreamSuspension**. Yet, this would burden implementors of new continuation functions to create objects of their respective functions in the step routine. Either class or object adaption scheme nicely takes care of this.

- *Storage requirements.* Also for reasons of presentation clarity I choose a direct modeling of streams with objects. In cases where the space requirements of the object model presented in section 8.10 on the following page are prohibitive, one can evade to more compact representations. Streams can, for instance, maintain a shared generic array of elements and use an internal cursor for their actual position. Access suspensions could then be much more efficiently be represented as an internal counter that increases with tail accesses and has to be decreased — inducing function evaluation — before element access.

8.10 Sample Code

We will now consider the implementation of lazy streams in detail. Again, note that a stream is just an example for Lazy Object and that there are many other applications for Lazy Object besides streams (see section 8.4 on page 119).

The most important issue to establish upfront is the stream interface[9] to clients:

```
deferred class Stream[G]

feature
  item : G is deferred end
  tail : Stream[G] is deferred end
  last : Boolean is deferred end

feature {Stream}
  isNotSuspended : Boolean is deferred end
  toValue : StreamVal[G] is deferred end

feature {StreamTail}
  forcedTail : Stream[G] is deferred end
end
```

Class **Stream** plays the role of **LazyObject** (see figure 8.4 on page 121) with the exception that it defers creation of suspensions to its heirs. For now, we do not care about the non-public features of **Stream** besides noting that `toValue` allows converting a stream element, which no longer needs to be suspended, into a value (see class **Value** in figure 8.4 on page 121).

We start simple by trying to establish a stream of natural numbers. In this case our **Suspension** (see figure 8.4 on page 121) will be a function (represented by class **NatFunc**) that yields the next natural number. Heading for a general scheme we introduce an indirection with **StreamFunc** that uses any **ContFunc** to generate a stream element (see figure 8.7) and make **NatFunc** an heir to **ContFunc** (see figure 8.8 on the facing page).

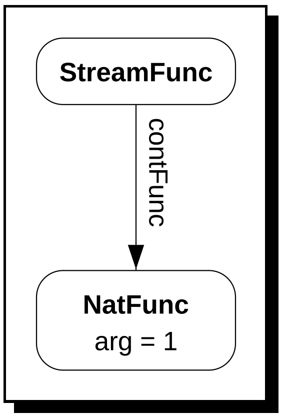


Figure 8.7: Natural number stream

[9]For brevity I omitted an `out` feature that is useful to display streams to the user.

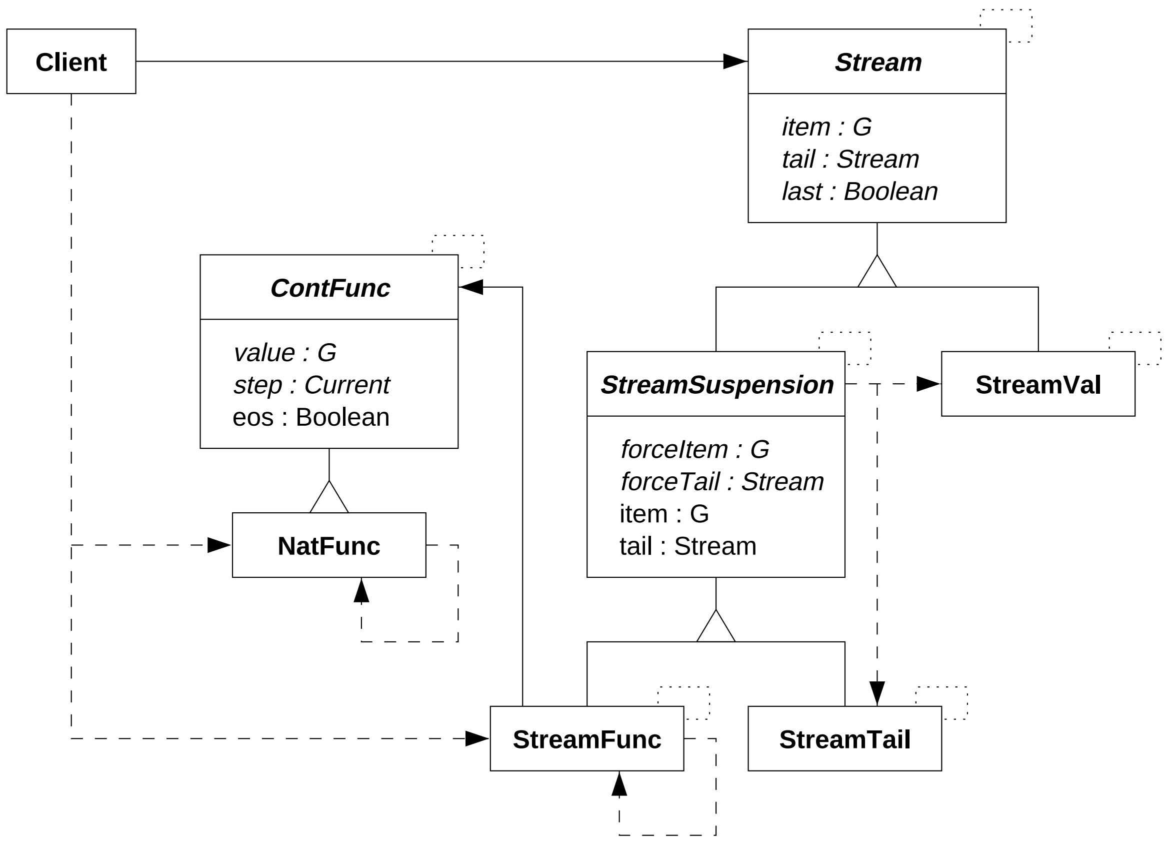


Figure 8.8: Using streams

A client then creates a **StreamFunc** by passing a **NatFunc** for initialization. The client can now access stream elements via the **Stream** interface. Requesting an item from **Stream** is — in this case — handled by **StreamFunc**. **StreamFunc** inherits the implementation of item from **StreamSuspension**:

```
deferred class StreamSuspension[G]
inherit Stream[G]

feature
  item : G is
  do
    if itemCacheFlag=False then
      itemCache:=forceItem;
      itemCacheFlag:=True;
    end;

    Result:=itemCache;
  end;
```

```
    tail : Stream[G] is
    do
      if not tailIsValue then
        if tailCache/=Void then
          if tailCache.isNotSuspended then
            tailCache:=tailCache.toValue;
            tailIsValue:=True;
          end;
        elseif not last then
            !StreamTail[G]!tailCache.make(Current);
        end;
      end;

      Result:=tailCache;
    end;

    last : Boolean is deferred end



  feature {Stream}
    isNotSuspended : Boolean;

    toValue : StreamVal[G] is
    do
      If valueCache=Void then
        !StreamVal[G]!valueCache.make(item, tail);
      end;

      Result:=valueCache;
    end;



  feature {StreamTail}
    forcedTail : Stream[G] is
    do
      if not isNotSuspended then
        tailCache:=forceTail;
        isNotSuspended:=True;
      end

      Result:=tailCache;
    end;
```

```
feature {None}
  itemCache      : G;
  itemCacheFlag  : Boolean;

  tailCache      : Stream[G];
  valueCache     : StreamVal[G];
  tailIsValue    : Boolean;

  forceItem      : G is deferred end
  forceTail      : Stream[G] is deferred end
end
```

Method `forceItem` (called by `item`) is implemented by **StreamFunc**:

```
class StreamFunc[G]
inherit StreamSuspension[G]
creation make

feature
  make(f : like contFunc) is
  do
    contFunc:=f;
  end;



feature {None}
  contFunc : ContFunc[G];

  forceItem : G is
  do
    Result:=contFunc.value;
  end;

  forceTail : Stream[G] is
  do
    !StreamFunc[G]!Result.make(contFunc.step);
  end;

  last : Boolean is
  do
    Result:=contFunc.eos;
  end;
end
```

In our example the invocation of `value` on `contFunc` (see implementation of `forceItem`) will return the current natural number:

```
class NatFunc
inherit ContFunc[Integer]
creation make

feature
  arg : Integer;

  make (a : like arg) is
  do
    arg:=a;
  end;

  value : like arg is
  do
    Result:=arg;
  end;

  step : like Current is
  do
    !!Result.make(arg+1);
  end;
end
```

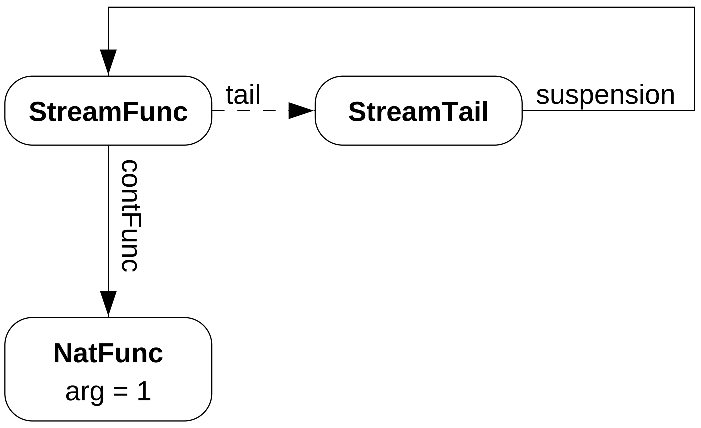


Figure 8.9: Suspension of a stream tail

Requesting the tail of a **Stream** usually results in the creation of a **StreamTail** (see figure 8.8 on page 131 and figure 8.9). An item call on a **StreamTail** object forces the generation of a tail from suspension and delegates the item request to the result of this operation. Again, the implementation of item in **StreamSuspension** will be called and — this time — will use forceItem from class **StreamTail**:

```
class StreamTail[G]
inherit StreamSuspension[G]
creation make

feature
  make(s : like suspension) is
  do
    suspension:=s;
  end;


feature {None}
  suspension : Stream[G];

  forceItem : G is
  do
    Result:=suspension.forcedTail.item;
  end;

  forceTail : Stream[G] is
  do
    Result:=suspension.forcedTail.forcedTail
  end;

  last : Boolean is
  do
    Result:=suspension.forcedTail.last;
  end;
end
```

Delaying the computation of stream elements with **StreamTail** is crucial for self referring streams such as the hamming numbers stream. Not before the hamming number represented by the head of the suspension is accessed is it safe to access any numbers immediately required by the tail of the suspension (see section 8.3.2 on page 116).

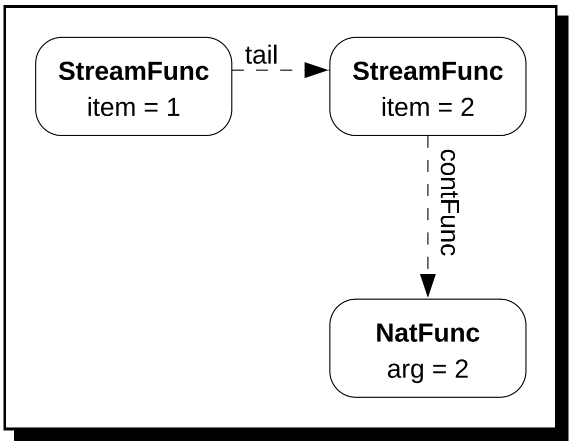


Figure 8.10: Streamfunction unfolding

Yet, given a suitable stream generation function and the optimization presented in section 8.9 on page 127, a direct "stepping" of the current **NatFunc** object can be achieved. The optimized forceTail method of **StreamFuncOpt** directly calls the step method of its continuation function and creates a new **StreamFuncOpt** with the obtained result. Hence, the intermediate creation of a **StreamTail** object is avoided — in this case without loss of generality.

Calling tail in the situation of figure 8.7 on page 130 creates the scenario in figure 8.10 on the preceding page. In this diagram — as well as in the following object diagrams — creation lines are drawn were requests on objects (e.g., tail) induce the creation of objects. Presenting the actual circumstances would unnecessarily complicate the pictures.

Further tail calls will finally replace **StreamFunc**s with **StreamVal**s (see figure 8.11).

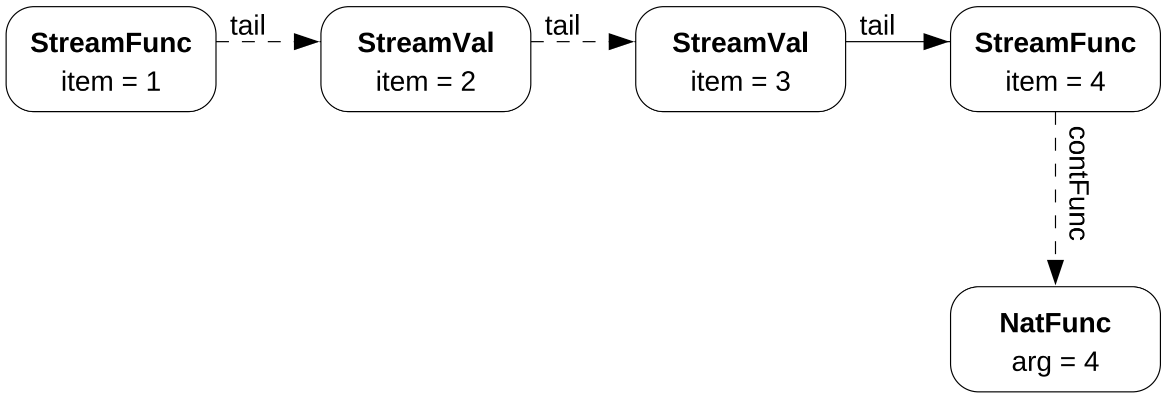


Figure 8.11: Streamvalue insertion

StreamVal objects, i.e., stream values are not suspended stream elements and allow discarding obsolete **StreamFunc** objects while representing their cached values.

Summarizing, we may observe that **StreamTail**s will be replaced by **StreamFunction**s, which in turn will be replaced by **StreamVal**s.

With our just established stream of natural numbers it is not very difficult to generate a stream of prime numbers. We make use of the Sieve of Erathostenes (see definition 1.5 on page 15) and obtain primes by filtering naturals. This time, however, we do not explicitely create a **StreamFunc** object but obtain the stream of prime numbers by applying a **Sieve** function on naturals:

```
nats          : StreamFunc[Integer];
primes        : Stream[Integer];
sieve         : Sieve;
...
!!sieve.make;
primes:=sieve @ (nats.tail);
```

Class **Sieve** encapsulates the stream function creation process and provides a nice application syntax. Its purpose is to create a **StreamFunc** instance that refers to a **SieveFunc** (see figures 8.12 on page 138 and 8.16 on page 147). The resulting situation is exactly analogous to the **NatFunc** case. I deliberately did not use an intermediate application syntax function for natural numbers in order to make the presentation of the underlying mechanisms as clearly as possible. However, clients are released from dealing with **StreamFunc** objects and are given a more natural way to deal with streams when provided with such adapter classes like **Sieve**.

For this purpose, class **Pipe** provides the generation of **StreamFunc** objects:

```
class Pipe[G, H]
inherit Function[Stream[G], Stream[H]];
creation init

feature
  pipeFunc : PipeFunc[G, H];

  init(f : like pipeFunc) is
  do
    pipeFunc:=f;
  end;

  infix "@" (s : Stream[G]) : Stream[H] is
  do
    !StreamFunc[H]!Result.make(pipeFunc @ s);
  end;
end
```

Class **Sieve** simply inherits **Pipe** and initializes `pipeFunc` to **SieveFunc** on creation (see also figure 8.16 on page 147).

The client code from above (`sieve @ (nats.tail)`) produces the situation of figure 8.12 on the following page. Here, natural numbers are not using the optimized **StreamFunc** in order to show that caching of tail values achieves that both the first natural number and the sieve function refer to the same stream suspension.

Figure 8.13 on page 139 shows the same structure after several item and tail calls. It can clearly be seen that **SieveFunc** employs **FilterFunc** objects to filter the stream of naturals:

```
class SieveFunc
inherit PipeFunc[Integer]
feature
  value : Integer is
  do
    Result:=stream.item;
  end;

  step : like Current is
    local filter  : expanded Filter[Integer];
          divides : expanded Divides;
```

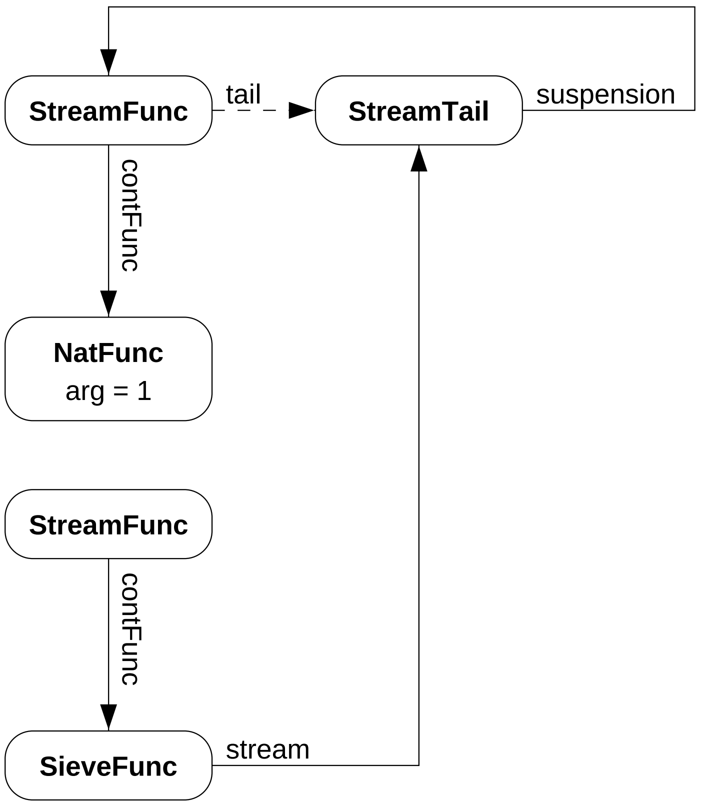


Figure 8.12: Stream of primes

```
            notf    : expanded NotF[Integer];
    do
      Result:=Current @
              (filter @ (notf @ (divides @ value)) @
               stream.tail);
    end;
  end
```

A **FilterFunc** will be generated by the application of **Filter**. Note how the three function objects are declared to be of expanded type in order to safe explicit creation.

SieveFunc makes use of the **PipeFunc** abstraction that specializes a continuation function to a function with one stream argument. Heirs to **PipeFunc** profit from a predefined eos function and are immediately subject to the nice application syntax provided by **Pipe** (see above). Providing application syntax and specializing the continuation function interface, **PipeFunc** is an heir to both **Function** and **ContFunc**:

```
  deferred class PipeFunc[G, H]
  inherit  inherit Function[Stream[G], PipeFunc[G, H]];
           ContFunc[H] redefine eos
  end
  feature
```

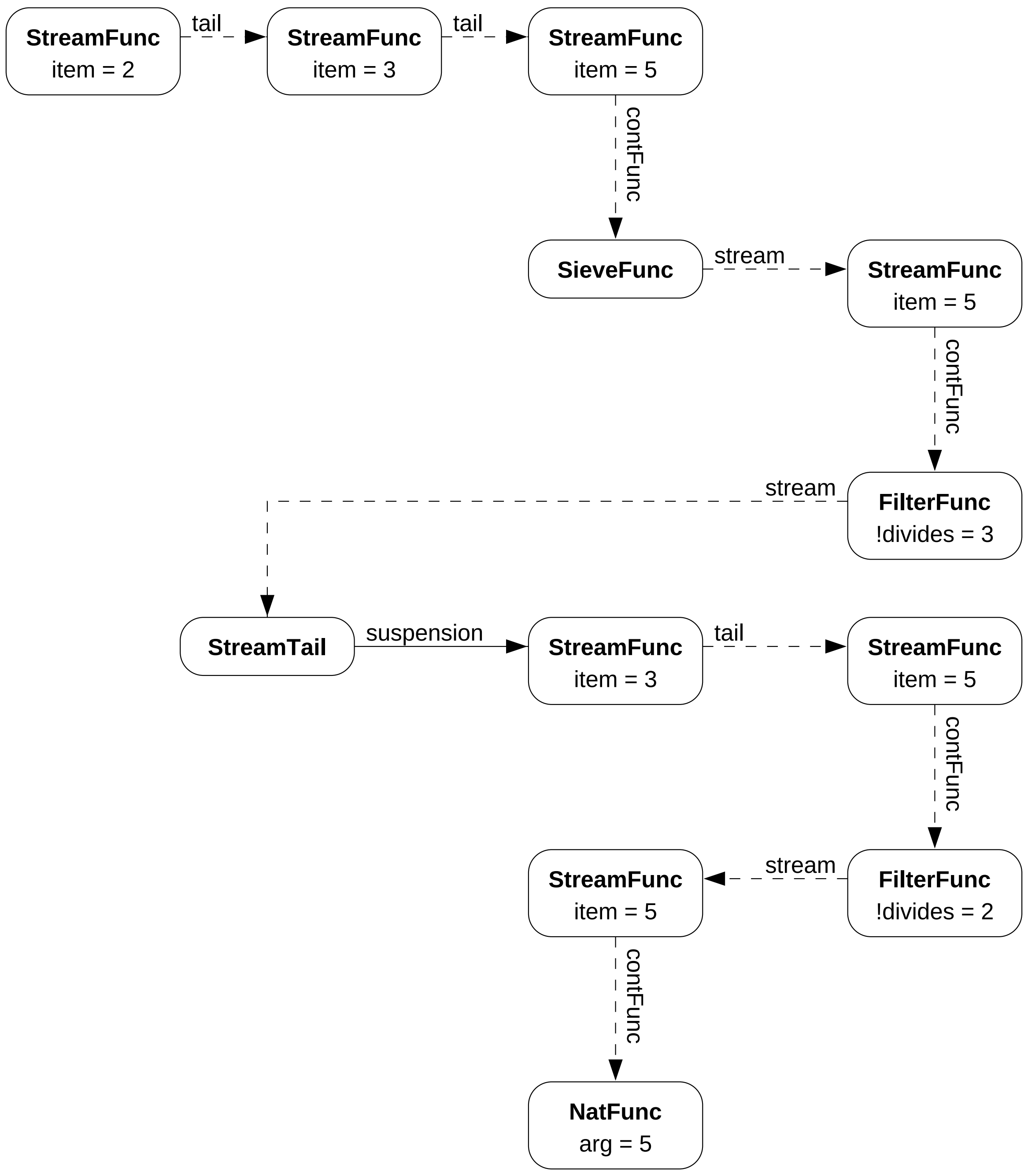


Figure 8.13: Unfolded primes

```
stream : Stream[G];

init (s: like stream) is
do
  stream:=s;
end;
```

```
      infix "@" (s : like stream) : like Current is
      do
        Result:=clone(Current);
        Result.init(s);
      end;

      eos : Boolean is
      do
        Result:=(stream.last);
      end;
    end
```

See also figure 8.16 on page 147 for all relations between **Client**, **Pipe**, **ContFunc**, and **PipeFunc**.

Requesting yet another prime from the structure of figure 8.13 on the page before results in figure 8.14 on the facing page. Starting from the bottom it can be observed that the current natural amounts to seven rather than five. The intermediate six has been filtered out. Note that the **StreamTail** object between the factor three and factor two filter has vanished and a new one was introduced right after a newly inserted factor five filter.

The same stream in a normalized state — resulting from some more item requests — is visible in figure 8.15 on page 142. Compared to figure 8.14 on the facing page,

1. a **StreamFunc** object has been converted to a **StreamVal** object,
2. a chain of **StreamFunc** objects has been reduced to one **StreamFunc** object,
3. and the natural number source has been advanced to number eleven.

From the dynamics of stream implementation — which is well hidden from the clients of streams — back to the static structure of the encountered participants: Figure 8.16 on page 147 shows all classes involved in the presented prime numbers example. Here, **Client** demonstrates the two ways to interact with stream functions. On the one hand it creates both **NatFunc** and **StreamFunc** and on the other hand just uses **Sieve**. The diagram also depicts that **Pipe** and **PipeFunc** support function application syntax by inheriting from **Function**.

It is interesting to follow the creation chain from **Sieve** via **SieveFunc** to **Filter**, which takes a predicate argument in order to create **Filter1** that in turn creates a **FilterFunc** object and, finally on receipt of a stream argument, will create a **StreamFunc** object (via class **Pipe**).

Note that the **Stream** subtree at the right part of the diagram exactly corresponds to the stream supporting classes of figure 8.8 on page 131. An abstracted version of figure 8.16 on page 147 with a client that deals with streams via intermediate stream functions only is presented in figure 8.17 on page 148. This diagram depicts the rich

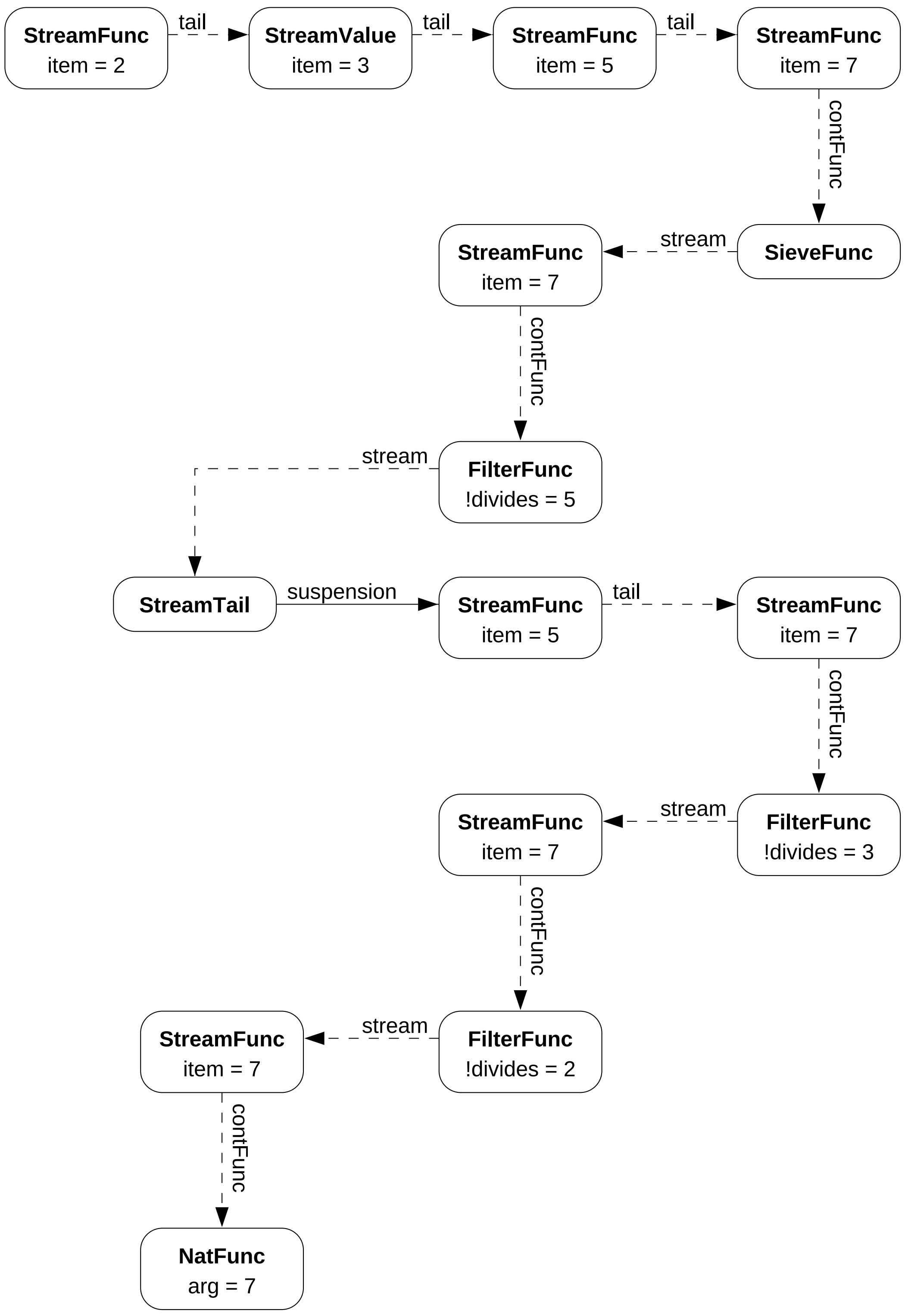


Figure 8.14: Expanding the sieve

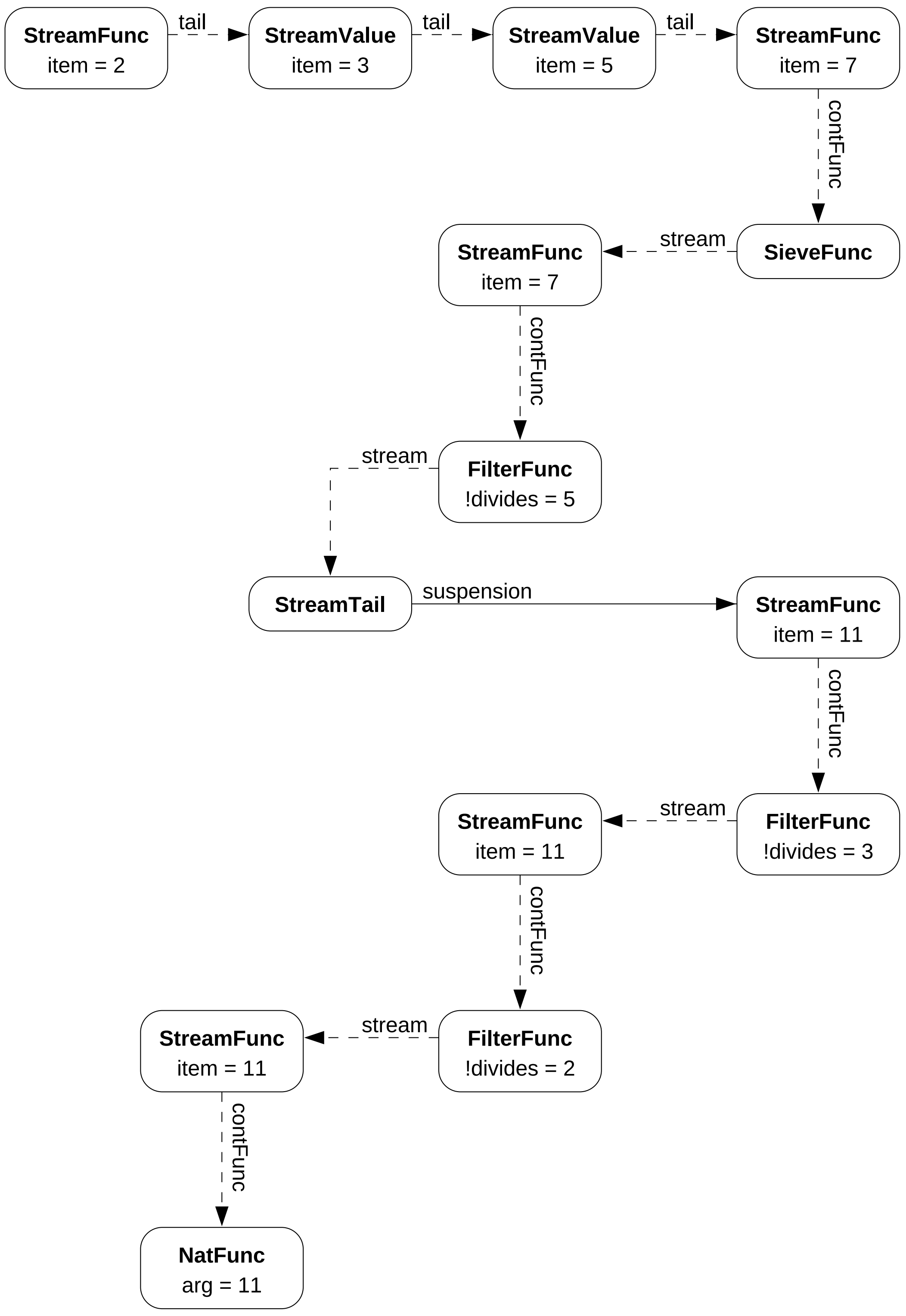


Figure 8.15: Normalized sieve

design of stream functionality that is available and ready to be used. Only the two classes in the lower left corner of the diagram with the “**Concrete**” prefix need to be supplied in order to introduce a new stream function. A new **ConcretePipeFunc** has to provide implementations for the value, step, and (only if deviating from the standard behavior) eos methods. The new **ConcretePipe** simply initializes class **Pipe**’s pipeFunc attribute to the new **ConcretePipeFunc**.

Often, it is not even necessary to introduce a new stream function but a standard function can be applied to a whole stream by using the stream function **Map**. Its implementation is as simple as this:

```
class MapFunc[G, H]
inherit PipeFunc[G, H]
creation make
feature
  mapFunc : Function[G, H];

  make (f : like mapFunc) is
  do
    mapFunc:=f;
  end;

  value : H is
  do
    Result:=mapFunc @ (stream.item);
  end;

  step : like Current is
  do
    Result:=Current @ (stream.tail);
  end;
end
```

8.11 Known Uses

The UNIX operating system owes much of its flexibility to stream based commands that can be interconnected via pipes [Ritchie84]. The common format of line separated ASCII characters allows the free composition of services. For instance, to uncompress a bitmap, convert it to POSTSCRIPT[10], and to finally view it, one can build a pipe of four commands: `cat bitmap.pbm.gz | unzip | pbmtolps | ghostview -`.

SATHER [Omohundro94] supports a restricted form of coroutine called an “iter” [Murer et al.93b]. Iters yield elements to be used in iterations. Their pri-

[10]PostScript is a registered trademark of Adobe Systems Incorporated.

mary difference to streams is the bounded lifetime, which ends with the iteration loop.

ALGOL 60 [Backus et al.63, Landin65] features call-by-name parameter passing. This mechanism enables delayed evaluation, save the “evaluate only once” strategy of call-by-need.

Programming languages with static array sizes (e.g., PASCAL [Rohlfing78]) force us to specify an array size that is often not know at the time of array creation. Programming languages that enable dynamic array sizes (e.g., EIFFEL) — in a way — allow for a lazy size specification for arrays. A `force` operation enlarges an array whenever necessary, so the initial size given does not restrict future array uses.

Graph algorithms in general are a popular target for lazy evaluation. David King and John Launchbury present algorithms with linear-time complexity that are constructed from individual components, which communicate via lazily generated intermediate structures [King & Launchbury93]. A lazy depth-first search tree of a graph will not cause the whole graph to be visited, when only part of the tree is requested. Even possible side-effects, such as marking nodes as visited, will only be performed as necessary [Launchbury & Jones94, Launchbury & Jones95]. In functional programming it is common to express, e.g., tree traversal problems first by flattening the tree into a list, and then by processing the list. The intermediate list can then provide a channel of communication between standard components. Lazy evaluation successfully avoids the transient creation of a large intermediate list and of large intermediate structures in general [Launchbury93].

Lazy graph algorithms also promise to be suited for multiprocessor architectures. The absence of side-effects and the memory friendly behavior of data bindings, as opposed to data structures, contributes to their suitability for parallel processing [Kashiwagi & Wise91].

The Convex Application Visualization System (AVS) is a visualization packages that organizes various modules, such as file readers and data processing, to a network of interconnected components [Convex93].

The Lazy Propagator pattern [Feiler & Tichy97] uses updates on demand in a network of distributed computations. A network node is updated only when requested for one of its values. A single request thus may cause a wavefront of backward updates until the system is just enough updated to answer the initial request.

Attribute grammars with lazy attributes can be specified without regard to the nature of dependencies (e.g., left-right, or even circular). Attributes will automatically be resolved in the order of their dependencies [Mogensen95].

For further examples the reader is referred to the literature given in section 8.2 on page 115.

8.12 Related Patterns

8.12.1 Categorization

Iterator: The interface of **Iterator** from the Iterator pattern is very similar to that of **Stream**. Both internal iterators and streams allow separating the iteration loop and collection navigation from the iteration body. Streams, however, do not provide a first method. As opposed to external iterators, streams are not stateful objects. To re-iterate a stream, one has to keep a copy of the stream's start.

Stream, Pipeline,
Pipes and Filters: are architectural views on the stream concept which can be provided by lazy evaluation [Ritchie84, Edwards95, Meunier95b, Buschmann et al.96, Posnak et al.96, Shaw96].

Singleton: A Singleton is a Lazy Object since it is created at most once. If created at all, every accessor will be given the same instance, which exactly corresponds to call-by-need semantics.

Co-Routine: A stream can be regarded as a co-routine. Resuming happens through `tail` calls and one can think of a series of subsequent continuation functions as co-routine incarnations distributed over stream element functions. The co-routining behavior of lazy objects is vital for their ability to avoid large intermediate structures. Calculation suspension allow building a pipeline that processes elements one by one rather then causing whole transient structures to be constructed in between.

8.12.2 Collaboration

Function Object: Function Object can be used for lazy evaluation as well. A function object represents a suspended calculation until requested to evaluate its result.

Streams are connected to function objects through class **Map**. It allows applying standard function objects to streams. Ergo, it is easy to obtain a stream function, if a corresponding a single element function already exists.

Translator: The result of a translation obtained using Translator can be a stream.

Transfold: Transfold processes lazy streams, that function as a *lingua franca* between collections and iterators.

Serializer: Serializer could be used to flatten data in order to transfer it via a standard stream channel. Instead of using Serializer on may alternatively simply use a specialized object stream.

Composite: A stream append operation (appending f and g yields $h = f \circ g$) can be regarded as a stream composite [Gamma et al.94, Manolescu97]. One may also use a **Component** to encapsulate the splitting of one stream into multiple branches and their re-joining into one stream again. For instance, one may filter various information types from a multi-media stream, compress them individually, and re-compose them to a compressed multi-media stream.

8.12.3 Implementation

Memoization: Being a technique rather than a pattern, memoization nevertheless is used by Lazy Object in order to perform calculations only once. Further requests are served by accessing a cache [Keller & Sleep86].

Value Object: A lazy object may use Value Object (see chapter 9 on page 149) to produce immutable cached values of calculated results, e.g., stream elements.

Command,
Function Object: **StreamTail** can be regarded as a function object with one argument (suspension) and three entry points with no further arguments. Its task is to separate the *creation* of an operation and its actual *invocation* in time, very similar to the Command pattern.

Factory Method: If clients defer the creation of stream functions to continuation functions — that know whether to choose a **StreamFunc** or **StreamFuncOpt** — the corresponding method is a Factory Method [Gamma et al.94].

Void Value: The behavior of a stream's end may be implemented with Void Value (see chapter 11 on page 191).

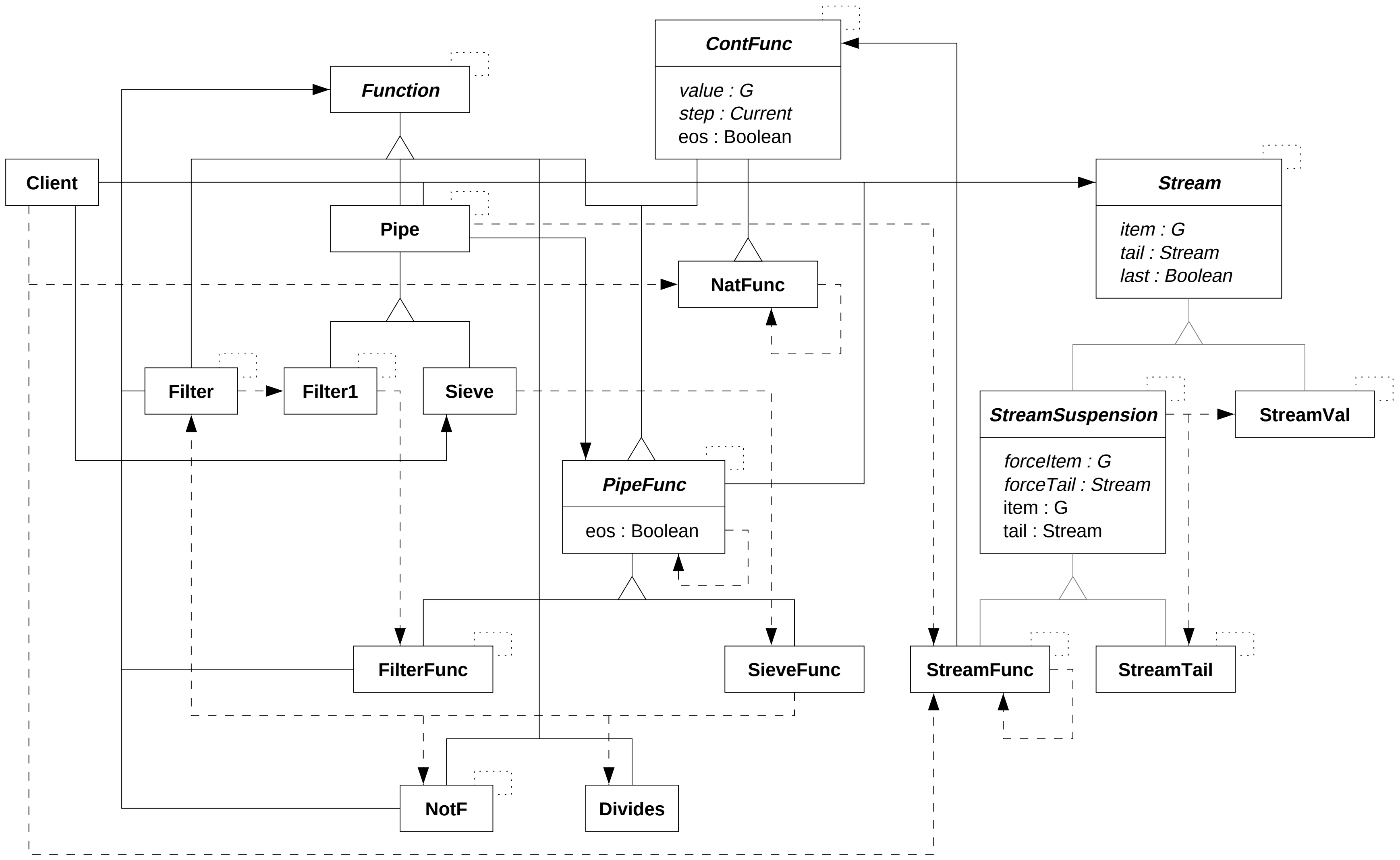


Figure 8.16: Sample structure

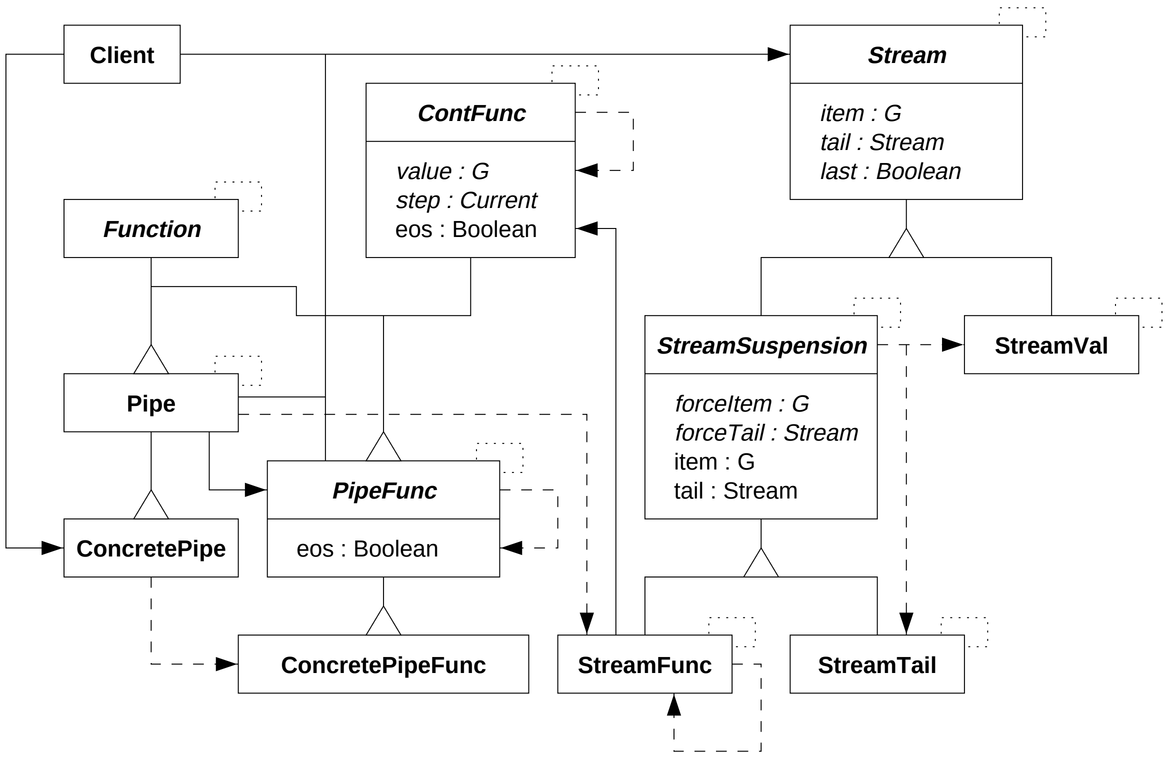


Figure 8.17: Stream structure diagram

9 Value Object

Even in change, it stays the same.
– Heraklit

9.1 Intent

Use immutable objects with generator operations for protection against side-effects and aliasing.

9.2 Also Known As

Immutable Object [Siegel95].

9.3 Motivation

Mutable objects are the bedrock of object-oriented design and programming. The ability to dynamically instantiate objects, change their value, retrieve their state, and share them among multiple users for communication allows the development of efficient software with a close relationship to real word models (see chapter 2 on page 29).

Yet, another bedrock of object-orientation is encapsulation (see section 2.2.2 on page 32). Encapsulated objects provide us with the peace of mind that every change to an object is controlled by the object itself. Obviously, an inconsistent or unexpected change of object state can be caused by the object itself only. Unfortunately, this is not quite true.

9.3.1 Problem

Surprisingly many object-oriented languages do not provide complex numbers either as primitive types or as a library component. Luckily, we can extend the existing library and provide a complex number class as if it has been part of the standard[1]. The straightforward approach is to write a class with a set of operations,

[1]As always the devil lurks in the details. See Matthew Austern's account on the difficulties of introducing a **Complex** class with EIFFEL.

e.g., add, mult, etc., that modify their receiver. There are countless examples of this approach in object-oriented libraries, especially for container classes. The change to an internal state is done by the most competent instance — the object itself — and it is the most efficient way to implement updates.

A partial implementation, therefore, could be:

```
class Complex
creation make
feature
  re : Real;
  im : Real;

  make(r : Real; i : Real) is
  do
    re:=r;
    im:=i;
  end

  plus(other : Const_Complex) is
  do
    re:=re + other.re;
    im:=im + other.im;
  end
end
```

Where is the problem? Let us have a look of a typical code fragment using the new class:

```
  ...
  local
    a : Complex;
    b : Complex;
  do
    !!a.make(1, 2);

    b:=a;
    a.plus(b);

    io.putstring(b.out);
  end;
```

Two variables of the new type **Complex** are declared and one is created with the value $1+2i$. It is only natural to move complex numbers between variables which happens in b:=a. Then a is changed by adding the value of b, i.e., it is effectively doubled. As the following print statement reveals, the value of b is not $1+2i$ anymore, but reflects the change that happened to a! Alas, what happened?[2] One the one hand nothing unusual took place: The value of a complex number variable was changed. Since another reference (b) to the value of a was created before, any change can be seen trough b as well.

On the other hand, our intuition about dealing with numbers has been severely hurt. The keyword to resolve this disturbing confusion is *reference*. We implemented complex numbers with reference semantics, i.e.,

1. variables are pointers to complex number objects,
2. operations change object state, and
3. assignment copies references only.

As a result, aliasing with unwanted effects is the standard behavior.

One way to control such introduced side-effects is to disallow the application of operations that mutate object state. Hence, it appears useful to introduce an immutable interface to complex numbers [Wilson95].

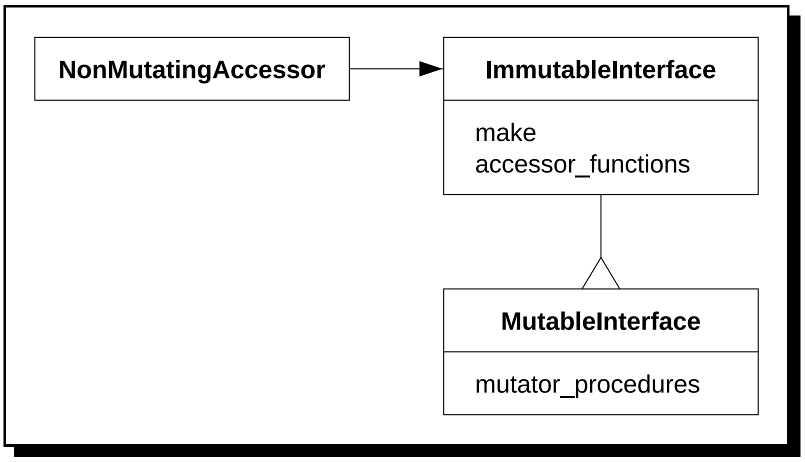


Figure 9.1: Providing a “const” interface

For instance, with

```
class Complex
inherit Const_Complex
creation make
feature
  plus(other : Const_Complex) is
  do
    re:=re+other.re;
    im:=im+other.im;
  end;
end
```

any caller of plus can be sure that the passed argument will still be unchanged after the execution of plus. This scenario assumes a hierarchy as depicted in figure 9.1. An attempt to assign `other` to a variable of type **Complex**, i.e., an mutable interface, results in a type error.

[2]Anyone not feeling irritated about this behavior is probably spoiled by commonplace reference semantics in object-oriented languages.

An immutable interface (i.e., the Constant Object pattern) is a powerful means to control side-effects and in

```
...
local
  a : Complex;
  b : Const_Complex;
do
  !!a.make(1, 2);

  b:=a;
  a.plus(b);

  io.putstring(b.out);
end;
```

one can be reasonably[3] sure that the value of `b` cannot be altered by passing it to plus. As already stated it is also not possible to modify the value of `b` by obtaining a mutable interface to it, e.g., by `a:=b`.

Unfortunately, the above code (using `b:=a`) will still compile and produce the unwanted effect. One might be tempted to reverse the hierarchy of figure 9.1 on the page before and make immutable interfaces inherit from mutable interfaces. Apparently, statement `b:=a;` would be illegal then. But,

- reversing the hierarchy of figure 9.1 on the preceding page is not sound in terms of subtyping. Class **ImmutableInterface** would have to *hide* operations from **MutableInterface** destroying substitutability. Therefore,
- it is no longer possible to pass mutable objects to servers with read-only interfaces. The const modifier property of immutable interfaces is lost.
- of course, it is still possible to create aliasing, e.g., with `a:=b;`.

In conclusion, immutable interfaces do not provide a suitable solution to reference semantics induced problems with complex numbers. After all, it should not be necessary to declare variables to provide read-only access in order to avoid the above aliasing effects. Moreover, we want to be able to modify complex numbers. There is no point in installing a safety mechanism that inhibits modification of values to such an extent.

9.3.2 Solution

Complex numbers are values, i.e., they are timeless abstractions and therefore unchangeable [MacLennan82]. They deserve to be implemented with value semantics. Instead of modifying the receiver of operation add, the receiver should be left

[3]Of course, a server may use a reverse assignment attempt to gain a mutable interface to an object passed through an immutable interface. Anyone doing so, is hopefully aware of the surprises possibly introduced.

untouched and a new instance of a complex number should be returned. In analogy, there are no operations that change integers. An expression like `41.inc` does not make sense, since 41 is a unique abstraction, rather than an object with identity [Khoshafian & Copeland86]. While it could be used to arrive at the abstraction 42 by adding 1 it is not possible to change it.

A version of **Complex** with value semantics is:

```
class Complex
creation make
feature
  ...
  infix "+",
  plus(other : Complex) : Complex is
  do
    !!Result.make(re + other.re, im + other.im);
  end;
end
```

Note that now it is now possible to use the more natural `infix "+"` notation for addition of complex numbers. It was added as a synonym for plus which we just retained for better comparison with the previous version of plus. The client code needs a bit of adaption to reflect the change —

```
  ...
  local
    a : Complex;
    b : Complex;
  do
    !!a.make(1, 2);

    b:=a;
    a:=a + b;

    io.putstring(b.out);
  end;
```

— and will, finally, work as expected. When `a:=a + b;` is executed, `a` is reattached to a new instance of a complex number. Variable `b` continues to point to the old value of `a`. A change to `a` finally no longer affects the value of `b`.

It is worthwhile noting that immutable interfaces remain useful for classes with reference semantics but are absolutely superfluous for classes with value semantics (e.g., method `plus` above does not need a **Const_Complex** argument interface anymore). There is no way a server could change a complex number anyway. Any modification results in a new complex number, leaving any passed instance intact. In other words, values do not require immutable interfaces.

9.4 Applicability

- *Modeling.* When clients are interested in values only and never care about identity, use Value Object for the object in question. Types representing values, i.e., abstractions with no identity for which creation, change, and sharing have no meaning, should be implemented with value semantics. Besides numbers, strings are a typical example. Sharing of string contents easily occurs with a straightforward implementation but is certainly unexpected[4].

- *Aliasing protection.* When a type should be immune to aliasing effects, replace its mutator operations with generator operations. For instance, a matrix multiplication using destructive updating of the receiver works fine until someone calls: `aMatrix.mult(aMatrix);`. This may destroy parts of the matrix that are still needed for input. With `aMatrix:=aMatrix.mult(aMatrix);` there is no such danger since target and source matrix are different.

- *Small objects.* Use Value Object preferably for small data structures. The larger the structure, the greater the overhead through duplication of unchanged data when creating a new instance in response to an operation. Using Value Object for larger structures can still be reasonable to avoid aliasing and does not necessarily have to be inefficient (see 9.9 on page 158, *Copy on demand*), but is most appropriate for objects with a small memory footprint.

- *Efficient value passing.* Creating new instances when changes occur is appropriate when the frequency of passing and assigning values is higher than their update frequency. If updating values is the predominant operation then consider to implement a copy-on-passing scheme (see 9.9 on page 158, *When to copy values*) or use a language's built-in support for value types (e.g., expanded types in EIFFEL or non-pointer variables in C^{++}).

9.5 Structure

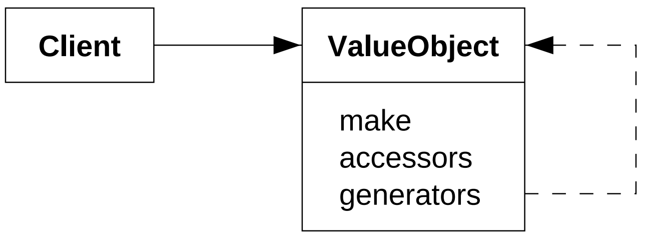


Figure 9.2: Structure diagram

[4]One of the few things the JAVA library got right: **String** objects are immutable and sharing can be achieved with **StringBuffer** only.

9.6 Participants

- **Client**
  - calls an accessor or generator operation of **ValueObject**.
- **ValueObject**
  - returns a value on access.
  - creates and returns a new object when asked for a modified version of itself.

9.7 Collaborations

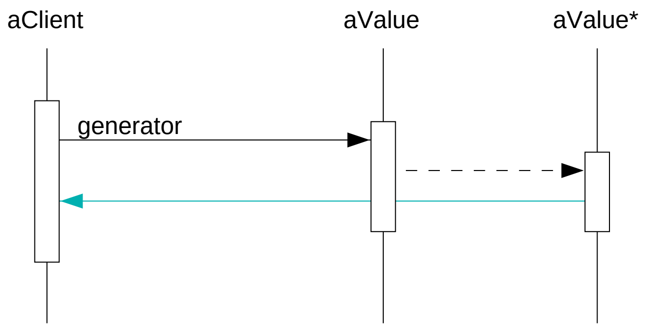


Figure 9.3: Interaction diagram

- A client performs an operation on a value object.
- The value object creates a new instance containing the information according to the requested operation.
- The value object returns the freshly created instance.

9.8 Consequences

- *Simplicity.* Code using values can be much simpler than code using objects, since there is no need to be cautious about side-effects and aliasing. All advantages of dealing with immutable values apply (e.g., validation of correctness, understandability, transformability (see section 1.3.1 on page 17)). Value objects do not offer mutator operations, but use generator operations to provide new instances. Note that object encapsulation is still preserved. Any changes to the object are still under the regime of the object itself. Plainly the target of the updated information is different to a conventional destructive update.

- *Encapsulation.* While direct aliasing (as presented in the example of section 9.3 on page 149) might be mastered with effort, indirect aliasing is a notoriously hard issue [Hogg91, Almeida97]. For instance, even expanded types in EIFFEL use a shallow copy on assignment only. As a result, components are shared and are subject to change while the owning object is not in control. Value Object provides full state encapsulation even for groups of objects.

- *Equality.* Testing value objects for equality cannot be accomplished by using a built-in equality operator such as "="[5]. The standard operator will compare pointer values, which can be different for otherwise equal values (but see also 9.9 on page 158, *Sharing of values*).

  The solution is to use a (usually already setup) equality method, like in

  ```
  if equal(aComplex1, aComplex2) then
    ...
  ```

  or using a custom infix operation:

  ```
  if aComplex1 #= aComplex2 then
    ...
  ```

  Often, as in EIFFEL, the setup equality method will perform as desired: It will compare all value components for equality. Nonetheless, it can be useful to override the equality method (`is_equal` in EIFFEL) in the value class, in order to achieve equality checking of abstract state rather than concrete state. For instance, a custom equality method could appreciate the equality of two complex numbers, even if one of them is represented with cartesian coordinates while the other uses polar coordinates.

- *Subtyping.* Usually "is-a" or specialization inheritance is at odds with subtyping. Although a set of integers "is-a" set of numbers it is not possible to pass it to a method expecting a set of numbers. This restriction is sound since the method could legally insert any number, e.g., a real number, violating the caller's integer set constraint.

  However, if the method's parameter is declared to be a *value* set of numbers then calling the method with a set of integer values is valid and safe. There is no way in which the called method could corrupt the passed set, since it is immutable.

  Other examples of "is-a" relationships that usually must be avoided but work in the context of value types are **Ellipse** being subclassed by **Circle** and **Rectangle** subclassed by **Square**. In both cases some mutations, e.g., stretching in one direction only, would throw subclass objects into trouble[6].

[5]Every JAVA programmer must discover that comparing strings with "=" does not work.
[6]Unless they can dynamically change their type, as with SMALLTALK's become: method.

Note that it is even save to pass a mutable square, i.e., one that offers destructive updates, to a method with a square value object interface (see figure 9.4 on the following page). Inside the method the mutator operations are not accessible and therefore subtyping is provided. Anyway, a generator method might return a result whose type is different from that of the receiver. For instance, stretching a **Square** yields a **Rectangle**.

Sadly, while subtyping with values actually works with the latter examples, there is no way to pursue EIFFEL's type system to believe that an integer instantiation of a generic class can be passed to a number instantiation. The type system rightfully assumes invalidness in the general case and cannot be hinted to the value semantics constraints that make it safe in this particular case (see section 14.5 on page 243).

- *Heterogeneity.* Reference types and value types cannot be used uniformly by clients. Besides the need for a different equality check[7], clients must either use

```
aComplex.add(aNumber);
...
aComplex:=copy(aNumber);
aComplex.negate;          -- keep aNumber intact
```

or

```
aComplex:=aComplex + aNumber;
...
aComplex:=aNumber.negate;  -- keep aNumber intact
```

This may appear unfortunate at first sight but it should be pointed out that value semantics — in this case for numbers — is not motivated historically but deeply roots in the fundamental difference between abstract values and changeable objects [Eckert & Kempe94]. Modeling of types and the corresponding syntax should reflect this difference.

- *Efficiency.*
  - + As value objects are immutable they can be shared safely, i.e., implemented with pointer semantics. Parameter passing and assignment are, therefore, cheap operations. Safe sharing of objects may also vastly simplify concurrent and parallel designs [Siegel95].
  - + Value objects can share subcomponents as long as these do not change. This reduces the amount of memory needed and speeds up copying of unchanged data to new instance created by generator operations.
  - – Creating a new object instead of modifying an existing one, increases memory demand and adds strain to the garbage collector.

[7] C^{++} with an overloadable "=" operator being an exception.

– Generator operations on large value object structure are expensive. Values with large components must recreate these even if only a tiny part of them changes. Changing a small part of a large value object is, thus, highly inefficient.

A mixed scheme — mixing generator and mutator operations — is possible (see figure 9.4), allowing mutator operations whenever safe. Unintentionally passing a value object to a method with a mutable argument interface, will be caught as an error by the type system.

If, on the other hand, the method uses an immutable argument (**ValueObject**) interface, callers are protected against changes to the argument, whereas code inside the method will be checked against unintended mutation attempts.

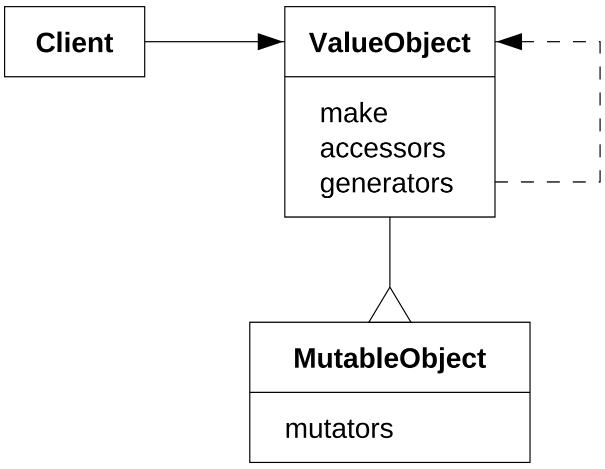


Figure 9.4: Optional mutators for Value Object

9.9 Implementation

- *When to copy values.* Value Object creates new values when values are generated. Another choice would have been to create copies whenever the possibility of aliasing is created, i.e., whenever a new access thread is created by parameter passing or assignment.

  In figure 9.5 on the next page an arrow pointing downwards denotes the creation of a new value by a generator operation. Indexes and shading indicate the generation of a value, i.e., the count of copy operations in its history. Note that the situation in figure 9.5 on the facing page is in favor of copy-on-passing rather than copy-on-change, but with less updates and more passing or assignments the score is reversed.

  Most programming languages implement their basic types with copy-on-passing. And in most languages we have no choice but implement Value Object with copy-on-change, because implementing copy-on-passing requires to modify the semantics of assignment and parameter passing. However, copy-on-change seems to be the better alternative in most cases, since assignment and passing should be more frequent than updates and it is more intuitive to expect a cost when operations are performed.

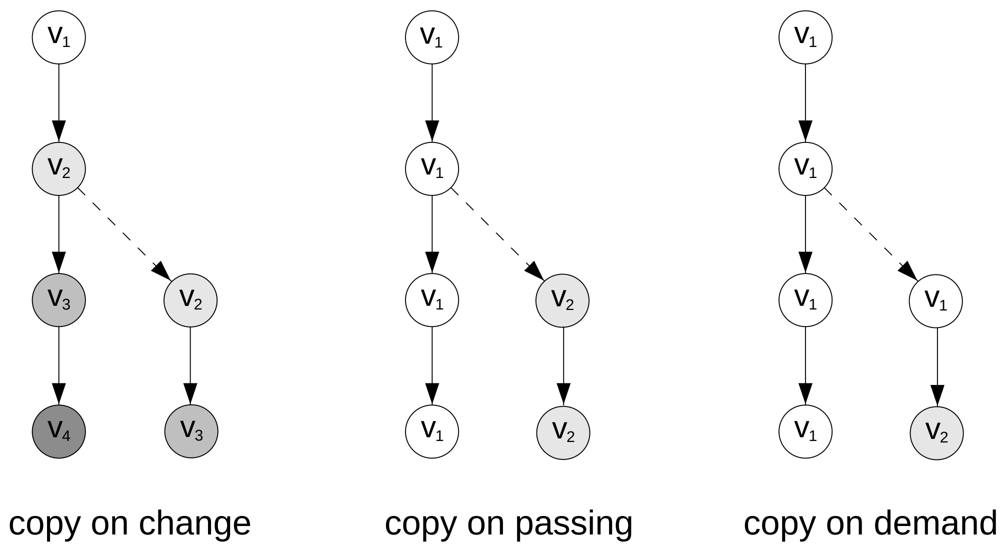


Figure 9.5: Copy strategies

- *Value Object versus language support.* Some languages offer support for the addition of types with non-reference semantics. Typically, however, another motivation was the driving factor, for instance, the ability to specify object containment rather than object referencing [Madsen et al.93, Meyer92]. Consequently, value semantics is often, e.g., in EIFFEL, not sufficiently supported [Kent & Howse96]: Expanded types, for instance,
  - do not allow polymorphism.
  - cannot be specified with deferred classes.
  - make shallow copies on assignment only.
- *Abstract state.* Value objects must preserve equality only modulo their respective equality operation. For instance, fraction numbers could decide to normalize their representation after addition but not after multiplication. Hence,

$$\frac{15}{6}+\frac{15}{6}=\frac{15}{3}$$

while

$$\frac{15}{6}*2=\frac{30}{6},$$

but comparing the two results would yield that they are equal.

- *Deep copy.* If a value object has subcomponents consisting of standard reference type objects, it has to create a deep copy of them in case it produces an altered copy of itself. Otherwise unintended sharing of subcomponents is inevitable.

- *Copy on demand.* A (deep) copy of a value subcomponent is not necessary if that component does not change due to the requested operation. At least for components a value object can implement the copy-on-demand strategy of figure 9.5 on the page before. Only if a subcomponent changes, a copy has to be created before any changes are made.

- *Sharing of values.* Ordinarily, values are generated without regard whether an equal instance of the result already exists. If every result is first looked up in a repository of already existing values, then

  - memory can be saved by avoiding multiple value instances with the same value.
  - equality testing can be achieved by pointer comparison again. This is much more efficient and allows to use standard syntax for comparison.
  - a repository has to be kept[8] and looking up equal values consumes time.

  If values are more frequently compared than transformed than it pays off to use the above — so-called Singular Object [Gamma95] or Identity Object [Smith95] — strategy.

9.10 Known Uses

It is needless to emphasize that functional programming languages treat their computational “objects” as values. Many object-oriented languages split their types into basic types (with value semantics) and reference types. Even in SMALLTALK, where everything is an object, numbers are *correctly* treated differently, i.e., they do not provide mutator but generator operations.

Ian Poole et al. use a combination of Lazy Object and Value Object, i.e., immutable lazy values to represent complex computation networks [Poole et al.98]. The benefits of their Lazy / RT[9] pattern (e.g., modular reasoning, parallel evaluation multiple level undo, exception recovery, etc.) is a combination of the consequences of both Lazy Object (see section 8 on page 115) and Value Object.

[8]In SMALLTALK such a collection can readily be obtained by sending `allInstances` to an object.

[9]Lazy and referentially transparent objects.

9.11 Related Patterns

9.11.1 Categorization

Constant Object: A value object interface to a mutable object fulfills the same purpose as an immutable interface (see section 9.3 on page 149). In addition to accessor functions, a value object interface, furthermore, provides generator operations, which can be used to create new values without altering the original.

Identity Object,
Singular Object: This pattern emerges when Value Object or Constant Object is implemented with unique value instances as described in section 9.9 on page 158. For instance, the datatype **Symbol** in SMALLTALK is implemented as an Identity Object.

Unshareable Object: Another way to prevent aliasing it to allow single references to an object only [Minsky95, Minsky96]. While values can be motivated from a modeling point of view, it is unclear what the real world equivalent of an unshareable reference is. Unshareable references are not copied but moved, i.e., the source of an assignment statement will be `nil` afterwards. A useful analogy might be a physical object that can only be moved but not shared or a unique item that is allowed to occur only once, e.g., a token in a network.

9.11.2 Collaboration

Function Object: If function objects accept arguments through a value object interface, they do not need to copy arguments in order to be safe from future changes. This, however, does not hold if there is also a mutating interface available (see figure 9.4 on page 158) through which arguments could be corrupted.

Strategy,
Function Object: The equality operation can be parameterized with a function object predicate. Depending on the application, several “views” on equality could be implemented (e.g., books which differ in publishers only could be considered equal for a certain purpose).

Void Value: Default values (e.g., $0+0i$ for complex numbers) can be provided with Void Value.

Transfold: Due to the absence of aliasing, it is less error prone to generate new values from a collection of values than dealing with shared objects. A value aggregate can be iterated without concerns for changes occurring during iteration.

Lazy Object: Pattern Lazy Object (see chapter 8 on page 115) may be used to lazily initialize value object components.

Decorator: A decorator could be used to provide a value semantics interface to standard reference types. A decorator must first copy the receiver to a new instance, then update that instance according to the operation, and finally return the new instance.

9.11.3 Implementation

Lazy Object: A value object may return a lazy result. Especially, if the calculation of a new value is costly and frequently only parts of the result are actually used then it is worthwhile to defer the calculation until the result is accessed.

Transfold: A value object may implement equality by using Transfold to compare reference type collections in subcomponents.

Translator: Extremely complex values may implement sophisticated operations — such as providing a certain view or an interpretation of a value — by *translating* their subcomponents.

10 Transfold

I can't understand why a person will take a year or two to write a novel when he can easily buy one for a few dollars.
– Fred Allen

10.1 Intent

Process the elements of one or more aggregate objects without exposing their representation and without writing explicit loops.

10.2 Motivation

Collections play a very important role in software design. They allow us regarding a bunch of objects as a single abstraction. Clients pass and store collections as single entities and defer the responsibility of reacting to a message to the collections. A single request to a collection may mean to forward the request to all collection members (e.g., most **Composite** services in the Composite pattern), find a member best suited to deal with the request (e.g., Chain of Responsibility pattern), extract one member with particular properties (e.g., a detect method), etc. Hence, with the exception of a few **Dispenser** types like **Stack** and **Queue**, that restrict access to one end, it is common to access collection members one by one until a condition is met or full coverage has been achieved. In other words, a frequent operation on collections is iteration.

10.2.1 Problem

It is clearly not an option to let clients iterate over collections — or more generally, aggregates — using knowledge about an aggregate's internals. In

```
...
  list    : List[Integer]
  l       : Link[Integer];
do
  from l:=list.first;
  until l=void
```

```
      loop
        io.putint(l.item);
        l:=l.next;
      end;
    ...
```

the client is committed to a linked list implementation of **List**. If the implementation of **List** changes to a balanced tree representation, e.g., to make searching a faster operation, each client iteration as above is subject to change. Even C^{++} code as commonplace and innocent looking as

```
    ...
      for (i=0; i<count; i++)
        cout << a[i];
    ...
```

is almost certainly more specific than necessary. Instead of relying on `a` to be an array or indexable, the above code should assume no more than the properties of an ordered collection in order to protect itself from future changes.

Evidently, an iterator abstraction is necessary that allows accessing the elements of a collection or the components of aggregates independently of their respective internal representation. With the decision for a dedicated iteration abstraction, however, the following issues must be resolved:

- *How to combine iterator and action?* How do we combine an iteration algorithm (e.g., a simple loop) with a particular function or action (e.g., print an element)? The iterator client may do it by calling both, the iterator could be subclassed for each function, or the iterator "takes-a" function.

- *Who knows how to iterate an aggregate?* Where is the best place to put the iteration logic? Shall the aggregate explore itself or is it better to externalize such functionality?

- *Who controls the iteration?* Who is in control of advancing the iteration and who may decide to stop prematurely, i.e., avoid a full exploration?

- *How to support several iteration strategies?* Non-linear aggregates (e.g., trees or graphs) support different traversal strategies like breadth-first-search and depth-first-search with variations such as pre-order, in-order, and post-order traversal. How do we allow a choice between alternatives without bloating the aggregate's interface?

- *How to allow several iterations in parallel?* In which way can we support multiple iterators using a shared aggregate? Two or more clients may want to process the same aggregate with interleaving execution. It may even be the case that a read-iterator follows the results of a write iterator.

- *How to deal with alterations during iteration?* What means are available to make iteration over aggregates which are changed during iteration a safe and unambiguous operation? The further discussion will not deepen the question of how to make iterators robust [Kofler93], since it is orthogonal to the issues of higher interest here.

Some proposed solutions can be easily dismissed:

- Combining iterator and iteration function with inheritance [Madsen et al.93, Meyer94b, Martin94] does not scale with respect to the number of traversal alternatives, iteration functions and implies other problematic issues [Kühne95b] (see also section 7.3 on page 93).

- Equipping aggregates with a cursor that allows iterating the whole aggregate (e.g., lists [Meyer88, Omohundro & Lim92]) is problematic in the presence of multiple iterations. Even if interference is prevented by providing a cursor stack, which clients use to push and pop cursors, the resulting scheme is inelegant and error prone [Kofler93]. This suggests that the state of iteration should be kept outside the iterated aggregate.

- Schemes relying on language support such as co-routines or specializations thereof [Liskov & Guttag86, Murer et al.93b], are not easily applicable in languages without these mechanisms.

We are left with two fundamentally different approaches:

1. External iterators place the iteration logic outside of aggregates and provide clients with an interface to start, advance, and inquire the actual element of an iteration.

2. Internal iterators are typically a part of the aggregate's interface. When given an iteration function they autonomously perform the traversal, thus, releasing the client to provide a control structure.

External and internal iterators are also referred to as active and passive [Booch94], with respect to the client's role. Internal iteration corresponds to functional mapping, i.e., the parameterization of higher-order iterators with iteration functions.

There are a number of arguments in favor of internal iteration:

> *"The code for initializing, updating, and testing iteration variables is often complex and error prone. Errors having to do with initialization or termination of iteration are sometimes called* "fencepost[1]" *errors and they are very common[Murer et al.93b]."* – Sather Group

This suggests to "Write a Loop Once" [Martin94], i.e., code the traversal control once inside the aggregate and let all clients rely on it. Hence, the duplication of virtually identical code in clients for stepping through a structure is avoided. A particular expensive incident was caused when the Mariner space-probe was lost due to an error in a loop [Neumann86].

[1]While genuine fencepost errors are avoided with external iteration as well, it is nevertheless easy, e.g., to forget to advance the iteration or doing it at the wrong place.

```
DO 3 I = 1.3
```
Code to be executed with I=1, 2, 3.

——— and if it had been C^{++}, maybe ... ———

```
for (i=1; i<3, i++);
```
Code[2] to be executed with I=1, 2, 3.

Figure 10.1: Code example "Goodbye Mariner"

In the FORTRAN code of figure 10.1 the dot should have been a comma. As it happened, just the value `1.3` was assigned to `I` without causing any iteration at all. An internal iteration might not have been applicable, but at least the above example demonstrates that (fatal!) errors can be introduced in even the most simple loops.

Traversal strategies often rely on internal aggregate details for stepping through aggregates [Murer et al.93b] and it is easier to use a recursive method for descending a structure than to memorize an access path externally [Gamma et al.94].

All the above observations argue in favor of making iteration an autonomous operation of the aggregate, but

> *"External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like* C^{++} *that does not provide anonymous functions, closures, or continuations like* SMALLTALK *and* CLOS. *But on the other hand, internal iterators are easier to use, because they define the iteration logic for you. [Gamma et al.94]."* – GOF Group

A number of other authors agree that either built-in closure support is needed to use internal iterators [Baker93, Kofler93, Gamma et al.94] or that internal iterators are inflexible to the extent of disallowing the comparison of two data structures [Murer et al.93b, Kofler93, Gamma et al.94, Budd95, Norvig96].

Also, it seems that clients know best when an iteration can be stopped, e.g., when an element has been found, and internal iterators do not account for this[3].

The Iterator pattern [Gamma et al.94] implements external iteration and resolves a number of issues. The combination of iterator and iterator function is trivial, since the client calls both in a dedicated loop. Pattern Iterator gives the client full control of iteration advancement and termination. It, furthermore, allows multiple traversal strategies and multiple iterators on the same aggregate. It is weak on requiring clients to duplicate control structures, time and again. It also may force aggregates to provide a (possibly protected) interface to allow their efficient scrutinization for traversal. Finally, an external iterator has to keep track of the iteration

[2]Can you spot all three C^{++} errors?

[3]This is not true for SMALLTALK where blocks returning a value cause control to be passed back to the iteration client.

state — which may involve record keeping of paths into tree-like structures — instead of allowing a self managed exploration which, in case of a recursive method, uses the method calling stack for storing the access path.

Diametrically, an internal iterator is strong on concentrating the control to one loop, information hiding of aggregate internals, and competence of aggregate exploration. Unfortunately, it requires closures for combining iterator and iteration function, takes termination control out of the client's hands, and allows only one iteration at a time. Furthermore, multiple traversal strategies cause the aggregates interface to be bloated with iteration methods. Table 10.1 summarizes the comparison of external with internal iteration again. The last point in table 10.1 refers to the

	Iterator kind	
	external	*internal*
combination of iteration and action is trivial	✓	(✓)*
record keeping of iteration state is straightforward		✓
no special access interface to aggregate is required		✓
no explicit loop needed for client iteration		✓
client may stop iteration early	✓	✓†
iterating multiple aggregates in lock-step is easy	✓	
traversal alternatives do not bloat aggregate's interface	✓	
multiple iterations sharing one aggregate	✓	
no parallel hierarchy of iterators and data structures		✓

*Support for closures required.
†By inelegantly passing a `continue?`-flag from function to iterator.

Table 10.1: External versus internal iteration

fact that external iterators typically depend on the properties of the data structures they traverse (e.g., trees call for different iterators than linear structures). Therefore, it is common to observe a class hierarchy of iterators paralleling that of the data structures [Gamma et al.94, Meyer94b].

While internal iterators seem to be more faithful to software engineering matters they apparently let clients down in terms of straightforward (active) use and flexibility.

10.2.2 Solution

An elegant way to resolve the forces is to

- provide a way to flatten aggregates to a stream of data and
- iterate passively over (multiple) streams.

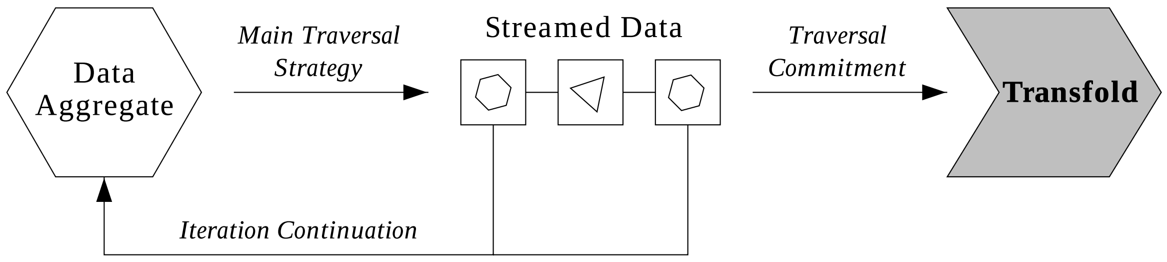


Figure 10.2: Iteration topology with lazy streams and Transfold

For instance, a tree is first tranformed into a linear stream and then this stream is consumed by an internal iterator. An element of the intermediate stream might contain a lazy exploration of further parts of the aggregate, i.e., an iteration continuation (see figure 10.2). The stream consumer, hence, can decide which parts are to be explored next. Therefore,

- the client does not need to provide a control structure.
- the aggregate keeps the record of its exploration by being responsible for stream generation.
- a client may stop iteration early, since it is in control of stream consumption. As streams are generated lazily, no unnecessary exploration will take place (Figure 10.2 shows some eagerly produced data elements for illustration purposes only).
- traversal (consumption) alternatives are defined outside the aggregate and do not bloat it's interface.
- it is possible to share an aggregate for multiple iterations by the independent consumption of a shared stream.
- no parallel class hierarchies evolve since stream consumption is invariant and stream generation is defined in data structures.

Two crucial points remain unresolved: How to circumvent the need for closure support and how to avoid the criticism of inflexibility towards internal[4] iterators?

Of course, the first problem is immediately solved by referring to the Function Object pattern [Kühne97] (see chapter 7 on page 93).

The second problem — the inability of internal iterators, e.g., to compare two aggregates with simultaneous iterations — requires a small but very effective idea. Indeed, it is practically impossible to consider a second iteration while an internal iteration focuses on its sole aggregate iteration. One possible solution consists of using a function object, curried with the second stream, that compares the element

[4]Note that "internal" now only characterizes the passive iteration style rather than the location of the iteration interface with regard to the aggregate.

passed from the iteration with the corresponding second stream element. Yet, the use of a stateful function object that consumes one of its arguments on application, is at most an escape route, but cannot be considered to be a solid software engineering answer to the original problem.

The problem, however, is easily resolved by generalizing internal iteration from one to many aggregates (see figure 10.3). An internal iterator, consuming two number streams, for instance, takes a function with two parameters and applies it to the two foremost numbers. Then, both streams are advanced simultaneously and the next application will be to the two following numbers (Chapter 13 on page 221 demonstrates the application of Transfold to a well-known problem of simultaneous iteration, the *samefringe* problem).

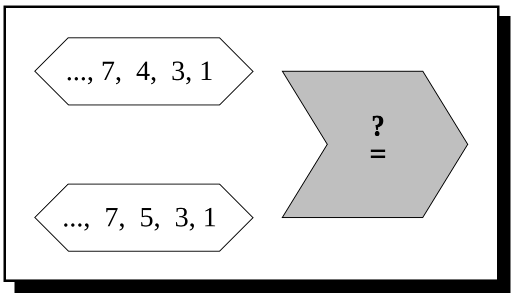


Figure 10.3: Iteration of two aggregates in lock-step

A functional programmer would probably first transform two list of numbers to one list containing 2-tuples of numbers, and then iterate over the resulting list. For instance, producing a list of sums of the respective elements of two argument lists can be expressed as:

$$\begin{aligned} sum\_lists\ xs\ ys &= map\ f\ (zip\ (xs,ys)^5) \\ &\quad where\ f\ (x,y)^6 = x+y. \end{aligned} \tag{10.1}$$

Instead of defining *f* in *sum* it would be better to make it an argument of *sum_lists*. Passing a different function then allows calculating the list of products, etc.

While two iteration arguments are sufficient for comparing and the above summation example, in general, it would be desirable to iterate over an arbitrary number of iteration arguments. The problem with tuples, however, is that it is not possible, for instance, to pass *sum_lists* a function that will sum up the all the components of its argument tuple, regardless of the component number. In order to be able to pass functions that work independently of the number of iteration arguments, the argument type of the iteration function has to be changed from tuple to list.

Of course, a generalization of *zip* is now required, i.e., a function that takes *n* input lists and transforms them to a list containing sublists of length *n* (see figure 10.4 on the following page). The length of the result list is determined by the shortest length of the *n* argument lists. Since the type of *transpose* and accordingly any other variable iteration argument function, should not change with the number of iteration arguments, we pass the *n* iteration arguments as elements of one argument list to *transpose*. Given this function — whose definition shall be

$$\begin{aligned} trans\ [\,] &= [\,] \\ trans\ xss &= if\ fold\ ((||) \circ ([\,] ==))\ False\ xss\ then\ [\,] \\ &\quad else\ (map\ head\ xss) : trans\ (map\ tail\ xss) \end{aligned} \tag{10.2}$$

[6]Tuple construction.
[6]Tuple pattern matching.

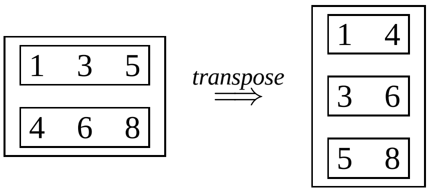


Figure 10.4: Transposing lists

— it is now possible to generalize *sum_lists* to a variable number of iteration arguments. Let us first re-implement *sum_lists* with *transpose*,

$$\begin{aligned} \mathit{sum\_lists}\ xs\ ys &= (\mathit{map}\ f)\ (\mathit{transpose}\ [xs\ ys\,]) \\ &\qquad \mathit{where}\ f\ zs\ =\ \mathit{fold}\ (+)\ 0\ zs \end{aligned}$$

and then — while making the iteration function f an argument — change its number of input lists, from two to an arbitrary number in one input list:

$$\mathit{sum\_lists}^{\,7} f \quad = \quad (\mathit{map}\ f) \circ \mathit{transpose}$$

It is now appropriate to call *sum_lists* with a more general name. Let us use *transmap*, because it maps a function to a transposed argument list. Passing a function that computes the product of its argument list, i.e.,

$$\mathit{transmap}\ (\mathit{fold}\ (*)\ 1)\ [[1\ 3\ 5]\ [4\ 6\ 8]] \quad = \quad [4\ 18\ 40]$$

creates a list of products (compare with figure 10.5) Now, it would be only a matter

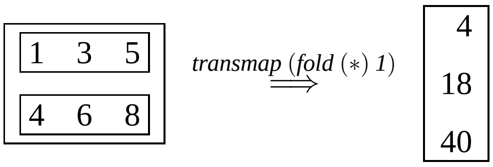


Figure 10.5: Computing a list of products

of summing up the values in the result list (vector) to obtain the inner product of the two argument lists (vectors). To further motivate the extension of *transmap* to *transfold* let us investigate how far we get, using *transmap* to perform the equality operation of figure 10.3 on the preceding page (see figure 10.6 on the next page). Note that a lazy transposition allows terminating the exploration of the (possibly infinite) argument lists when a non-equal argument pair has been found.

[7]No argument list is explicitely mentioned, because it would be just passed on to *transpose*, i.e., we made an η-reduction.

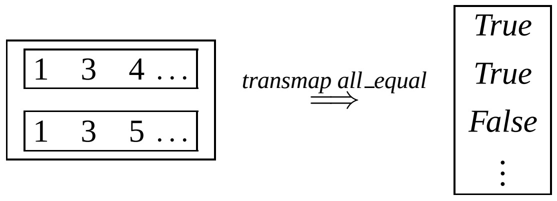


Figure 10.6: Transmapping equality

Obviously, we need to reduce the result list with the logical *And* (&&) operator to obtain a single equality result. Likewise, the result list of figure 10.5 on the facing page requires reduction with "+" to obtain the final inner product.

Therefore, *transfold* is defined to be

$$
\begin{array}{rrcll}
 & \textit{transfold}\ f\ a\ g & = & (\textit{fold}\ f\ a) \circ (\textit{transmap}\ g) & \\
 & & \text{or} & & \\
 & \textit{transfold}\ f\ a\ g & = & (\textit{fold}\ f\ a) \circ (\textit{map}\ g) \circ \textit{transpose} & (10.3)\\
\text{with type} & \textit{transfold} & :: & (b \to c \to c) \to c \to ([a] \to b) \to [[a]] \to c & (10.4)
\end{array}
$$

See table 10.2 for type and meaning of *transfold*'s parameters.

Parameter	*type*	*purpose*
f	$b \to c \to c$	function that finally reduces the intermediate result of element type b to a result of type c, using the initial element.
a	c	the initial element for producing the final result, used as the induction base for an empty list.
g	$[a] \to b$	the function that is applied (mapped) to each row of the transposed argument, transforming a row $[a]$ to an element of the intermediate result of type b.
	$[[a]] \to c$	resulting type of *transfold* after all arguments but the last are supplied. Transforms a matrix (list of lists) with element type a into a result of type c.

Table 10.2: Transfold's arguments

For instance, with $\textit{all\_equal}\ xs = \textit{fold}\ ((\&\&) \circ ((\textit{head}\ xs)\ ==))\ \textit{True}\ (\textit{tail}\ xs)$[8]:

$$
\begin{array}{rcll}
\textit{transfold}\ (+)\ 0\ (\textit{fold}\ (*)\ 1)\ [[1\,3\,5]\ [4\,6\,8]] & = & 62,\ \text{and} & (10.5)\\
\textit{transfold}\ (\&\&)\ \textit{True}\ \textit{all\_equal}\ [[1\,3\,4\ \ldots]\ [1\,3\,5\ \ldots]] & = & \textit{False}. & (10.6)
\end{array}
$$

[8]This function appears to be overly complicated but a moment of thought reveals that one cannot just reduce with $==$, since the latter has a boolean result type, unsuitable for further comparisons.

Note that we achieve reduction with *fold* which can be thought of being composed of a *map* and a *reduce* function, i.e.,

$$\mathit{fold}\ (\oplus \circ f)\ a\ xs \quad \equiv \quad \mathit{reduce} \oplus ((\mathit{map}\ f\ xs) +\!\!+ [a]) \tag{10.7}$$

Here, *reduce* places its binary operator $\oplus$ between the elements of a list. This operation is also referred to as *fold1*, whereas *fold* uses an additional element[9] to account for lists with less then two elements. Anyway, *fold* is clearly more versatile than a simple *reduce* (see also section 10.3 on the facing page, *Versatility*).

To illustrate the flexibility of *transfold*, which is reflected in the intermediate type *b* (see table 10.2 on the page before), let us calculate the sum of all row products from a *magic square* (see figure 10.7). We use

$$\mathit{transfold}\ (+)\ 0.00\ (\mathit{fold}\ (*)\ 1.0) \tag{10.8}$$

where the input matrix contains integer elements, 1.0 denotes a real, and 0.00 denotes a double. Hence we establish the mapping

$$[a \mapsto \mathit{integer},\ b \mapsto \mathit{real},\ c \mapsto \mathit{double}\,].$$

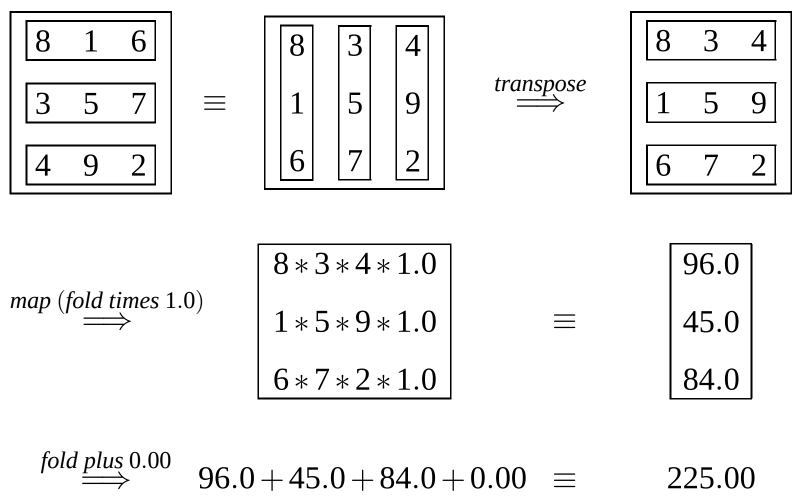


Figure 10.7: Transfolding the inner product of a matrix

Folding does not have to imply reduction, though. Using functions *reverse* and *add_back*, that establish the mapping $[a \mapsto \mathit{integer},\ b \mapsto [\mathit{integer}],\ c \mapsto [[\mathit{integer}]]\,]$, we may transpose a matrix along its minor axis (see figure 10.8 on the next page).

Quite similar to the situation in definition 10.7, it is possible to get rid of *transfold*'s parameter *g* by passing a different function to parameter *f*, i.e.,

$$\mathit{transfold}^{*}\ (f \circ g)\ a \quad \equiv \quad \mathit{transfold}\ f\ a\ g.$$

[9]Typically the right-identity value for $\oplus$.

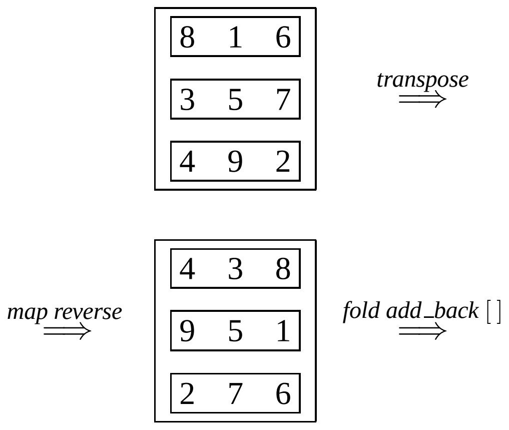


Figure 10.8: Minor axis matrix transposition

I felt, however, that it is clearer to separate the imaginative horizontal and subsequent vertical processing functions explicitely. First, the combined function $f \circ g$ can become hard to read, e.g., see definition 10.5 on page 171 repeated with *transfold**:

$$\mathit{transfold}^* \; ((+) \circ (\mathit{fold}\; (*)\; 1))\; 0\; [[1\,3\,5]\; [4\,6\,8]] \quad = \quad 62.$$

Second, special cases can be pre-defined with default parameters (e.g., the net effect of either function is often just the identity function; see section 10.7 on page 176, *Separation*). Furthermore, frequently occurring functions (e.g., reversal) can be reused and intermixed without performing function composition first.

10.3 Applicability

- *Abstraction.* Transfold allows accessing the elements of a collection or the components of an aggregate without exposing its internal representation.
- *Concentration.* Use Transfold if you want to release clients from the duty of providing a control structure.
- *Multiple traversals.* When a structure is to be iterated by multiple clients in alternation, Transfold allows sharing the structure's stream for independent consumption by multiple clients.
- *Connectivity.* If you want to cascade transformations and / or want to convert collections into each other, possibly with intermediate processing, use stream producing transfolds and collection constructors with stream arguments for a flexible interconnection scheme (see figure 10.9).

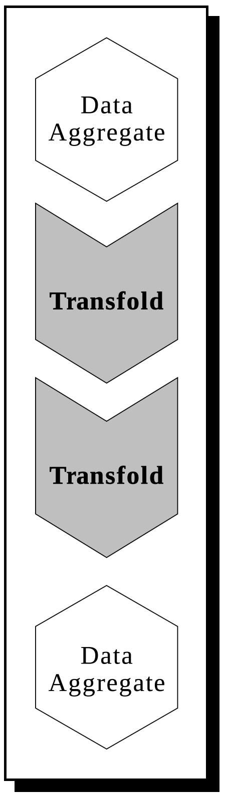


Figure 10.9: Transformation chain

- *Separation.* If, on the one hand, you want to assign the responsibility of exploration to the data structure itself, but on the other hand, do not want to bloat the structure's interface with iteration methods, use Transfold's intermediate stream interface.
- *Polymorphic iteration.* The same transfold object can be used for a number of alternative traversal strategies (e.g., diverse tree exploration orders). By using an abstract **Function** interface for the passed stream processing functions, it is possible to dynamically dispatch on the traversal alternatives.
- *Versatility.* Use Transfold to implement a wealth of operations, for instance for lists: sum, product, length, average, max, min, map, filter, reduce, reverse, copy, append, exists, all, variance, horner[10]. According to [Waters79], 60% of the code in the Fortran Scientific Subroutine Package fits neatly into the maps, filters, and accumulations (i.e., transfold) paradigm.

Do not apply Transfold in case of tight memory and time constraints, where the overhead of an intermediate stream and emulation of lazy evaluation is not tolerable (see also the counter indications to Lazy Object in section 8.4 on page 119): Management of stream elements consumes time. Stream suspensions and lazy function closures represent a memory overhead. In most but a few cases, however, system performance should be absolutely no problem.

10.4 Structure

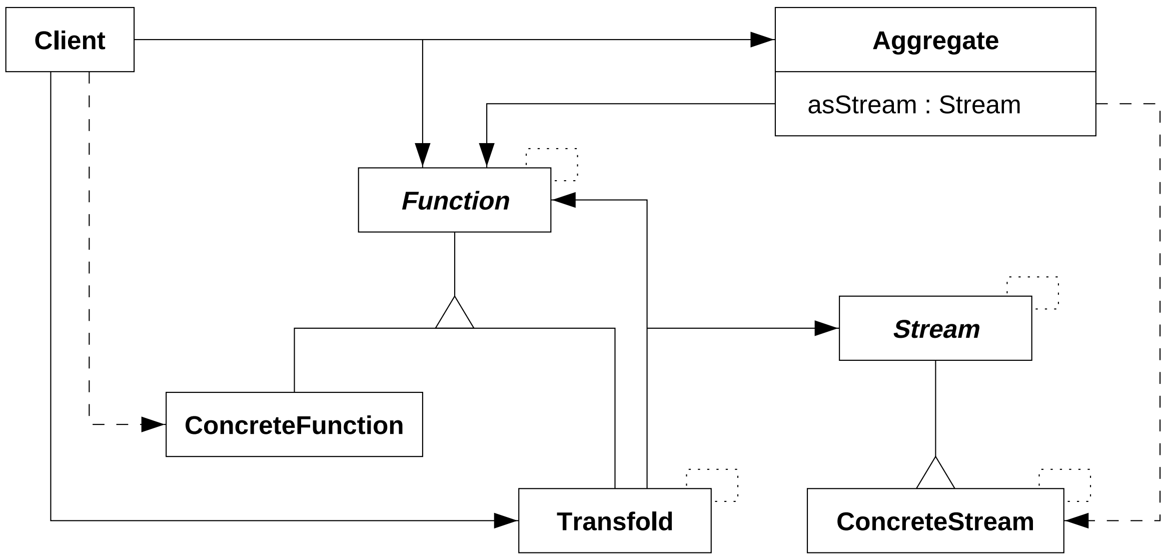


Figure 10.10: Structure diagram

[10]For a list representing a polynomial.

10.5 Participants

- **Client**
  - requests an **Aggregate** to provide a **Stream** of itself.
  - passes two **Function**s and a value as parameters to **Transfold**.
- **Aggregate**
  - provides a **ConcreteStream**, containing a flattened version of itself.
  - uses a lazy **Function** to produce a **ConcreteStream**.
- **Function**
  - provides an application interface for all functions including the stream building function, the **Transfold** parameters, and **Transfold**.
- **Stream**
  - provides an interface to access any concrete streams.
  - implements a lazy, infinite list semantics.
- **Transfold**
  - takes two **Function**s and a value as processing parameters.
  - transforms its input (a **Stream** of **Stream**s) to an arbitrary result type.

10.6 Collaborations

- A client requests an aggregate to flatten itself to a stream.
- The aggregate's asStream method and a lazy function mutually call each other to explore the aggregate lazily, while producing a stream.
- The client uses or creates two function objects, which it passes — along with an initial value — to a transfold object.
- The transfold object lazily accesses the aggregate stream, applying the passed functions accordingly.

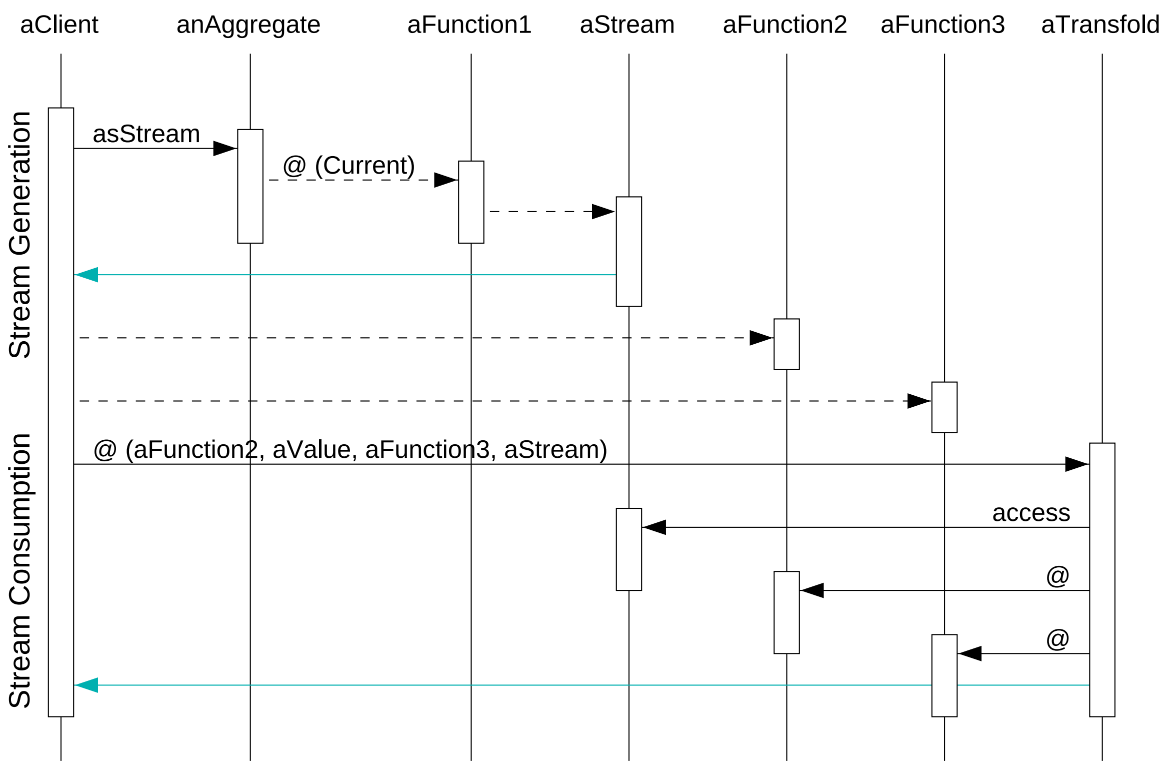


Figure 10.11: Interaction diagram

10.7 Consequences

- *Inverted control.* Clients do not repeatedly call iterator operations, but a transfold repeatedly calls functions supplied by the client. Therefore, a client does not have to provide a control structure to drive the iteration (see section 10.8 on page 180 for a comparison of transfold implemented with external and internal iteration). For the reason that there is only one iteration loop, used by all clients, loop-related errors are much easier to avoid and to discover. If the Mariner-loop (see figure 10.1 on page 166) had been used by another client, tests of that client might have revealed the problem. More time can be spent on the validation of a single loop[11] and any errors are removed for all clients.

- *Traversal alternatives.* Since iteration (stream consumption) is external to structures it is easy to support a variety of traversal strategies and to dynamically dispatch on these. The stream consuming process (a transfold object) is in command of the exploration order, because the stream contains iteration continuations, which may be invoked in any order (see bullet *Separation*).

- *Multiple traversals.* Each transfold manages its own stream consumption progress and, therefore, enables multiple pending iterations on a shared struc-

[11]By referring to "loop" we also include the stream generation processes.

ture. All transfolds share the same exploration stream. Hence, any exploration effort by the structure is beneficial to all consumers. A once explored subpart, does not need to be traversed again due to the call-by-need semantics of streams (see chapter 8 on page 115).

- *Flexibility.* The combination of function object passing, (variable) stream generation, and possibly parallel consumption of multiple streams makes Transfold a truly flexible tool:

  + A particular operation can be performed by just passing function objects, without requiring inheritance or client control structures. See section 10.3 on page 173, *Versatility*, for an impression of operations expressible with folding.
  + Both stream consumption (traversal) and generation strategies (see section 10.8 on page 180, *Stream creation*) are easy to vary.
  + By supporting the parallel processing of multiple structures, the formerly observed big inflexibility problem becomes a small equation: A general operator to compare n structures for equality is

    $$eq \quad = \quad transfold\ (\&\&)\ True\ all\_equal.$$

    Note that the definition of *eq* does not make any assumption about its argument, except requiring elements to be comparable. All functions obtained by purely combining other functions, expose this desirable generic property.

- *Separation.* The Transfold pattern successfully separates a number of concerns:

  - *Termination control.* Both transfold and most importantly the client are in control of iteration advancement and termination. Through the use of lazy stream processing functions, the structure exploration is completely demand driven (see section 10.9 on page 182). When a stream processing function does not evaluate its second argument — e.g., an *And* does not need to examine the second argument, if the first is already *False* — the whole transfold process stops. The same happens, if a stream returned by transfold is not fully consumed. This scheme is far more elegant than letting an iteration function return a `continue?`-flag, as designed in the internal version of the Iterator-pattern [Gamma et al.94].
  - *Exploration & Consumption.* The Transfold pattern is able to to combine the best of both external and internal iteration worlds, by separating the *exploration* of a structure and the subsequent *consumption* of the exploration result.

    + Since iteration (consumption) is defined outside aggregates, their interfaces can be kept small. The only trace of an iteration is a asStream method, which is of general interest anyway (see bullet *Streamable collections*).

+ The responsibility to explore a structure is assigned to the most competent instance, the structure itself. The structure may use all its internal knowledge and recursive calls — thereby memorizing an exploration stack — to perform its exploration.

  It is possible to separate consumption and exploration without having the overhead of a full exploration, because the intermediate stream has lazy semantics, and is lazily produced. Note, that without such an intermediate structure any iteration type, like transfold, would have been defined for each structure separately.

+ Streams work as a *lingua franca*, between aggregates and iterators. As a result, iteration schemes, such as transfold, must be defined only once, for all streamable structures.

  While different stream processing functions are required for differing stream types (e.g., varying in the number of iteration continuations), there is no parallel hierarchy of iterators and data structures. This is achieved by letting structures explore themselves and using streams to uniformly communicate to iteration schemes. Special traversal orders may depend on stream organizations but not on data structures, which is a useful indirection to decrease coupling.

• *Functions.* As mentioned in section 10.2.2 on page 167, a transfold object takes two function objects. This corresponds to a separation of horizontal and vertical processing of the "stream matrix" argument. Again, thanks to laziness, there is no drawback in execution overhead, compared to a single function transfold, since no horizontal processing will take place without being demanded by vertical processing.

  The horizontal processing function was deliberately not constrained to a *fold*, in order to allow the passing of already existing stream processing functions without a need to cast them into the form of a *fold* first. The first function, however, is predefined to be a *fold* to avoid tediously passing a *fold* every time it is needed anyway, to make the user of transfold pass (and think of) a function operating on a stream rather than a stream of a stream, and to increase the utility of transfolding, by making it more specific.

• *Reuse.*

  + Instead of providing an iteration function in a self provided loop body (external iteration), the user of transfold is forced to write a function object, thus, making it available to other clients as well.

  + The developer of a data structure simply provides a way to stream the structure and immediately gets a bunch of iterators (and especially their instantiations to operations (see section 10.3 on page 173, *Versatility*) for free.

- *Streamable Collections.* The asStream method of data structures can also be used for many other purposes, such as a persistence or net-transfer protocol mechanisms.

  Collections may be transfered into each other — streaming a **Bag** to a **Set** is an elegant way to remove duplicates — by means of an intermediate stream. No special mechanisms, e.g., the Serializer pattern [Riehle et al.97], will be needed anymore.

  That also implies that there is a uniform way to construct collections, e.g., from constants. Any collection type, that allows manifest constants in the syntax of a language (e.g., arrays), could be used to be transformed to the desired collection type.

- *High-level mind-set.* A capable, high-level operation like transfold enables to approach problems with a much more powerful decomposition strategy, compared to a procedural paradigm, restricted to e.g., array indexing.

  Timothy Budd tells an anecdote of a FORTRAN programmer, designing a three-level nested loop to find a pattern repetition in a DNA sequence. The competing version of an APL programmer ran much faster, although APL is interpreted, whereas FORTRAN is compiled. This was caused by a difference in algorithm complexity, being $O(M * N^2)$[12] for the FORTRAN program and $O(M * N \log N)$ for the APL program [Budd95]. The difference can be explained by differing programmer mind-sets. The FORTRAN programmer is predetermined to think in terms of loops and array access. The APL programmer used high-level operations like vector to matrix conversion, sorting, and matrix reduction (all akin to and expressible with Transfold). As the anecdote suggests, high-level operations allow approaching problems from a different, valuable perspective. Another supporting example is the task to swap two areas A (ranging from 0 to i) and B (ranging from $i+1$ to n) of different size in a sequence without using any extra storage. A programmer who thinks of sequence reversal as a single operation is more likely to arrive at the very elegant solution $BA = \mathit{reverse}\ (\mathit{reverse}\ A)\ (\mathit{reverse}\ B)$. An implementation will best use a procedural swap-elements reverse approach, but the initial spark to the solution is most likely to be initiated by a high-level mind-set.

- *Choice of style.* In cases where no predefined iteration scheme, like transfold, seems appropriate, it is possible to consume a structure's stream with an external iterator, i.e., to write a control loop which consumes the stream.

- *Robust Iteration.* Transfold does not in particular contribute to the notion of robust iterators. Yet, a so-called *iterator adjustment* scheme [Kofler93], is particular well implementable, since the structure controls its own exploration and, thus, may adjust an exploration process according to element removal

[12] M = pattern length; N = sequence length.

or insertion. Consequently, no registering of active iterators [Kofler93] is necessary.

In cases where updates should have no effect on the iteration process, it is possible to simply iterate on a copy of the structure.

- *Lazy Functions.* A pre-requisite to achieve a demand driven semantics of transfolding is to use lazy, i.e., non-strict, stream processing functions as argument to transfold. For instance, a non-strict *And* with a first argument value of *False*, will cause termination of an exploration process, since the latter is invoked only if *And* evaluates its second argument (see the definition of *fold* in section 10.8). Unfortunately, stream processing functions have to treat their lazy arguments differently. If, for instance, the argument is made lazy by packaging it into a function object, it is necessary to apply the function object to a dummy argument (see section 8.8 on page 122, *Information hiding*). Hence, a standard **Times** function object can not be used, because it assumes its second argument to be a number, rather than a delayed number.

10.8 Implementation

The remarkable separation of iteration and demand driven structure exploration relies on lazy intermediate stream semantics. It is, therefore, vital to use lazy streams (see chapter 8 on page 115) and both lazy generation and consumption functions.

- *Stream consumption.* Whenever we referred to *fold*, we meant *foldr* [Bird86] and not its counterpart *foldl*. The *foldr* function —

$$\begin{aligned} \mathit{foldr}\ f\ a\ [\,] &= a \\ \mathit{foldr}\ f\ a\ (x : xs) &= f\ x\ (\mathit{foldr}\ f\ a\ xs) \end{aligned} \tag{10.9}$$

  — has the nice property that the passed function f controls the recursive call of *foldr*, i.e., determines the extent of structure exploration. An implementation aiming at true laziness must, consequently, wrap the recursive call (the expression in parentheses) into a lazy object, e.g., a function object with a dummy parameter. For the reason that *fold* may produce any result type from a stream, it is unfortunately not possible to design it as a stream pipe like *map*. This underlines the importance of language supported laziness, which would make the difference between strict and non-strict values transparent to clients (see section 14.6 on page 246).

- *Stream creation.* Streams may be created eagerly but typically the structure will return a stream containing iteration continuations. In other words, the structure exploration method will create values but also stream elements, that when accessed invoke further exploration of the structure. As explained in chapter 8 on page 115, this is transparent to the consumer who only sees a sequence of values. Yet, it means that by skipping a stream element, a subpart exploration will not be executed until and only if the skipped element is accessed later. The structure exploration method and the exploration function it uses for stream generation will, therefore, call themselves in a mutual recursion scheme.

By steering the evaluation order of stream elements the client may implement a variety of traversal orders but if, for instance, a stream of a list starts with the first element and provides only forward continuations, then a backward iteration can not be achieved (see figure 10.2 on page 168, defining the locations of main traversal strategy and subsequent order commitment). As a result, the structure should provided a parameterized asStreamWithFunction method. With the passed function any exploration order may be achieved. The standard asStream method should satisfy the most frequent needs, though.

- *Keyword parameters.* Two important settings for transfold's arguments are cons and the empty list for the vertical processing and the identity function for the horizontal processing respectively, since their net effect is the identity function (leaving transfold to simply transpose its argument). Using these as defaults, keyword parameters allow passing just one and do not bother about supplying the default case for the other. For instance,

  ```
  Result:=transfold.foldWith(count, 0) @ yss
  ```

  (counting rows without row processing) or

  ```
  transfold.mapWith(innerProduct @ xs) @ yss.
  ```

  (transforming rows into inner product results without subsequent reduction).

- *Iterator usage.* The purpose of this pattern description was to present the mechanics of Transfold. This is why the client had the burden to ask the aggregate for a stream and feed it into a transfold. It would be more convenient to call a client method that automatically sets up an appropriate transfold object and returns the result. Using a C^{++} template method this is a feasible approach. With EIFFEL, however, there is a problem how to specify the type of the return value. The type depends on the functions passed passed to transfold and it is unfortunately not possible[13] to express that dependency in the method's signature. The usual way out is to use the same generic variable for constraint values. However, this implies to add such generics to the aggregate class, since it contains the method that must enforce the constraint. Without debate, this presents a true kludge and the best one can achieve with EIFFEL is a convenience function, that takes transfold's parameters (except the stream matrix) plus an aggregate, and then applies transfold to a streamed version of the aggregate.

[13]Not even with anchored types.

10.9 Sample Code

To calculate the inner product of a matrix (see figure 10.7 on page 172) we code the inner product product operation (see definition 10.8 on page 172) in EIFFEL as:

```
...
local
  ip : Function [Stream [Stream [Integer]], Double]
...
  ip:=transfold @ plus @ 0.00 @ (fold @ times @ 1.0);
...
```

We use the same type progression (integer, real, double) for the intermediate results as in figure 10.7 on page 172. Hence, `times` and `plus` must promote from integer to real and real to double respectively, but — apart from that — are standard function objects. In case of a fully lazy treatment — which we omit here for the sake of clarity — `times` and `plus` must respect the laziness of their second argument.

The new function `ip` can be applied to a matrix, so presuming

```
vec1, vec2, vec3 : Stream [Integer];
vecs             : Stream [Stream [Integer]];
...
vec1:=cons @ 8 @ (cons @ 1 @ fromconst (6));
vec2:=cons @ 3 @ (cons @ 5 @ fromconst (7));
vec3:=cons @ 4 @ (cons @ 9 @ fromconst (2));

vecs:=conss @ vec1 @ (conss @
              vec2 @ (conss @ vec3 @ void));
```

— note that `cons` produces an integer stream, while `conss` produces a stream of an integer streams — the examples

```
io.putdouble (ip @ vecs);

io.putdouble (ip @ vecs.tail);
```

will produce the output as given in figure 10.12.

$$225 = \mathrm{IP} \mathbin{@} \texttt{vecs} \left\{ \begin{array}{l} \begin{array}{ccc} 8 & 1 & 6 \end{array} \\ \left. \begin{array}{ccc} 3 & 5 & 7 \\ 4 & 9 & 2 \end{array} \right\} \mathrm{IP} \mathbin{@} \texttt{vecs.tail} = 71 \end{array} \right.$$

Figure 10.12: Inner product application

The horizontal processing function is a *fold* declared as

```
fold : expanded Fold[Integer, Real].
```

The use of `expanded` is an idiom to save the otherwise necessary explicit attachment of an object to `fold` (see section 7.9 on page 105, *Creation*). Apart from the argument-collection classes, `Fold` is implemented with

```
class Fold2[E, A]
inherit Function[Stream[E], A];

creation make

feature
  func : Function[E, Function[A,A]];
  init : A;

  make(i : like init; f : like func) is
  do
    init:=i;
    func:=f;
  end;

  infix "@" (stream : Stream[E]) : A is
  local fold : expanded Fold[E, A];
  do
    if stream=void then
      Result:=init;
    else
      Result:=func @ stream.item @
              (fold @ func @ init @ stream.tail);
    end;
  end;

end
```

and, thus, shares a striking similarity to its functional counterpart (see definition 10.9 on page 180).

With the help of `fold` and `map` (which was established in chapter 8 8.10 on page 143), the class implementing the body of transfold is quite straightforward:

```
class   TransFold3[E, I, A]
inherit Function[Stream[Stream[E]], A]
        StreamUtility[E]
creation make

feature
```

```
      foldFunc : Function[I, Function[A, A]];
      init     : A;
      mapFunc  : Function[Stream[E], I];

      make(f : like foldFunc; i : like init;
           m : like mapFunc) is
      do
        foldFunc:=f;
        init:=i;
        mapFunc:=m;
      end;

      infix "@" (streams : Stream[Stream[E]]) : A is
      local
        map  : expanded Map[Stream[E], I];
        fold : expanded Fold[I, A];
      do
        Result:=fold @ foldFunc @ init @
                ((map @ mapFunc) @ transpose(streams));
      end
    ...
```

The application method directly corresponds to the functional transfold (see definition 10.3 on page 171). I intentionally left out the `transpose` method of class `TransFold3`, because it nicely demonstrates the difference between a high-level, functional, internal iteration style —

```
    transpose(streams : Stream[Stream[E]]) : like streams is
    ...
    do
      newRow:=mapToHeads @ head @ streams;
      tails:=mapToTails @ tail @ streams;

      if (fold @ oneEmpty @ False @ tails) then
        tails:=void;
      else
        tails:=transpose(tails);
      end;

      Result:=consS @ newRow @ (tails);
    end
```

— and its "von Neuman[14]", procedural, external iteration style:

[14]Referring to the critique of John Backus towards a "von Neumann [programming] Style" [Backus78].

```
transpose(streams : Stream[Stream[E]]) : like streams is
...
do
  from
    rows:=streams;
    !!rowArray.make(1, 7);
  until rows=void
  loop
    rowCount:=rowCount+1;
    rowArray.force(rows.item, rowCount);

    rows:=rows.tail;
  end;

  from
  until shortestEnds
  loop
    from
      row:=rowCount;
      newRow:=void;
    until row=0
    loop
      newRow:=cons @ (rowArray @ row).item @ newRow;
      rowArray.put((rowArray @ row).tail, row);
      shortestEnds:=shortestEnds or
                     ((rowArray @ row) = void);
      row:=row-1;
    end;

    Result:=consS @ newRow @ Result;
  end;
end
```

The latter version uses nested loops to construct new rows by repeatedly collecting heads and checking for the shortest row. The functional version (implementing definition 10.2 on page 169) replaces this nesting with normal mapping and a recursive call.

The procedural version uses an array to memorize the respective positions of the partially consumed rows. The functional version avoids this by transforming the argument matrix into the desired residual matrix.

It is noteworthy to observe that the procedural version features one loop counting downward, rather than upwards to get the element order right. No such detail must be respected in the functional version[15].

[15]Admittedly, on other occasions one has to think about whether to add to the front or to the back of a stream.

Also, although both versions do not transpose matrices with an infinite number of rows (infinite rows are handled), the functional version is easier to extend towards this property. First, because it is simpler and easier to understand. Second, because the recursive call to `transpose` exactly defines the extension hot spot: The strict recursive call must be replaced by a lazy stream generation, using the framework of **StreamFunc**s, presented in chapter 8 on page 115.

A final example, hinting at the expressiveness of transfold (see section 10.10 on the next page for a comparison with APL), shall be the implementation of matrix multiplication with transfold. To show the result of the multiplication of a *magic square* with another matrix, as depicted in figure 10.13,

$$\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 9 & 14 & 7 \\ 8 & 10 & 12 \\ 13 & 6 & 11 \end{bmatrix}$$

Figure 10.13: Matrix multiplication with a magic square

one simply calls

```
showMatrix(matMult(vecs, rows));
```

using

```
matMul(xss, yss : Stream[Stream[Integer]]) : like xss is
local
  transMapInner : expanded TransMapInner[Integer];
  map : expanded Map[Stream[Integer], Stream[Integer]];
do
  Result:=map @ (transMapInner @ yss) @ xss;
end;
```

We exploit the fact that the elements of the result matrix are defined by

$$z_{ij} = inner\_product\, x_i\, y_j^t,$$

i.e., the inner products of all rows of the left matrix x with all columns of the right matrix y determine the result matrix.

To implement this, we process each row of the left matrix with a function `transMapInner`, that applies the inner product operation to its argument row of the left matrix with each column in the right matrix, i.e., the body of `transMapInner` is:

```
  Result:=transFold.mapWith(applyInner @ xs) @ yss;
```

Each row (`xs`) of the left matrix is transfolded with the right matrix, i.e., combined with each column of the right matrix, and the inner product operation is applied to each such pair.

The `applyInner` function simply constructs a two row matrix out of its two stream arguments and applies the inner product operation to it:

```
...
ip:=transFoldInner @ plus @ number.zero @
                     (fold @ times @ number.one);
Result:=ip @ (cons @ xs @ (cons @ ys @ void));
```

The use of `number.zero`[16] achieves that the function is generic, since either `0`, or `0.0` would create a type-mismatch with the wrong generic instantiation of matrix multiplication[17].

Summarizing, matrix implementation was implemented by mapping, and two transfolds, one realizing the inner product operation.

10.10 Known Uses

Although C++ usually promotes external iteration, there is an example of an internal iterator (`foreach`) interface in the Borland C++ libraries [Borland94]. Since it builds on passing function pointers, it must use an extra, unsafe `void` type for passing parameters, though.

Folding is commonplace in functional programming [Bird & Wadler88], but also used in SCHEME [Abelson & Sussman87] and available in the SMALLTALK library [LaLonde94]. It is not a frequently used operation in SMALLTALK [Smith95], which can be attributed to the peculiar name (inject: into:), but also to the unfamiliarity of most SMALLTALK users with the nature of that operation. The SMALLTALK collection library even contains a with: do: method, allowing to iterate two structures in parallel, which represents a special case of transfolding. SMALLTALK also uses streams to efficiently implement concatenation of collections. The caching effect of streams avoids to repeatedly generate the prefix of sequenced concatenations like `coll1 + coll2 + coll3 + coll4 + ...`.

APL [Harms & Zabinski77] is well-known for its high-level operations on vectors, matrices, and structures of even higher dimension. Three of its four primitive extension operators[18], reduction (`f`/A), scan (`f`\\A), and innerProduct (A`f.g`B), can directly be expressed with *transfold*. The fourth, outerProduct ($A\circ$`.f`B), is expressible with a combination of *transfold* and *map*. The matrix multiplication example from section 10.9 on page 182, builds an outer product of the matrices' rows and columns, and then reduces it with an inner product.

[16]`number` is a variable declared to be of the generic matrix element type.

[17]Although an escape was possible, one would wish that simply `0` has had sufficed.

[18]Extend an operation to a collection.

10.11 Related Patterns

10.11.1 Categorization

Function Object: Transfold's interface to clients is that of a Function Object.

Iterator: Transfold realizes internal iteration, whereas the Iterator pattern uses external iteration [Gamma et al.94]. There is also an internal variant of the Iterator pattern [Gamma et al.94], but it uses inheritance for iterator and action combination and exhibits the normal one-structure-iteration inflexibility.

Bridge: The stream between structure and iterator is like a Bridge [Gamma et al.94], separating a consumption interface from the implementation of a structure (including the exploration implementation).

Serializer: The use of an intermediate stream resulting from structure exploration is akin to the Serializer pattern [Riehle et al.97], whose purpose is to flatten structures for persistence.

10.11.2 Collaboration

Lazy Object: Transfold is very well suited to transform one lazy structure into another.

Value Object,
Composite: Iterators are often applied to recursive (possibly value) structures like **Composite**. Furthermore, Composite [Gamma et al.94] may be implemented with Transfold to enumerate its children.

Translator,
Visitor: Structure consuming patterns may defer their structure exploration to Transfold.

Chain of Responsibility: An event may be transfolded over a collection of handlers to find the most appropriate handler.

Observer: An Observer [Gamma et al.94] may use Transfold to iterate over its dependents-collection.

Keyword Parameter: Transfold may use the keyword parameter variant of Function Object (see section 7.10 on page 107) to ease supplement of its parameters.

10.11.3 Implementation

Lazy Object: Transfold uses a lazy stream as well as lazy generation and consumption functions.

Void Value: A void value may be used to define the base case of the exploration process.

Function Object,
Strategy: A structure may use a Strategy [Gamma et al.94] or a Function Object to allow variation of its exploration process. Function Object is used to provide closures for function parameter passing and is also employed to implement Transfold's functionality.

11 Void Value

I'm not completely worthless! I can still serve as a bad example.
– Mark Twain

11.1 Intent

Raise Nil to a first-class value. This allows to treat void and non-void data uniformly, is a way to provide default behavior, facilitates the definition of recursive methods, and enables to deal with error situations more gracefully.

11.2 Motivation

Classes are often regarded as representing abstract data types (ADTs). Typically, the *empty* constructor of an ADT is translated to Nil. For instance, an empty **BinaryTree** is represented by Nil and a tree leaf is represented by setting both child attributes to Nil. It is tempting to identify *empty* constructors with the special pointer content Nil, since there are some reasons in favor of it:

- Operations are often partial due to *empty*, well accounted for by Nil.
- Nil does not waste any storage.
- Most languages initialize references to Nil, i.e., provide the empty case by default.

It is, for instance, suggested to represent an unknown book author with a void reference [Meyer97].

11.2.1 Problem

However, such a design decision puts a great burden on the clients of the ADT. Before the client can apply operations on a tree, it must check whether it is void or not. Otherwise, invoking a method through a Nil reference would produce a runtime-exception or even -error. Typical (EIFFEL) code[1] is depicted at the upper part of figure 11.1 on the next page.

[1] All code examples in this chapter have been extracted from a commercially available library; identifiers have been adapted for presentation, though.

```
Result:=(tree /= Void)     if tree /= Void then
        and then             io.putstring(tree.out);
        tree.has(v);
------------------------------------------------------
Result:=tree.has(v);       io.putstring(tree.out);
```

Figure 11.1: Code example: Robust behavior

11.2.2 Solution

The code in the upper part of figure 11.1 can be replaced with the code at the lower part of figure 11.1, if we abandon Nil (Void) and replace it with an extra class (**VoidTree**) (see figure 11.2).

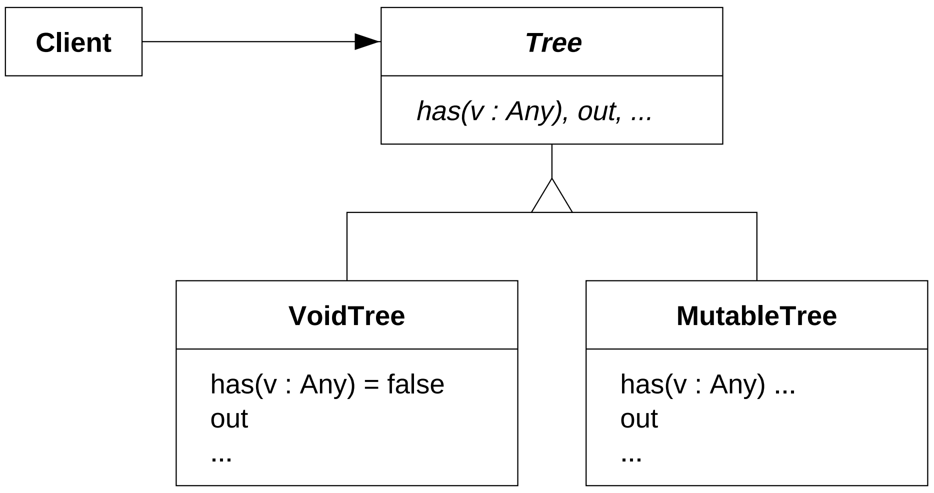


Figure 11.2: Empty tree as void value

Now we can define **VoidTree** to return false on a has message and to produce no output on out. In fact, we applied the general pattern to replace case analysis (testing for Nil) with dynamic binding (provision of an extra type constructor **Void-Value**). As a result, many such if statements as above can be removed from client code.

An important special case of robust behavior is a Null iterator (also see Iterator [Gamma et al.94]). Instead of external iteration (upper part of figure 11.3 on the next page) one should use internal iteration, e.g., and let void sets do nothing on iteration (lower part of figure 11.3 on the facing page).

Answering queries to void values with default values again allows replacing the upper part of figure 11.4 on the next page with its lower part. This obviously improves the treatment of trees from a client's perspective, but there is more to gain. Let us count the number of nodes in a tree.

```
if set /= Void then
```

    loop over all set elements

```
setIterator.do_all
```

Figure 11.3: Code example: Null iterator

```
if tree /= Void then
  maxPlane:=tree.maxPlane
else
  maxPlane:=DefaultMaxPlane
end
```

```
maxPlane:=tree.maxPlane
```

Figure 11.4: Code example: Default Behavior

The recursive calls in the upper part of figure 11.5 must be guarded with if statements when tree leafs are represented with Nil. If void tree values return 0 as their count, then the code becomes much more readable and better expresses the function's spirit (see lower part of figure 11.5).

```
Result:=1;
if left /= void then
  Result:=Result+left.count;
end
if right /= void then
  Result:=Result+right.count;
end
```

```
Result:=1+left.count+right.count;
```

Figure 11.5: Code example: Base case definition

The provision of base cases by void values is a special case of the general scheme to distribute cases. Complicated nesting, like depicted in the upper part of figure 11.6 on the following page becomes much clearer code when distributed to **VoidAny** and **AnyObject** class definitions. Note that the use of `obj.is_void` is different to `obj = Void`. The former tests for voidness as well, but allows the representations for void objects to be changed. This is useful for multiple void values as well as for standard objects which just happen to be in an empty or void state.

```
equal(me: ANY; you: ANY): ...
Result:=(me=Void and you=Void)
or else ((me/=Void and you/=Void)
         and then me.is_equal(you))
```

```
Result:=you.is_void      (in class VoidAny)
Result:=is_equal(you)    (in class AnyObject)
```

Figure 11.6: Code example: Case distribution

It is worth mentioning, that we may provide void trees, -nodes, and -leafs independently with possibly differing behavior. See section 11.10 on page 199 for an analogous example when this is useful. Also note that while void values are a natural concept for container classes their applicability extends to all types of classes. For instance, the subordinates method of a **VoidEmployee** can return an empty list. Its salary method may create a no-valid-employee exception, return 0, or a **VoidSalary** instance, depending on the desired semantics. All three possibilities may coexist in separate void values. Application parts can individually choose to inject the appropriate void value into server calculations.

11.3 Applicability

- *Uniformity.* Clients benefit from Void Value by uniformly treating void and non-void data. Inquiring information and invoking behavior can be done independently of the data's void state, without the eventual need for creating initial instances first.
- *Default behavior.* Void values allow the specification of default behavior. In contrast to Nil, a void value may specify results and behavior for any of its interface methods. This is useful for providing reasonable behavior for uninitialized data and void result values.
- *Error handling.* A void value can be used just like an exceptional value [Cunningham94] when most of the code should not deal with error situations. An exceptional value may either behave properly indicating an error situation (e.g., by describing the error when by default displayed to the user) or can be caught with error checking code by the application's top-level code.
- *Termination.* Use Void Value to relieve recursive definitions from testing the base case. A recursive call can then be made regardless of possibly void arguments. Accordingly, the definition for the base case can be given at the base values (i.e., void values) instead of one step before. Here, Void Value plays the role of a first-class terminator.

11.4 Structure

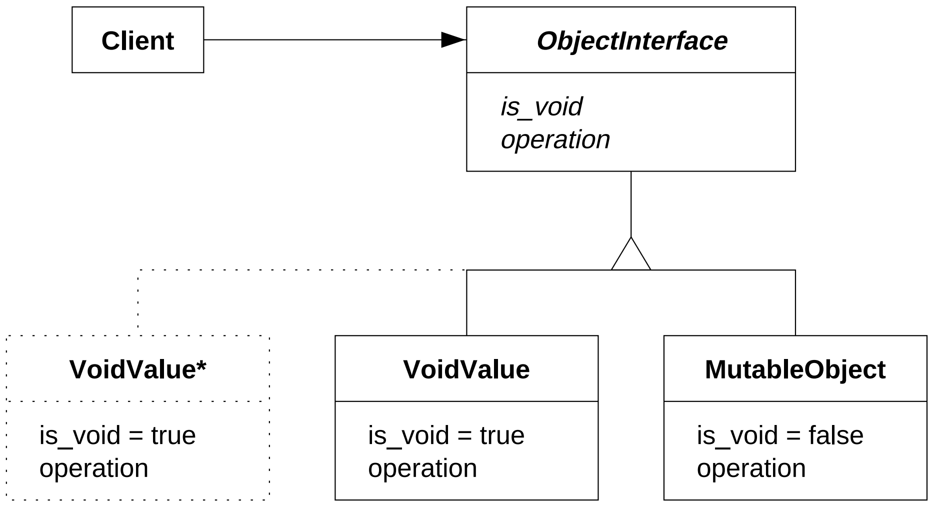


Figure 11.7: Void Value structure diagram

11.5 Participants

- **Client**
  - accesses both **MutableObject** and **VoidValue** through **ObjectInterface**.
- **ObjectInterface**
  - provides the common interface for **MutableObject** and **VoidValue**.
  - may provide common abstract[2] behavior to **MutableObject** and **VoidValue**.
- **MutableObject**
  - defines the standard object behavior.
  - introduces attributes for object state.
  - does not care for recursion base cases, default- and error behavior.
- **VoidValue**
  - replaces Nil.
  - defines base cases for recursion, default- and error behavior.

[2]Concrete methods do not rely on state (attributes), but on an abstract interface, only.

11.6 Collaborations

- Clients use the **ObjectInterface** to manipulate void, non-void, default, and error objects. If **MutableObject** extends the interface of **ObjectInterface**, clients may maintain specialized references to **MutableObject**, provided they are guaranteed to receive objects of type **MutableObject** only.

11.7 Consequences

- *Abstraction.* Void Value abstracts from the implementation detail to represent void or uninitialized data with Nil. Clients, therefore, are relieved of the need to treat data with Nil differently from data which does not use Nil for its representation.

- *Object-Orientation.* Checking for Nil, i.e., case analysis, is replaced with dynamic binding. This moves all case analysis from the program to a single distinction between normal *object* and void *value*. Object behavior must be defined at two places (object and void value). One the one hand, this allows combining several void values with a single object. On the other hand, changing a single method might mean to change two class definitions (object and value class).

- *Efficiency.* It is more storage friendly to use void values, instead of full blown objects that simply have the state of being empty or uninitialized, but carry the overhead of unused attributes. Note, however, that void values are not as easily turned into objects again without support for “Implicit creation” (see sections 11.8 on the facing page & 14.7 on page 248).

  Depending on the implementation of dynamic binding a calling overhead may be incurred, compared to if statements. Note, however, that the traditional solution at the upper part of figure 11.5 on page 193, unnecessarily checks $\frac{arity^{depth}-1}{arity-1}$ nodes for being void, in addition to the following `count` call.

- *Immutability.* If void values should not need more storage than Nil, they have to be immutable. Mutable objects without attributes just claim code space for their methods once, and possibly a little storage for housekeeping mechanisms such as RTTI (Run Time Type Identification) and object identification.

- *Separation.* Program code relying on Void Value describes the standard case only. In analogy to language support for exception handling, framework code or algorithm descriptions thus become easier to write and read (see code examples). Handling of errors, denoted by void values, can be done in a few places at the application’s top level. Lower level code can be designed to smoothly pass error values around, until they are identified by the top level.

- *Implicit initialization.* In order to avoid any occurrences of Nil, one must tie the creation of objects to the declaration of references. See section 11.8 for suggestions how to automatically achieve initialization to void values.

- *Initialization errors.* Exceptions, caused by the application of operations on Nil, are dealt with more gracefully. Instead of triggering an exception or halting the system a possibly inappropriate behavior is executed, but users are still able to continue the application. The downside of this is that a missing initialization might go unnoticed, only to be discovered as unexpected application behavior very much later in the execution. According to the experiences of the UFO project [Sargeant93], however, unnoticed missing initialization has not been found to be a problem in practice. Additionally, it is almost as easy to pass Nil around undetected. Error messages simply become more meaningful with void values [Sargeant96b]. If the void value is one among several alternatives for a type, its identity allows tracing back its origin.

  Yet, care should be taken to never allow the misinterpretation of an erroneous value to be a valid result. Calculations must preserve the exceptional status of void values to prevent the masking of errors.

11.8 Implementation

- *Interface extension.* In analogy to the Composite pattern [Gamma et al.94], **MutableObject** might provide more operations than **ObjectInterface**, in order to provide operations that make no sense to **VoidValue**. Each such extension, however, will minimize the applicability of **VoidValue** to play the role of an error value. Clients using the extended **MutableObject** interface, will not be able to transparently work on void values too. In this context, it is better to have the full interface at **ObjectInterface** and to define exception (e.g., error reporting) methods in **VoidValue** respectively.

- *Storage requirements.* Any behavior that goes to **ObjectInterface** should rely on abstract state only, i.e. “Implement behavior with abstract state” [Auer94]. Any attributes will incur a space penalty on **VoidValue**.

- *Value initialization.* If representation of a complex constant (requiring creation effort) is more important than preserving minimal space requirements, then a void value may calculate the constant information (possibly taking creation arguments) and store it in attributes. Accordingly, **VoidValue** is best implemented as a Singleton [Gamma et al.94] in order to allow sharing of the constant data.

- *Multiple inheritance.* In statically typed languages **VoidValue** needs to multiply inherit from its interface class and another void value class, if code reuse is desired between void value classes. One of the inheritance paths is used only for interface inheritance though, unless the interface classes implement common behavior for **VoidValue** and **MutableObject**. In the latter case the language should properly support repeated inheritance of code (e.g., as in EIFFEL).

- *Reference initialization.* The value to be gained from Void Value partly depends on how thoroughly we can eliminate any Nil values from program execution. To this end, one may replace standard references with *default-references,* akin to smart references [Edelson92].

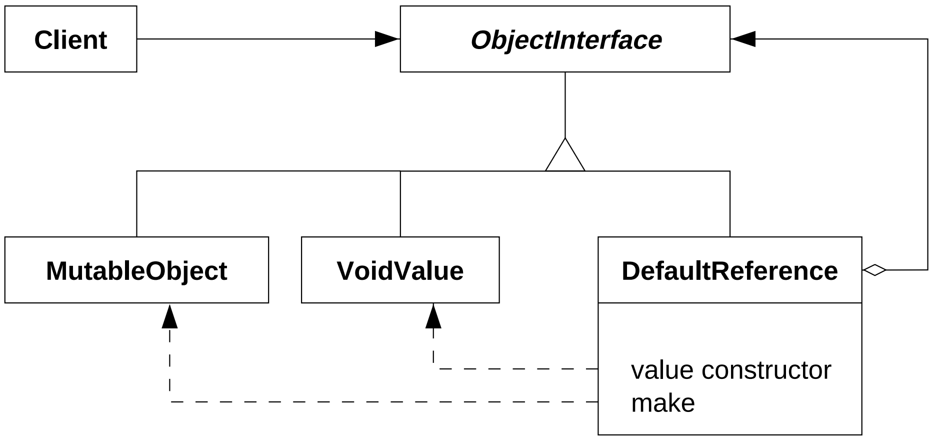


Figure 11.8: Automatic reference initialization

A **DefaultReference** (see figure 11.8) is a value (e.g., non-reference type in C++, expanded type in EIFFEL, part-object in Beta, etc.). Consequently, it is initialized by declaration, i.e., cannot take the value Nil. After initialization it delegates any message to **VoidValue** by default. Hence, any usage of **DefaultReference** without prior initialization results in applications to **VoidValue**, instead of Nil. Ultimately, **DefaultReference** will be initialized (e.g., using a method called make) to reference a **MutableObject**. In the structure of figure 11.8 **DefaultReference** plays the role of an **ObjectProxy**. Implementation issues, such as using overloading of the member access operator (->) in C++, are discussed in the implementation section of Proxy [Gamma et al.94]. Note that using a **DefaultReference** involves a delegation overhead for each access to either **VoidValue** or **MutableObject**. This might be a price too high to pay just for gaining automatic initialization of references.

Of course, we can achieve automatic initialization of **MutableObject** by directly declaring it to be a value, resulting in a much simpler structure and no delegation overhead. Yet, this is possible only if value semantics (e.g., no sharing, no efficient updates) is intended [MacLennan82].

- *Implicit creation.* SMALLTALK allows changing the type of an object without affecting its identity. Ergo, a void value may change to an object and vice versa. If a void value cannot deal with a message it may delegate it to its associated object and then become:[3] the object itself. Hence, there is no need for explicitly creating objects anymore, since void values can be made responsible for this. In the absence of a become mechanism, implicit creation can still be achieved by "Reference Initialization" (see above).

11.9 Known Uses

NoController, **NullDragMode**, **NullInputManager**, **NullScope** are classes in the class hierarchy of VISUALWORKS SMALLTALK that are used as void values [Smith95].

11.10 Related Patterns

11.10.1 Categorization

Composite: Both Composite and Void Value provide uniform access to components (value & object) of a structure (type). A **VoidValue** plays the role of a special immutable component leaf. As a difference, Void Value does not involve issues of child management.

Value Object,
Constant Object: A void value "is-a" value object, in that it does not offer mutator operations. As value objects, void values lend themselves to be implemented with Identity Object (see next bullet and section 9.11.1 on page 161).

Singleton: A void value is a Singleton [Gamma et al.94], in that it can be shared by all objects of a type. As it is immutable, there is no need to have multiple instances.

Flyweight: **VoidValue** can be regarded as a **ConcreteFlyweight**, as it holds intrinsic data and is shared among all clients that use it.

State: When used as described in *Implicit creation* at section 11.8 on page 197, **VoidValue** and **MutableObject** play the role of **ConcreteStates**, representing the void and non-void states of data.

11.10.2 Collaboration

Lazy Object: Pattern Lazy Object (see chapter 8 on page 115) may be used to lazily initialize void value components.

Command: Void Command may stand for default or idle behavior.

[3] SMALLTALK's method name to replace objects.

Template Method: While *subclass responsibility* [Goldberg & Robson83] normally requires at least one concrete subclass, Void Value can be used as a default subclass, allowing the execution of abstract methods. Of course, Void Value is just a special concrete subclass, so this collaboration is a lot more dramatic with language support for void values (see section 14.7 on page 248).

Memento: A Void Memento can be used as a default and reset state.

Factory Method,
Abstract Factory: A void value can be used as a return value for not available products, e.g., unsupported widget types with regard to a specific window factory.

Iterator, Visitor,
Chain of Responsibility: All these patterns are based on iteration and Void Value can help to define their termination behavior, analogous to a base case in recursion.

State,
Strategy: Void State may represent the initial or an error state. While used like Void State, Void Strategy would allow **StrategyContext** creation without a Strategy selection.

11.10.3 Implementation

Proxy: **DefaultReference** (see figure 11.8 on page 198) works as a Proxy since it keeps a reference to an object whose interface it shares.

Bridge: **DefaultReference** behaves like a bridge, in that it shields the clients from the two “implementations” **VoidValue** and **MutableObject**.

12 Translator

The art of programming is the art of organizing complexity.
– Edsger W. Dijkstra

12.1 Intent

Add semantics to structures with heterogeneous elements without changing the elements. Separate interpretations from each other and use local interpretations that allow for incremental reevaluation.

12.2 Motivation

Many operations on data structures can be viewed as homomorphisms, that is, as structure preserving mappings from one domain into another. For instance, compilers typically map the abstract syntax of the source language into a specific machine code language[1]. Other kinds of abstract interpretations (e.g., pretty-printing and type-checking) should be expressed as homomorphisms between source and target domain as well. The reason for this recommendation can be explained by means of an equation that holds, if a homomorphic relationship between two structures exists:

$$\phi(op(a,b)) = op'(\phi(a),\phi(b)) \tag{12.1}$$

An interpretation ϕ on an operation *op* (from a source domain) with subcomponents a and b is defined as a new operation *op*′ (from a target domain) whose subcomponents are determined by again applying ϕ to a and b [Wechler92]. An instance of this general equation for a compiler is, e.g.:

```
compile(assign(lhs, rhs)) =
store(compile(lhs), compile(rhs))
```

Note how in the above equations an interpretation is shifted down from operators down to operands. Also, the right hand side of the equations has a structure that allows us to account for incremental modifications to the source structure. In case of changing the left-hand-side (`lhs`) of `assign`, there is no need to rebuild the

[1]Historically, homomorphisms are closely connected to syntax-directed translations [Aho et al.86] and correspond to compositional definitions [Nielson & Nielson93].

whole result term. One simply has to apply `compile` to the changed `lhs` and plug the result into the first operand of `store`.

12.2.1 Problem

Consider a programming environment that represents programs as abstract syntax trees. It will need to perform various interpretations on the abstract syntax tree like type-checking, code generation, and pretty-printing. Figure 12.1 depicts two sample transformations.

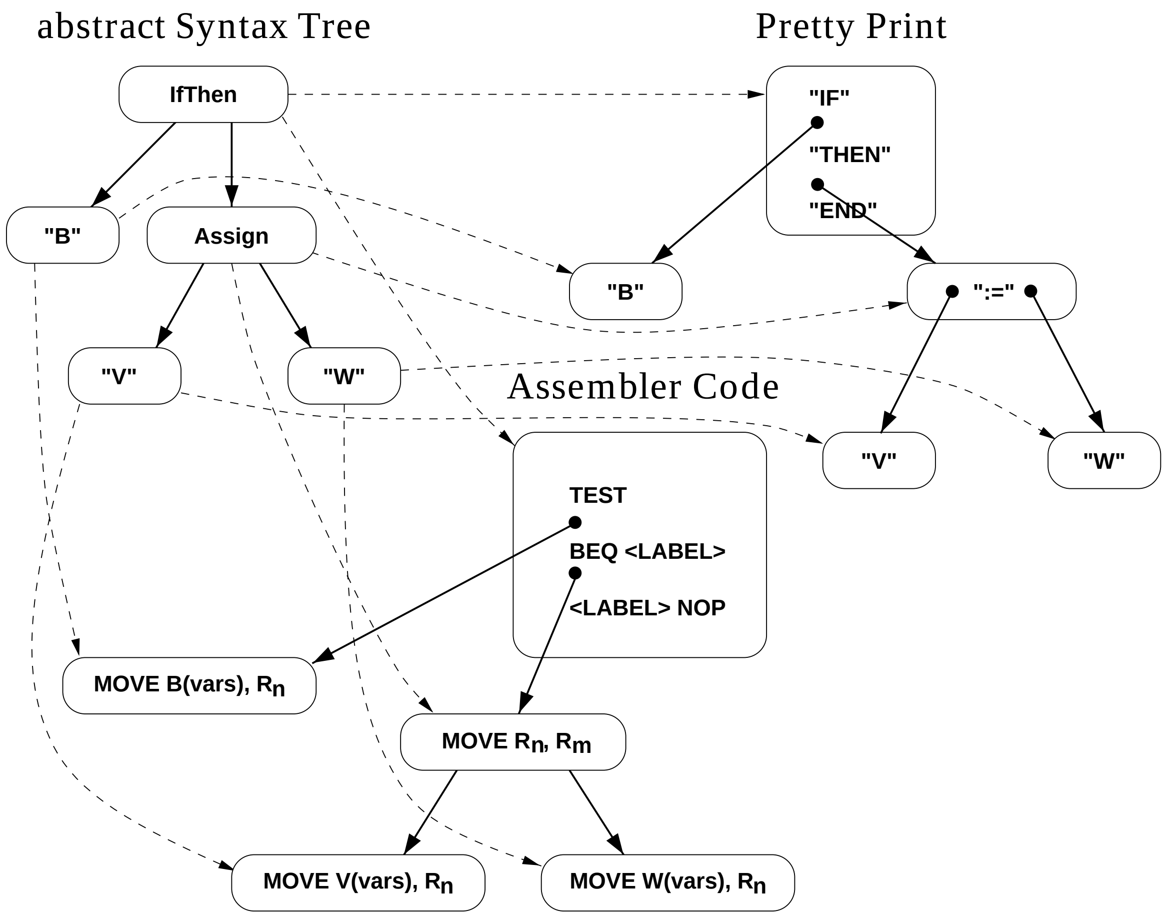


Figure 12.1: Homomorphic translations of trees

The result of a mapping (dashed arrows in figure 12.1) depends on the interpretation (e.g., compilation) and concrete node type (e.g., assign) involved. One may put all various interpretations (type-check, pretty-print, etc.) into the node interface in order to rely on dynamic binding. However, this is often not a good idea:

- It leads to a system that is hard to understand, maintain, and change.
- Adding a new interpretation means changing and recompiling all node types.

- An interpretation cannot be added without changing the node interface.
- The interface of nodes will grow until it becomes bloated.

The first two arguments are also addressed by the Visitor pattern [Gamma et al.94]). Visitor also addresses the problem of adding functionality to each node-type (represented by a class) in a conceptual hierarchy (e.g., abstract syntax, construction data, etc.) but does not aim at incrementality and demands node-types to know about external interpretations (see section 12.10 on page 217).

The last two arguments of the above list especially apply to data structures other than abstract syntax trees. Consider a data structure that represent the logical structure of a building. It is probably only well after designing the interface to that structure that one wishes to perform some interpretation like computing the total rent income. In this context, it is useful to differentiate between intrinsic properties (e.g., nodes have descendents) and extrinsic properties (e.g., pretty-print). There is no end to extrinsic properties and it does not make sense to lump all of them into one interface.

Now, if we provide interpretations as external features we are facing a problem with an implementation language that provides single-dispatch only[2]. As already mentioned, the code to be executed for each node when we traverse an abstract syntax tree depends on two variabilities:

```
find-implementation(node-type, interpretation)
```

Note that we already rejected `node-type.interpretation` with the argumentation above. The reverse, `interpretation.node-type`, does not make sense, since unlike the interpretation type the node type always changes during tree traversal; that is, dispatch isn't required for the receiver but for the argument.

12.2.2 Solution

What we need is double-dispatch on both `node-type` and `interpretation`. Fortunately, there are ways to emulate double-dispatch and its generalization multi-dispatch, with a single-dispatch language. We opt for a solution which can be characterized as external polymorphism (see section 12.10 on page 217 for Visitor type double-dispatch). Unlike Cleeland et al., however, we do not use a combination of C^{++}templates, Adapter, and Decorator [Cleeland et al.96]. We simply use the natural notion of a generic function [Kühne97].

When a generic function object is applied to a node, it determines the node's type (by using runtime type information), creates the corresponding specialized function object, and returns the result of applying the specialized function object to the node.

Figure 12.2 on the next page depicts how concrete element types (e.g., **IfThen**) induce the creation of their corresponding specialized functions. A specialized function knows the exact type of its argument and, therefore, can appropriately exploit the argument's full interface.

[2]Languages with multi-dispatch, e.g., CLOS, Cecil, or Dylan are not in widespread use.

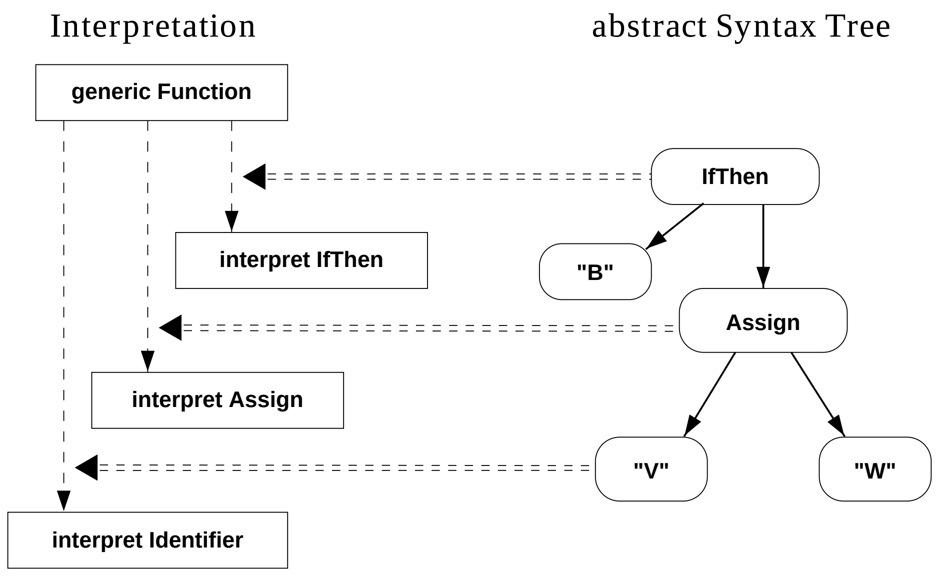


Figure 12.2: Generic interpretation on an abstract syntax tree

Note that it is not only natural to deal with generic functions to achieve double-dispatch, but also very natural to employ functions for translations. The approach of formally defining the semantics of a programming language called denotational semantics is entirely based on semantic functions, i.e., functions that transform phrases into denotations [Schmidt86].

Figure 12.3 on the facing page shows the structure diagram that corresponds to the domains used in figure 12.1 on page 202. Only relationships relevant to Translator have been included. For instance, language nodes like **ToyIf** will typically have an aggregation relation with **ToyLang** which is not shown here. Exclamation marks denote places of possible extension (see section 12.7 on page 209, *Extensibility*).

Class **Language** in figure 12.3 on the next page is not required in general (see figure 12.4 on page 206). Also, it is not required that **ToyIf**, **ToyAss**, etc. have a common ancestor (like **ToyLang**). Hence, one can define semantics on heterogeneous collections where element types may come from different libraries.

12.3 Applicability

Use the Translator pattern for

- *Adding semantics.* When you want to add an interpretation (even without having planned for it) to a number of classes that have different interfaces, Translator allows accessing the heterogeneous interfaces individually. The classes need not belong to same hierarchy or library.

- *External functionality.* Adding interpretations outside of elements avoids bloating the elements' interfaces with extrinsic concepts. Also, if interpreta-

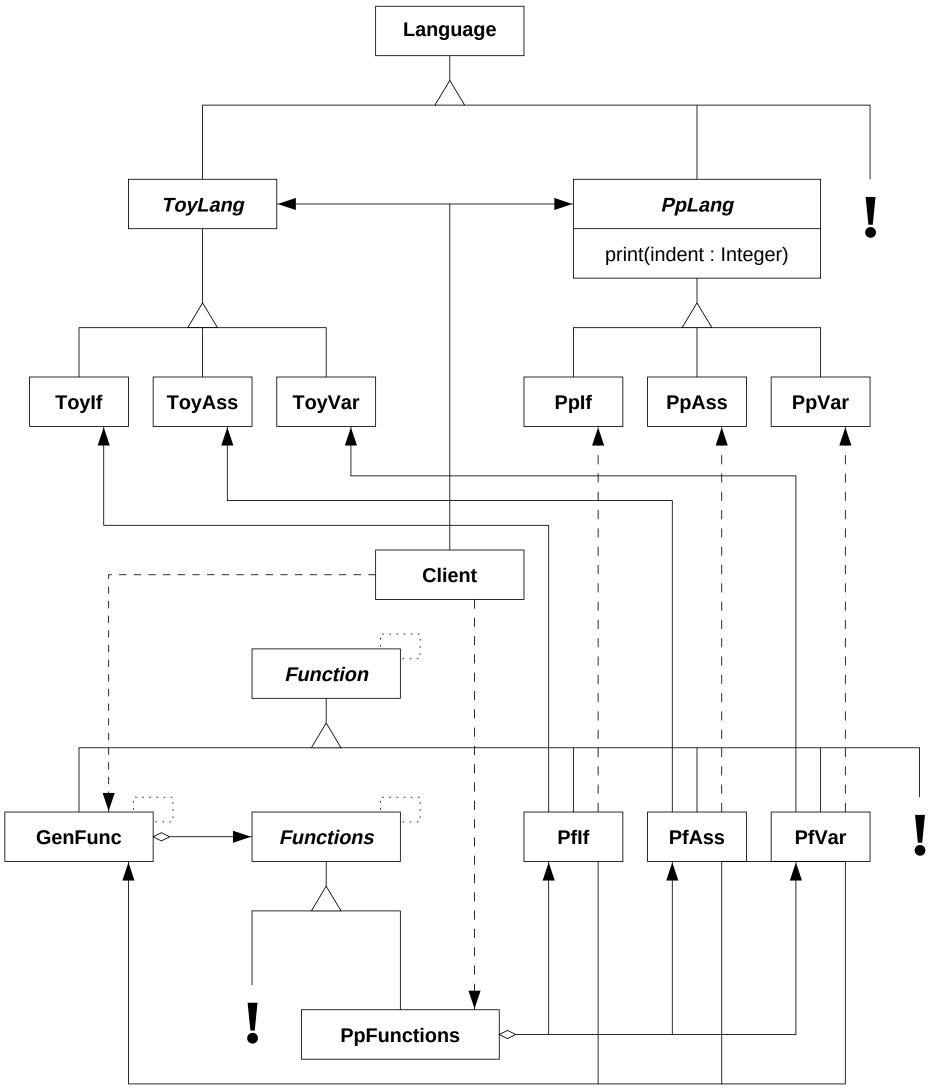


Figure 12.3: Sample structure

tions require additional servers (e.g., environment lookup for type-checking) the *interpretations*, as opposed to the *elements*, will depend on the servers, i.e., require recompilation in case one server changes.

- *Incrementality.* When small changes to big structures should not cause reevaluation of the whole structure, exploit the homomorphic properties of Translator and use the intermediate structure (see figure 12.6 on page 209) for storing computed results.

Do not use the Translator pattern in case of

- *Unstable elements.* When new elements are frequently added to the source structure it is probably better to define the interpretations in the elements.

Otherwise, one has to constantly change all associated function packages (see figure 12.4 and also section 12.7 on page 209).

- *Space constraints.* Unless you can make use of the benefits of a target structure (see section 12.7 on page 209, *Separation.*), avoid the space overhead by directly translating to results.

12.4 Structure

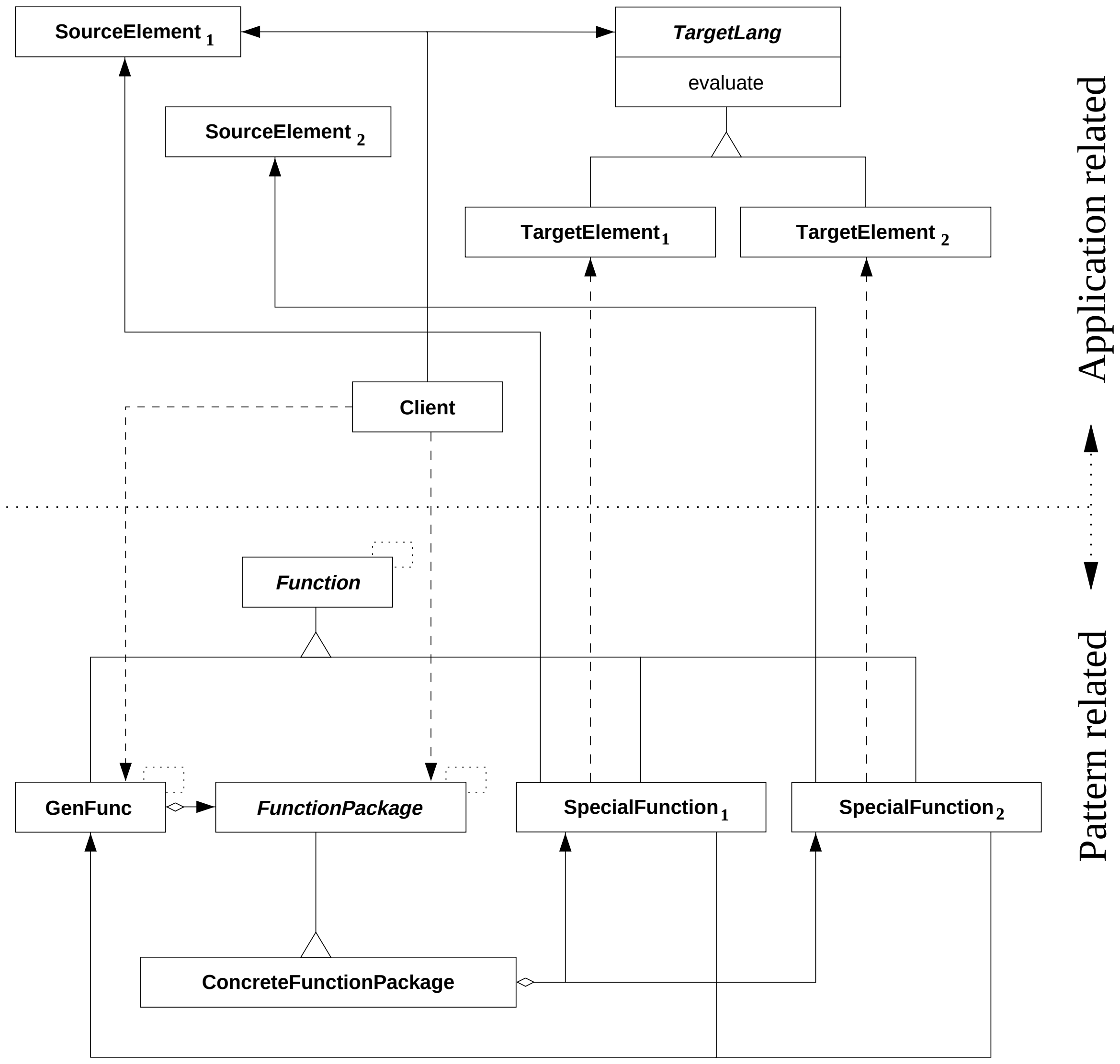


Figure 12.4: Structure diagram

12.5 Participants

- **Function** (**Function**)
  - declares an interface for function application. Its two type parameters specify argument and result type respectively[3].
  - is used as the interface specification for both generic and specialized functions.
- **Generic function** (**GenFunc**)
  - corresponds to a denotational function definition.
  - uses function package **Functions** and runtime type information to choose and then delegate to a specialized function.
- **Specialized function** (e.g., **PfIf**)
  - corresponds to one pattern matching branch of a denotational function definition.
  - defines a local transformation for a source element (e.g., **ToyIf**) to a corresponding target element (e.g., **PpIf**).
  - recursively transforms subcomponents of its argument as well.
- **Function package** (e.g., **Functions**)
  - conceptually bundles related specialized functions.
  - declares a generic package type for specialized functions to be refined by concrete function packages.
- **Concrete Function package** (e.g., **PpFunctions**)
  - defines a mapping from source elements to their corresponding specialized functions.
  - creates prototypes of — and then aggregates — specialized functions.
- **Client**
  - creates or uses a source structure (e.g., **ToyLang**).
  - initializes or uses a function package (e.g., **PpFunctions**).
  - creates or uses a generic function (**GenFunc**).
  - applies a generic function to a source structure.

[3]As is the case with all generic functions of figure 12.4 on the preceding page.

12.6 Collaborations

Figure 12.5 shows important object interactions. It refers to "@" for an infix function application syntax (see sample code in section 12.9 on page 214 or design pattern Function Object at chapter 7 on page 93).

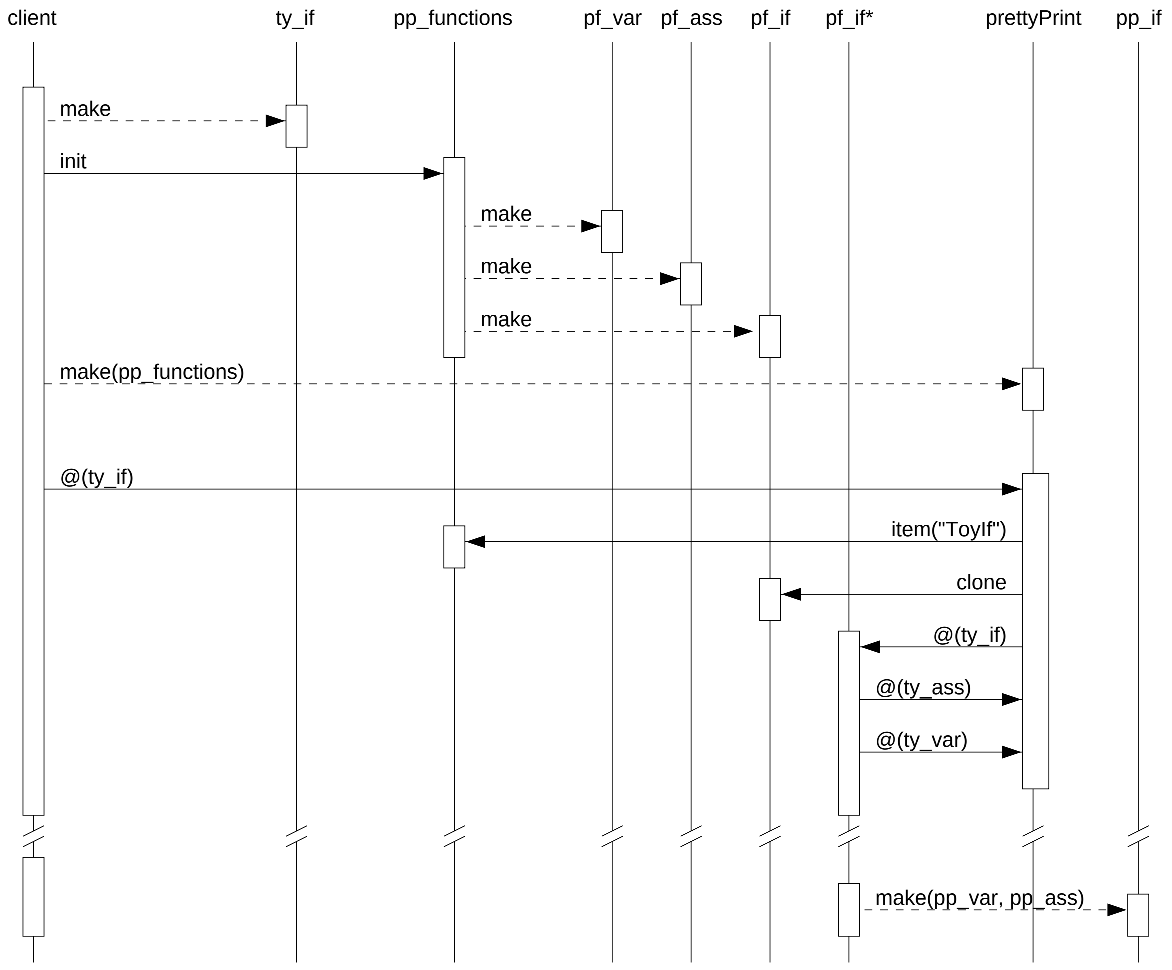


Figure 12.5: Interaction diagram

- A client initializes a function package in order to create a generic function from it. The client applies the generic function to the source structure in order to obtain the translation result.
- The generic function consults the function package for a specialized function that matches the type of the argument. Then it applies a cloned exemplar of the specialized function to the argument.
- A specialized function recursively calls its associated generic function to the subcomponents of its argument. Then it creates the target element while providing it with the results of the subcomponent evaluation.

Note that the time line gaps in figure 12.5 on the preceding page denote potential recursive mappings of subcomponents.

12.7 Consequences

Tradeoffs of Translator are:

- *External functionality.* Translator makes it easy to add interpretations to data structures with heterogeneous elements. In contrast to Visitor [Gamma et al.94] there is no need to impose an `Accept` method on the elements. Spreading interpretations over all elements (corresponding to object-oriented design) would demand changing all elements when introducing a new interpretation. Gathering all related behavior into one generic function (corresponding to functional design) — thus separating unrelated behavior (e.g., compilation from pretty-printing) — results in a clean partition and allows to hide interpretation specific details (like interpretation specific data structures and accumulated state) in generic functions.

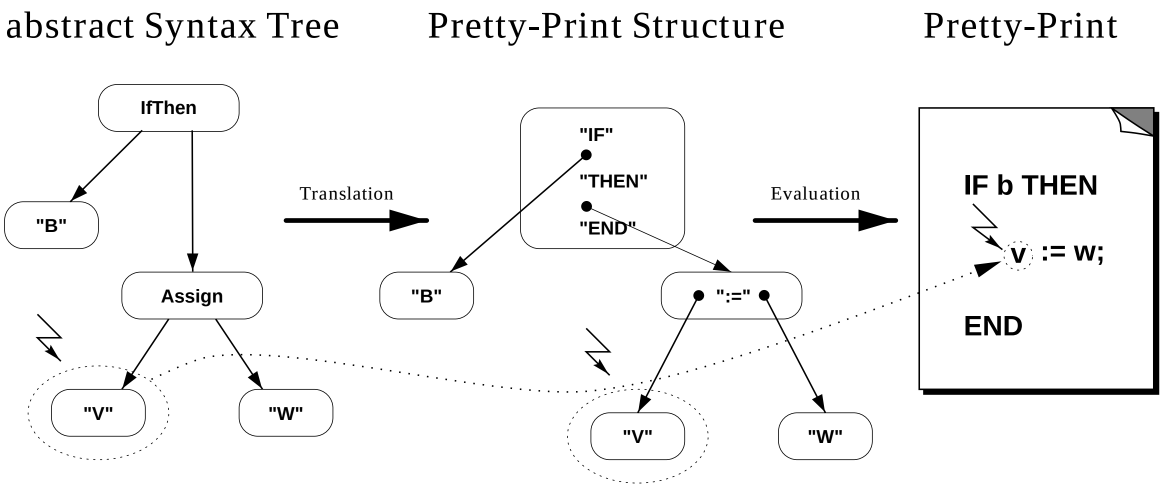


Figure 12.6: Distinct interpretation phases

- *Instability.* When using Translator, adding new elements (e.g., changing the abstract syntax of a language) becomes difficult. Unlike Visitor, Translator does not demand that one extend all interpretations with a meaning for a new element (e.g., compilation is not affected by adding a new type-declaration node). However, updating of concrete function packages and creation of specific functions is required. Note that the latter point must be done anyway, but if interpretations are element methods then their completeness can be enforced by the compiler. Using Translator, runtime errors caused by an undefined mapping to a specialized function may occur.

- *Extensibility.* It is easy to add new translations and/or target structures. A new interpretation simply requires

  1. defining a new target structure (top exclamation mark in figure 12.3 on page 205),
  2. defining specialized functions (rightmost exclamation mark), and
  3. providing a function package that maps the source elements to their specialized functions (leftmost exclamation mark).

  The last action is a tribute to the emulation of generic functions.

- *Flexibility.* Elements to be translated need not be from one class hierarchy. In any case, the full individual interface can be accessed by the associated specialized functions.

- *Broken encapsulation.* Since specialized functions can access the public interface of elements only, the interface may be forced to be wider than necessary for other clients. A remedy is to use selective export or another kind of "friend" mechanism.

- *Separation.* The semantics of an interpretation is defined in terms of a target structure semantics (see figure 12.6 on the page before). Thus, a clear separation between translation (mapping to a target) and target semantics (meaning of target) is achieved. Figure 12.7 depicts how an *interpretation* is split into a

$$\begin{tikzcd}
\mathit{Source} \arrow[r, "\mathit{translate}"] \arrow[dr, swap, "\mathit{interpretation}"] & \mathit{Target} \arrow[d, "\mathit{evaluation}"] \\
 & \mathit{Result}
\end{tikzcd}$$

Figure 12.7: Splitting the interpretation

translation to a new *Target* structure and an *evaluation* function that produces the final result. During the translation several simplifications are possible (see table 12.1 on the next page).

A pretty-print, therefore, is not a sequence of side effects but, at first, a hierarchical structure of print-elements, combinators, and possibly layout functions. In a second step the, e.g., string representation of a pretty-print is produced from this structure. Note that now it is perfectly alright to implement the semantics of the target structure as member methods of the target structure. Since that structure is meant for only one purpose, as opposed to the

Notation	*Meaning*	*Example*
$\phi(op_1) = op'$, $\phi(op_2) = op'$.	Map distinct source elements onto one destination element.	Translate both `assign` and `initialize` to `store`.
$\phi(op(a,b)) = op'(\phi(a))$.	Drop source operands.	Do not consider type declarations for compilation.
$\phi(op(a,b)) = \phi(a)$.	Prune source operators.	Compile procedure bodies only and forget about headers.
$\phi(op(a,b,c)) = \phi(b)$, *if* $pred(a)$.	Perform static analysis to select operands.	Compile `if-then-else` to its `then` branch, if the boolean condition is always true.

Table 12.1: Simplifications possible while translating

abstract syntax tree which has many interpretations, there is no drawback involved. Figure 12.8 shows the separation between interpretations and their

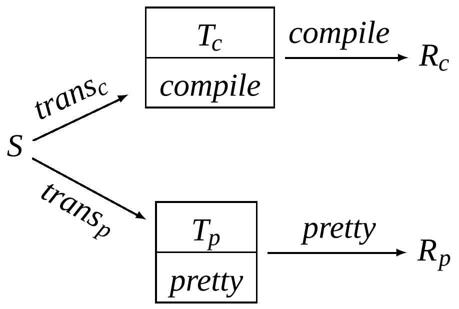


Figure 12.8: Non-interfering semantics

associated target structures (T_c and T_p). The operation names inside the target structure boxes denote their definition as member methods. There will be no new interpretations on target structures that would require to open and change the whole set of their classes. The interpretation can be defined locally for each element, exploiting inheritance and redefinition, without the disadvantages mentioned in section 12.2 on page 201.

Besides the nice partition between translation- and semantic related code, the target structure also may serve as a logical structure for the final representation. For instance, a mouse click onto a keyword could cause highlighting the whole corresponding statement and its subcomponents. While this could be achieved by back-pointers into the original abstract syntax tree as well, it is cleaner and more appropriate (as no reinterpretation is necessary) to refer to the pretty-print structure. This argument becomes more obvious in case of interpretations whose logical structure bear less resemblance to the abstract syntax tree (e.g., type-checking information).

Also, assuming multiple users are working simultaneously on one abstract syntax tree, multiple intermediate structures allow them to, e.g., use different compilation options to achieve various code results. If semantic results were stored directly in the source structure, this would be a cause for interference.

Furthermore, an intermediate structure is also helpful when aiming for incremental updates.

- *Incrementality.* Naturally, target structures are subject to fast incremental recomputation since they are produced by homomorphic mappings from their source structure (see section 12.2 on page 201). Assuming N to be the number of elements in the source structure, the asymptotic amount of recalculation needed to update the result of an abstract interpretation is reduced from $\mathrm{O}(N)$ down to $\mathrm{O}(\log N)$. However, it is necessary to store previously computed results somewhere. Our approach of non-intrusive addition of interpretations forbids storing these at the source elements. The target structure, however, (see figure 12.6 on page 209; the small flash signs denote places of change) can serve as a store for incremental results. The target structure can play the role of an Observer [Gamma et al.94] that becomes notified by changes in the source structure and starts the necessary recomputations. The target structure, in turn, is observed by the final result.

  Note that though a local source structure change will only cause a local target structure change, the reinterpretation in general has to be done globally. Sometimes local changes to the interpretation result suffice (e.g., pretty-print of identifier), sometimes the path from changed location up to the root has to be followed (e.g., context-relations [Snelting86]), and sometimes the whole target structure must be revisited (e.g., dynamic semantics). Also with regard to this aspect, the target structure serves as a physical anchor to distinguish between the two phases of translation and incremental reevaluation.

- *Space overhead.* If incremental evaluation is not an issue and space efficiency is a priority, then the intermediate structure should be avoided in favor of a direct translation to the final semantics. In a distributed setting, however, we may purposely want to trade space for speed. The intermediate structures can be made local to their users, while the single source structure is accessed only when needed. Ergo, frequent evaluations of target structures do not add to net traffic.

12.8 Implementation

Here are two issues to consider when implementing Translator (also see section 12.10 on page 217, *Collaboration* & *Implementation*):

- *Mapping elements to functions.* Translator uses runtime type information (RTTI) to determine the concrete type of a generic function argument. This mechanism is very language dependent and may also vary with different compilers for one language. Usually, it is possible to obtain a string of the class name or test for successful downcasting to a specific type. If no such mechanism is available, one is left with explicitly programming a type inquiry interface. This is, however, incompatible to the otherwise non-intrusive nature of Translator.

  In any case, a generic function may also dispatch on values of objects as opposed to their type. Consequently, you may represent musical notes and quarter notes by the same class. The corresponding objects will differ in a value, e.g., of attribute duration. Nevertheless, it is still possible to use a generic function to dispatch on this note representation.

- *Hiding function packages.* It is easy to shield the client from the existence of particular function packages, by providing a tailored generic function that creates a standard generic function with a fixed function package (e.g., `prettyFunctions`). The client code, i.e., using a generic function is simplified to

  ```
  source:           ToyLang;
  prettyPrint:      PrettyPrint;
  ...
  !!prettyPrint.init;
  (prettyPrint @ source).display;
  ```

  (compare to the version on page 215 of section 12.9). In order to achieve this code shortening and client encapsulation from function packages one simply needs to provide a tailored generic function like the one below.

  ```
  class PrettyPrint
  inherit Switch[ToyLang, PpLang]
  redefine functions end
  creation init
  feature
    functions : PpFunctions;

    init is do
      !!functions.init;
    end;
  end
  ```

12.9 Sample Code

The following use of EIFFEL code should not conceal the fact that you may use Translator with any language featuring runtime type information such as SMALLTALK, C^{++}, and JAVA.

Assume a toy source language with an if-statement (see figure 12.3 on page 205):

```
class    ToyIfthen
inherit  ToyLang
creation make
feature
  exp, stat:  ToyLang;
  ...
end
```

The corresponding pretty-print element could be:

```
class    PpIfthen
inherit  PpLang
creation make
feature
  pexp, pstat:  PpLang;

  make (e, s : PpLang) is
  do
    pexp:=e;
    pstat:=s;
  end

  display is
  do
    io.putstring ("IF ");
    pexp.display
    io.putstring (" THEN ");
    pstat.display;
    io.putstring (" END")
    io.new_line;
  end;
end
```

Now we need the specialized function that maps an if-statement to its pretty-print element.

```
class PrettyFunctionIfthen
inherit Function[ToyIfthen, PpIfthen]
creation make
```

```
feature
  genFunc: GenFunc[ToyLang, PpLang];

  make(s: like genFunc) is
  do
    genFunc:=s
  end;

  infix "@" (ift: ToyIfthen) : PpIfthen is
  do
    !PpIfthen!Result.make(
      genFunc @ ift.exp,
      genFunc @ ift.stat)
  end;
end
```

The creation argument of type **GenFunc** specifies the generic function to be used for evaluation of subcomponents (`exp` and `stat`). Its generic[4] parameters denote the function type to be going from **ToyLang** to **PpLang**.

The method for function application (@) simply creates the pretty-print element while supplying the results of recursively evaluating the subcomponents (`exp` and `stat`).

The client code for performing a full interpretation is:

```
  source:           ToyLang;
  prettyFunctions:  PpFunctions;
  prettyPrint:      GenFunc[ToyLang, PpLang];
  ...
  !!prettyFunctions.init;
  !!prettyPrint.make (prettyFunctions);

  (prettyPrint @ source).display;
  ...
```

Prior to its usage, a function package must be initialized by calling `init`. Then, a generic function (`prettyPrint`) is created by suppling a pretty-print function package (`prettyFunctions`). Next, the generic function is applied to the source structure, yielding a target structure. The semantics are finally produced by invoking `display` on the target structure.

A concrete function package appears as follows:

```
class    PpFunctions
inherit  Functions[ToyLang, PpLang]
```

[4]This time generic means (static) parametric polymorphism, whereas we imply (dynamic) inclusion polymorphism in case of generic functions.

```
creation init
feature
  init is
  local
    pfVar:      PfVar;
    pfAssign:   PfAssign;
    pfIfthen:   PfIfthen;
    prettyPrint: GenFunc[ToyLang, PpLang]
  do
    make(3);
    !!prettyPrint.make(Current);
    !!pfVar;
    !!pfAssign.make(prettyPrint);
    !!pfIfthen.make(prettyPrint);
    put(pfVar,    "ToyVar");
    put(pfAssign, "ToyAssign");
    put(pfIfthen, "ToyIfthen")
  end;
end
```

Each concrete package inherits from an abstract function package class which, in turn, inherits from **Hash_Table**:

```
deferred
class Functions[Source, Target]
inherit Hash_Table[Function[Source, Target], String]
feature
  init is deferred end;
end
```

So, `make(3)` initializes the function package to a hash table with three entries. Next, a generic function is created in order to serve as the creation argument for the three specialized function prototypes. Note that the `Current` argument in the creation of the generic function causes the very function package that is currently being initialized to become the argument for the generic function that is supplied to the specialized functions. The function to print variables (`pfVar`) does not need to recursively evaluate subcomponents, ergo it does not require a generic function for its creation. Finally, the specialized function prototypes are put into the hash table using their corresponding source element class names as keys.

Therefore, the application method of the generic function definition —

```
class    GenFunc[Source, Target]
inherit  Function[Source, Target];
         Internal;
creation make
```

```
feature
  functions: Functions [Source, Target];

  make (fs: like functions) is
  do
    functions:=fs
  end;

  infix "@" (source: Source): Target is
  do
    Result:=clone(functions.item(class_name(source)))
            @ source
  end;
end
```

— can simply access the class name of the source element (method `class_name` is inherited from the system class **Internal**), use it to retrieve the correct specialized function prototype (call item on the function package)[5], and then apply a cloned exemplar to its own argument. Instead of a hash table we also could have used a dictionary or even a type case switching statement in order to achieve dispatching on the type of arguments.

12.10 Related Patterns

12.10.1 Categorization

Function Object: Translator's interface to clients is that of a function object.

Interpreter: Interpreter suggests inventing and representing small languages for reoccurring problems [Gamma et al.94]. Translator and Interpreter do not address parsing, i.e., already presume the existence of an abstract syntax representation. Translator is well-suited defining the interpretation part of Interpreter which defines interpretations in member methods of elements (see discussion in section 12.2 on page 201) or by using Visitor.

Visitor: Visitor [Gamma et al.94] has similar motivations as Translator. Yet, besides the fact that Visitor does not cover homomorphic and incremental translations, it also uses a different means of achieving double-dispatch. Visitor relies on the straightforward technique of encoding an argument's type into method names [Ingalls86]. However, several disadvantages are aligned with this approach:

[5]At this point a runtime error may occur due to a missing specialized function. Some exception handling or other kinds of gracefully dealing with such a situation would be appropriate.

- A mutual dependency cycle is introduced between elements (abstract syntax tree) and visitors (interpretations) [Martin97]. This has implications on recompilation and regression tests.
- The elements are forced to know about interpretations because of the need to provide an `Accept` method.
- One has to provide code for interpreting all elements of a hierarchy, although only a subset will actually be considered by certain interpretations [Martin97].

Facet: Like Translator this pattern aims at supporting the addition of new and unforeseen interfaces to existing classes without impacting clients that do not require the new interfaces [Gamma97]. Thus, both patterns preserve the initial key abstraction, i.e., allow element interfaces with intrinsic properties only. Also, both patterns allow for dynamic extensions of classes. Facet differs in that it mainly aims at role modeling and does not take translations or incrementality into account.

External polymorphism: This pattern provides an alternative way (cleverly exploiting C^{++} templates) to achieve polymorphism with classes which have no common ancestor [Cleeland et al.96].

Acyclic Visitor: The dependency cycle in the original Visitor [Gamma et al.94] design caused many suggestions for improvements such as Acyclic Visitor [Martin97] and Dynamic Visitor [Nordberg96]. Both alternatives also address the issue of partial visitations, i.e., when a particular interpretation does not need to be defined on all elements. Translator may deal with such a situation as well, since there is no obligation to provide a complete set of specialized functions.

Serializer: Because Serializer [Riehle et al.97] is a specialization of Visitor, it is also related to Translator. One can think of Serializer as translating objects to a flattened form (e.g., for persistence). In fact, Translator might be more appropriate in some cases since it does not require objects to know about their ability to be serializable.

12.10.2 Collaboration

Transfold: The exploration of the source structure may be deferred to Transfold.

Composite: Translator can be used for interpretations on Composite structures [Gamma et al.94].

Observer: On may use Translator to create a view presented with the Observer [Gamma et al.94] pattern.

Flyweight: Leaves of abstract syntax trees can be represented as Flyweights [Gamma et al.94].

Void Value: A void function, i.e., a void value defining a default function, may be used to define the behavior in case a generic function can not determine an appropriate special function.

12.10.3 Implementation

Function Object: The specialized functions that map source elements to target elements are function objects [Kühne97].

The result produced by Translator may be a hierarchy of function objects, i.e., a parameterized result. For instance, a pretty print result could be a function (itself calling other functions) awaiting the indentation level of the current subtree. Such a structure corresponds to *translucent procedures* [Rozas93]. Such a function structure is able to evaluate, but also subject to examination and decomposition into individual functions again. Consequently, a translator may also interpret such a translucent function structure.

Generic Function Object: The generic functions capable of realizing double- and multi-dispatching interpretations on heterogeneous data structures are generic function objects [Kühne97].

Lazy Object: Translator may employ Lazy Object to implement lazy interpretations, i.e., both intermediate structure and result may be lazy objects.

Prototype: Function packages contain function prototypes [Gamma et al.94]. Further instances are created by cloning and the associated generic function is a preset attribute value.

Singleton: Instead of using an attribute genFunc (see section 12.9 on page 214) in specialized functions, these may alternatively access a Singleton class [Gamma et al.94] in order to retrieve their corresponding generic function. EIFFEL allows for a particularly easy Singleton implementation, using the "once" mechanism.

Observer: A chain of Observers [Gamma et al.94] can be employed to account for the data dependency between source structure, target structure, and target semantics.

13 Collaboration

Learn the patterns, and then forget 'em.
– Charlie Parker

The previous chapters introduced six functional patterns. Each is of value on its own. A language, design toolkit, pattern system, or paradigm is of special value, however, if the elements are orthogonal to each other and allow mutual combination to the extent of being a generative[1] system.

> *"A pattern system for software architecture is a collection of patterns for software architecture, together with guidelines for their implementation, combination and practical use in software development [Buschmann et al.96]."*
> – PoSA Group

The presented functional pattern system exhibits a remarkable amount of interconnection and collaborations (e.g., compared against the 23 GOF patterns [Gamma et al.94]). Figure 13.4 on page 226 shows the full relationship graph for the pattern system, but it is more instructive to view partial diagrams first. According to our separation of the *Related Patterns* section, we organize pattern relationships into three[2] items:

- *Categorization.* Which pattern "is-a" kind of another pattern?
  This relationship reveals what other *pattern roles* a pattern may adopt in a particular use.

- *Implementation.* Which patterns do implement other patterns?
  Which patterns are useful to realize the inner mechanics of another pattern?

- *Collaboration.* Which patterns can collaborate with each other?
  This is a symmetric relationship where individual pattern strengths are connected like pieces in a jigsaw puzzle.

The following diagrams and explanations collects and concentrates pattern relationships to create new perspectives. For a full understanding and more example relationships, it may be required to (re-) consult the individual *Related Patterns* sections.

[1]Pattern languages are also called "generative". They allow generating a product by following course patterns to finer grained patters through the language. Here, we mean bottom-up generation from composable components.

[2]Usually, the term "collaboration" subsumes both *Implementation* and *Collaboration* items.

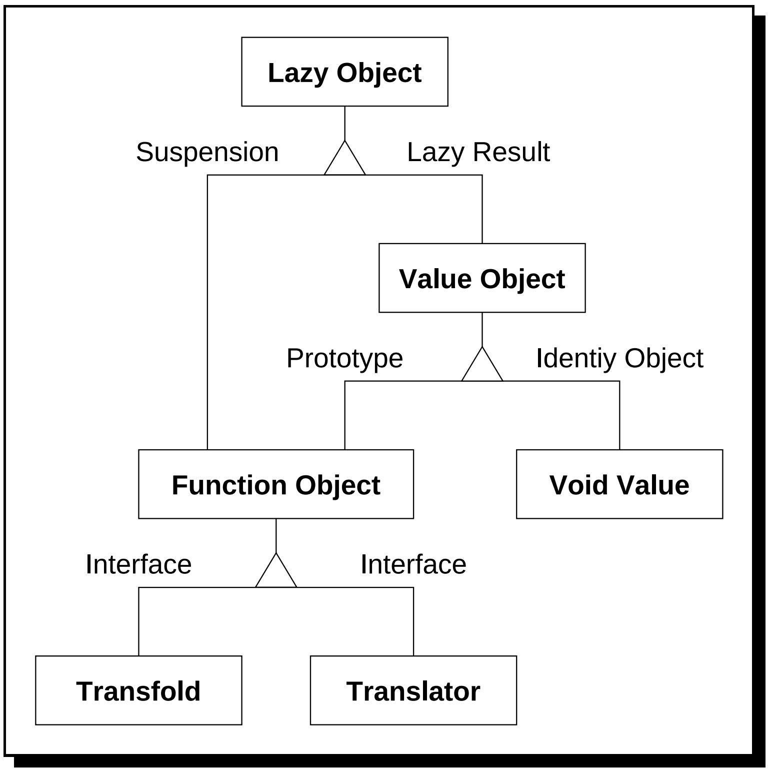


Figure 13.1: Pattern categorization

Figure 13.1 illustrates the pattern "is-a" hierarchy in OMT notation. Transfold and Translator are both used through a Function Object interface by clients. Clients apply transfolds and generic functions to structures respectively. A Function Object, which still awaits arguments, represents a suspended calculation and, therefore, "is-a" Lazy Object. A Function Object also behaves like a Value Object, in that it never alters its state (except Procedure Object, see section 7.10 on page 107), but returns a freshly created instance, akin to a Prototype [Gamma et al.94].

Pattern Void Value is a Value Object, because it has an immutable interface, might provide generator methods (see chapter 9 on page 149), and shares the unique instance (Identity Object) property with Value Object.

Finally, the result of a Value Object generator method can be a lazy Value Object. Note that figure 13.1 does not include — to provide a clearer presentation — that every pattern except Lazy Object "is-a" Void Value: Function Object and its descendents may be default functions for a parameterizable component, hence, playing the role of a Void Value. Even Value Object may be a Void Value, by defining a default value for a value type. Note, that this does not introduce a cycle into the hierarchy, since a new Default Value "class/role" would inherit from both Value Object and Void Value[3].

The "is-a" hierarchy of figure 13.1 is helpful in understanding the individual patterns and also allows keeping individual pattern descriptions shorter. For instance, pattern Void Value "is-a" Value Object, hence, the discussion about Singular Object and Identity Object in pattern Value Object may also be applied to Void Value. Or, using transitivity of the "is-a" relationship, a Transfold could be used as a Lazy Object calculation in a Function Object callback situation.

Figure 13.2 on the facing page depicts how patterns support each other in terms of implementing a pattern's functionality. Transfold uses Function Object for struc-

[3]It is interesting, though, how multiple inheritance can establish a symmetric "is-a" relationship between two inherited classes.

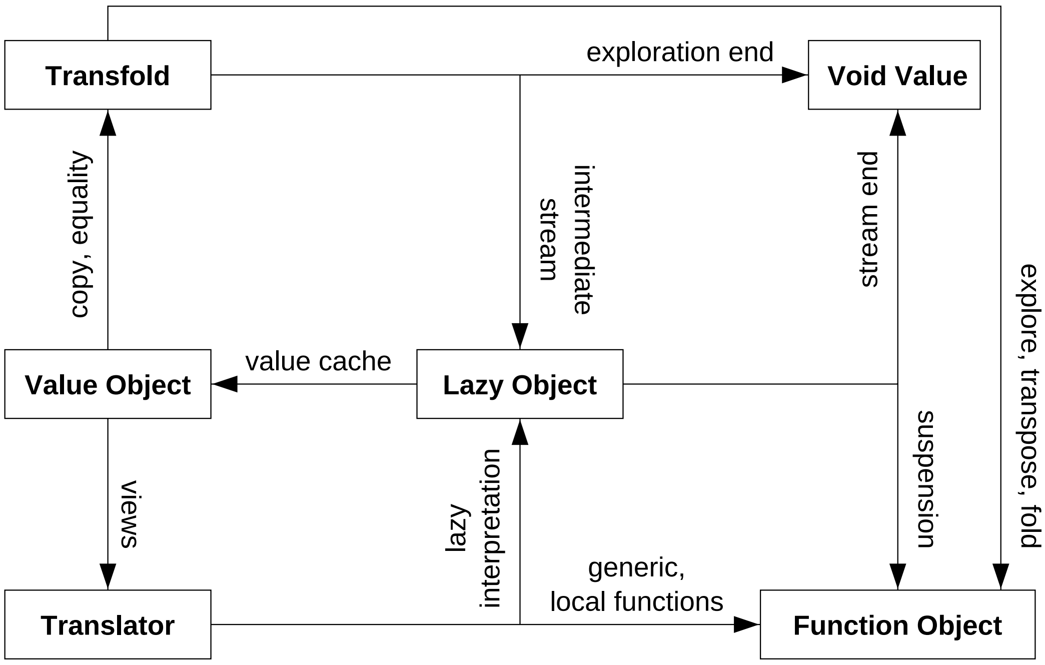


Figure 13.2: Pattern implementation relations

ture exploration, stream transposition, and folding. Note that the association labels in figure 13.2 are role names [Rumbaugh91]. They denote the service a pattern provides for the one that originates the association.

Also note that the responsibilities of Transfold could have been implemented by using other techniques, but Function Object exactly provides the properties and services needed for the tasks. Function Object, furthermore, provides the local transformation functions and the concept of a generic function to Translator, and implements stream suspensions for Lazy Object.

Void Value can help both Transfold and Lazy Object to define the end of an exploration or a stream respectively. Transfold, in turn, may support Value Object in defining copy and equality operations by iterating over value components. A complex Value Object may use a Translator to implement sophisticated interpretations on it. Value Object again, is used by Lazy Object to cache stream values. Lazy Object itself, provides lazy intermediate streams for Transfold and enables Translator to implement lazy interpretations.

The diagram in figure 13.2 is useful for designers who want to apply a pattern and look for supporting patterns to ease its realization (see the individual *Related Patterns* sections for possible non-functional pattern support). It should be clear that even if a pattern is supported by the language, it may still use another implementation pattern. For instance, even built-in lazy streams could still rely on Void Value to define the behavior of a stream at its end. We do not count it as an collaboration, though, since making Lazy Object's realization easier with Void Value does not benefit any client.

It is the desirable property of a collaboration — in the the sense as we use the term here — that it combines the strengths of two patterns to the benefit of clients, using the resulting product.

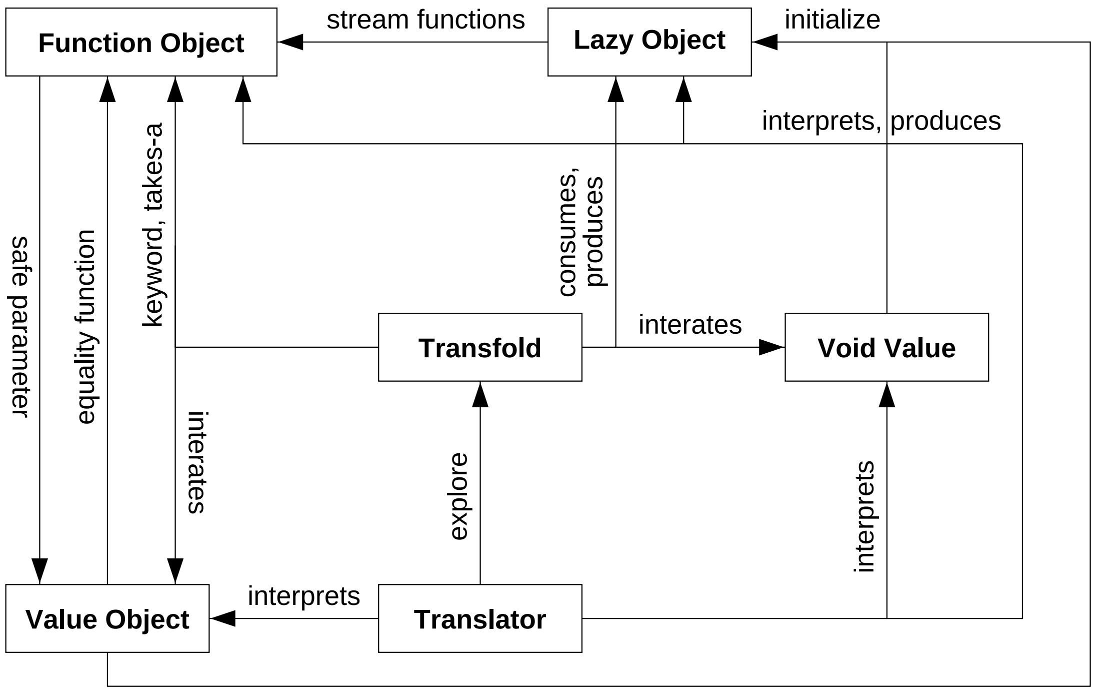


Figure 13.3: Pattern collaborations

Figure 13.3 illustrates possible pattern collaborations. This time, the labels are placed in the center of an association because they name a symmetric relationship. The arrows indicate the direction of reading the association, e.g., Function Object may collaborate with Value Object in order to avoid parameters changing their values, after having been passed to a Function Object.

Value Object, in turn, may offer clients to change its equality checking function with a Function Object (see the hierarchy in figure 13.1 on page 222 to realize that the passed function could be a Transfold). Value Object and Void Value can both help to define structures that are interpreted by Translator.

Translator itself, may employ Transfold to defer the structure exploration process and both interprets and produces Lazy Objects. As a special case of that, Translator may interpret Function Object structures and produce parameterized Function Object results. Pattern Transfold also consumes and produces Lazy Objects, while Lazy Object may collaborate with Function Object to offer clients functional parameterization of stream functions. Clients similarly benefit from Transfold's parameterization with Function Object, and in particular also from Keyword Parameter Function Object that facilitates managing Transfold's parameters. As Translator does, Transfold also iterates over structure containing Value Objects and Void Values. Finally, both Value Object and Void Value may use Lazy Object to lazily initialize components.

Again, it might be worthwhile to consult the hierarchy of figure 13.1 on page 222, e.g., to see that both Void Value and Function Object are also safe parameters to Function Object, since they are a kind of Value Object.

A collaboration diagram, like figure 13.3 on the facing page is a useful map for creating a design with interacting and each other mutually enhancing patterns.

The high cohesion of the presented functional pattern system can be best seen in a diagram including all three covered types of relationships.

Figure 13.4 on the next page features three arrow types, corresponding to the associations of the three former diagrams. An arrow pointing from pattern X to pattern Y should be interpreted — in the order of the arrows in the legend of figure 13.4 on the following page — as

1. X uses Y in its implementation.
2. X uses Y in a collaboration.
3. X plays the role and / or has the properties of Y.

Admittedly, the diagram is quite complex, but apart from providing a single-diagram overview, it also allows some interesting observations:

- All patterns have about the same number of edges, indicating high overall cohesion, but
- Function Object has the highest number of in-going edges, i.e., is heavily used. It is a kind of three other patterns, demonstrating its versatility. The remaining patterns are kinds of Function Object. All but Void Value use Function Object, giving evidence to the ubiquitous presence of this key pattern.
- Void Value has almost only in-going edges (except one classification edge to Value Object and a collaboration with Lazy Object). This underlines its nature as a base case definition and suggests its realization as a basic, atomic language feature.
- Although, Value Object could be suspected to be as primitive support as Void Value, it is interesting how balanced its in and outgoing count is, that is, how well it also benefits from other patterns.
- Lazy Object shares the highest number of collaboration uses with Function Object, i.e., composes well with other patterns to yield improved client services.
- Transfold exhibits more usage of other patterns than it is used itself. This is consistent with its higher-level conception and intended usage in libraries and client code.
- Translator has only one ingoing edge (denoting usage by Value Object for interpretations). This confirms its characterization of a high-level, client oriented service.

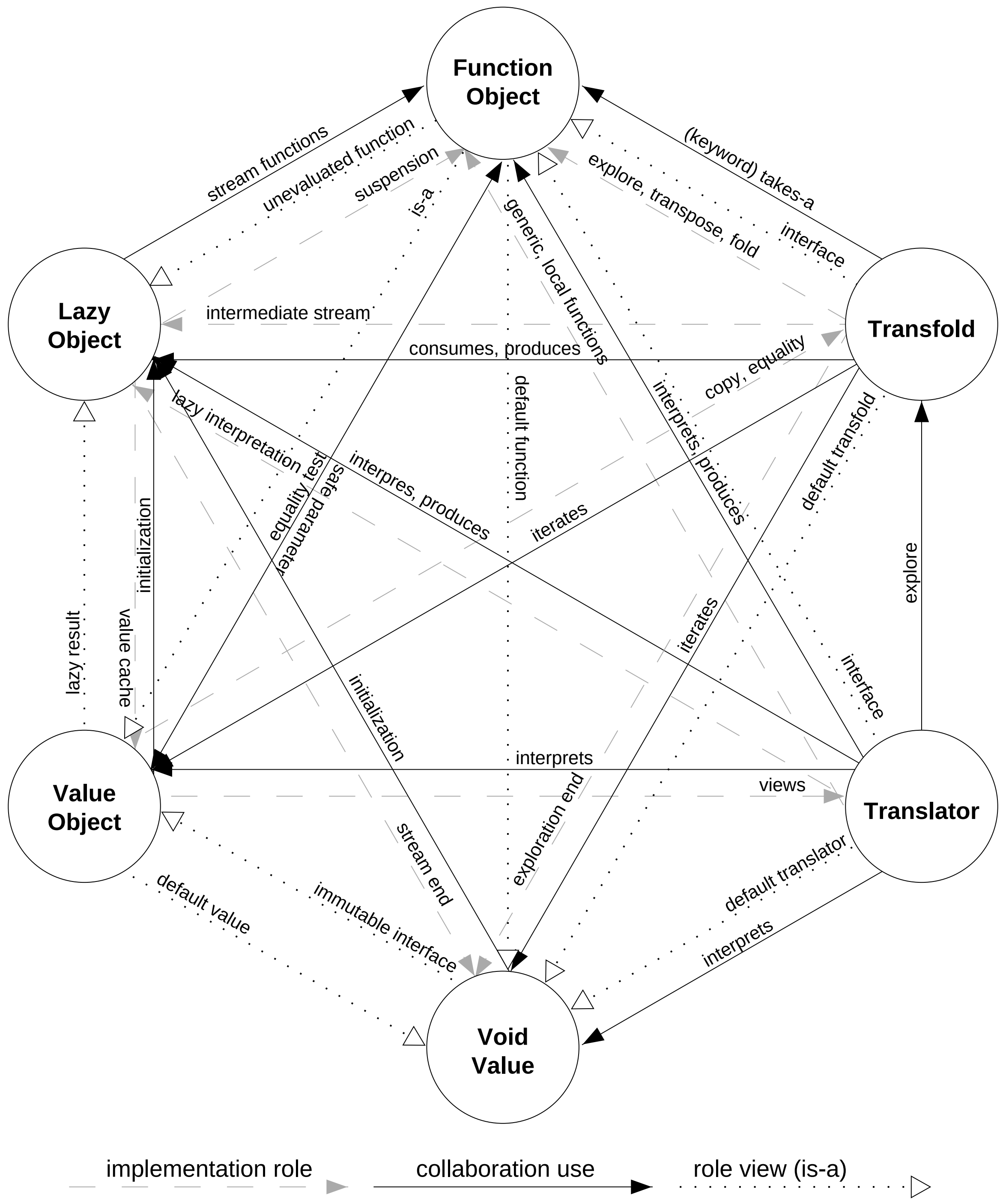


Figure 13.4: Pattern system relations

As a result, it comes to no surprise that Function Object and Lazy Object vividly claim language support due to their importance to other patterns. Pattern Transfold seems to be reasonably implementable as a principle of library and iteration organization (see section 14.4 on page 240 for arguments to not just leave it to that).

However, it is interesting to see to how well Value Object — being of good use to other patterns — can benefit from the existence of other patterns. Hence, the respective patterns, maybe surprisingly, contribute to a "low-level" property like value semantics.

Figure 13.4 on the facing page does not claim completeness. First, some relationships, e.g., Function Object "takes-a" Lazy Object, have been left out since they seem not important enough. Second, there might be yet unexplored interactions that still wait to be discovered.

13.1 Example collaboration

This section demonstrates all patterns in a single example. A well-known, non-trivial problem consists of determining whether two trees contain the same elements with respect to an in-order traversal. The challenge is to stop searching when a difference is found, i.e., to avoid unnecessary exploration of subtrees. This task is known as the *samefringe problem* [Hewitt77] and has been used to both, argue in favor of lazy lists [Friedman & Wise76], and also to demonstrate the inferior iteration mechanisms available with C^{++} [Baker93][4].

In this example (see figure 13.5 on page 229) we are comparing machine code consisting of instructions like load-data (LD data), load-address (LA address), etc. The machine code is generated by interpreting an abstract syntax tree, whose nodes are represented in conformity with the Void Value pattern. The interpretation to machine code is defined with the Translator pattern. Translator employs the Generic Function Object pattern for its local transformations. The result of the abstract syntax tree interpretation is a tree of machine code instruction nodes, which is flattened[5] to a lazy stream of instructions using the Lazy Object pattern. Lazy Object generates a lazy stream of instructions using the Value Object pattern to cache already generated results. Finally, two differently generated streams are compared using the Transfold pattern. Transfold itself, is parameterized according to the Function Object pattern.

In addition to be a readily accessible design, the configuration depicted in figure 13.5 on page 229 has several benefits listed in table 13.1 on the next page.

Note that Function Object alone would have sufficed to solve the samefringe problem [Baker93, Läufer95], by defining two lazy functions to generate and compare two streams. Still, pattern Lazy Object helps to better understand the delaying aspect of Function Object and makes the stream concept explicit. Transfold generalizes the parallel consumption of two streams to an iteration framework and Translator allows the two tree arguments to be the result of complex interpretations. Void Value and Value Object help to simplify and optimize the underlying data structures.

Ergo, by no means all patterns are *required* to solve the problem, but each contributes to the accessibility and extendibility of the presented design.

[4]This article did not consider the use of Function Object and, hence, did not tell the whole story.

[5]Using a post-order traversal rather than the usual in-order traversal.

Property	*Description*	*Contributor*
Case-less programming	There is no need for a case analysis of tree leaves. Any related behavior, especially that of non-existent children, is distributed to the leaves.	Void Value
Non-intrusive traversal	Abstract syntax trees do not need to provide support for their interpretation. Multi-dispatching operations can be implemented without affecting the source structure.	Generic Function Object
Hierarchical, incremental interpretations	Interpretations can be defined in terms of local transformations and support incremental updating.	Translator
Demand-driven computations	Flattening a tree causes its traversal only to a certain extend, depending on the amount of flattened data consumed. Tree contents can be accessed through a defined and efficient interface.	Lazy Object
Safe caching	Stream elements are eventually represented as immutable values who can be safely used for caching computed results.	Value Object
Internal iteration	Multiple streams can be processed in parallel without the need to explicitely code control structures. A powerful, general scheme can be parameterized to implement numerous useful operations.	Transfold
Parameterized functionality	True closures allow parameterizing general mechanisms in a black-box, plug-in fashion. Functions may be composed and exchanged at runtime.	Function Object

Table 13.1: Benefits of the functional pattern design example

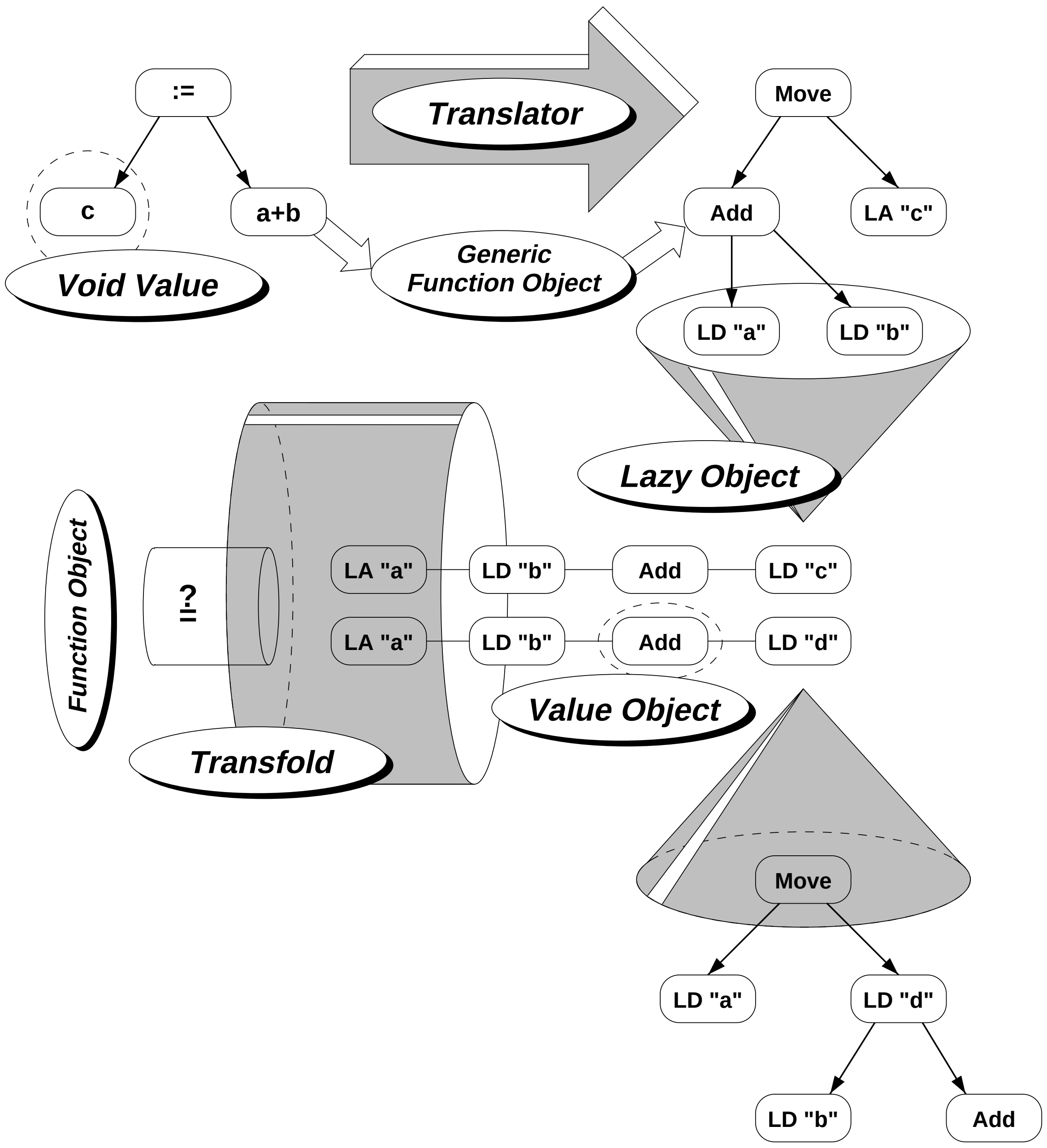


Figure 13.5: Functional pattern decomposition of “Samefringe”

Part III

Language design

14 Pattern Driven Language Design

Die Grenzen meiner Sprache sind die Grenzen meiner Welt.
– Ludwig Wittgenstein

14.1 Introduction

The functional pattern system presented in the previous part makes a twofold contribution to language design. First, implementing patterns that represent flexibility and reuse promoting designs, puts an implementation language to the test. Any encountered problems are likely to occur in other attempts to produce flexible designs as well. Second, most patterns claim to be supported by the implementation language directly instead of being coding conventions. How are the suggested language concepts interrelated and what type of individual support should be provided? As the emphasis of this book is the functional pattern system itself, the intent of the following observations is to provide an outlook to further developments, i.e., the aim is to identify impulses rather than complete solutions.

This part is organized as sections of patterns, each commenting on both above mentioned aspects. It concludes with an overall reflection and suggests to reevaluate the roles of languages and their associated environments.

14.2 Function Object

Undeniably, function objects enrich object-oriented programming with many useful properties. Function objects hide the number of both parameters and results to clients. This can be viewed as an aid to modularization, just like classes in object-oriented design [Parnas72] or higher-order functions and lazy evaluation in functional programming [Hughes87]. Accordingly, aggregation (“has-a”), inheritance (“is-a”), and behavior parameterization (“takes-a”) should be equally well-known to designers. “Takes-a” realizes object-composition, as opposed to breaking encapsulation with inheritance [Snyder86]. It is therefore a means of reaching the goal of component oriented software [Jazayeri95, Nierstrasz & Meijler95]. In combination, inheritance and Function Object allow for flexible prototyping as well as safe *black-box* composition. Indeed, function objects reintroduce some flavor of structured analysis and design to object-orientation. This is definitely useful. While adding new objects to a system is caught by an object-oriented decomposition, adding functionality often is better handled by extending functional abstractions. The *control-objects* in Jacobson's "use-case driven approach" [Jacobson et al.94] represent such points of functional extendibility.

Obviously it is a tedious and superfluous activity to code a set of classes to support partial parameterization for function objects (see section 7.4 on page 98). While it is straightforward to support partial parameterization by a tool that — given just the final function object definition and a specification of all parameter types — generates all necessary classes, it is quite clear that language support is desirable. Language support, in general, should take care of

- a short syntax for function application and composition,
- implicit function object creation,
- garbage collection,
- support for partial parameterization,
- a means to define anonymous function objects, and
- a type system that allows specifying type constraints.

The following paragraphs expand on the above items. Clearly, a short syntax for function application dramatically increases code readability. There is no doubt that

```
(streamAdd @ hammings @ primes).show(9);  -- Eiffel
(streamAdd(hammings)(primes)).show(9);    // C++
```

is superior to

```
streamAdd.apply(hammings).apply(primes).show(9); // Java
```

Neither should it be possible to forget the creation of functions objects nor should it be necessary to make an idiomatic use of a language feature, which was meant for a different purpose (see section 7.9 on page 105). Implicit creation of function objects is a natural consequence with language support for Void Value (see section 14.7 on page 248).

Many arguments against closures are made on the grounds of their unpredetermined lifetime which requires heap rather than stack allocation. The usefulness of closures partly stems from this property (e.g., making it possible to use them for decoupling implementation from invocation and suspending calculations) and object-oriented languages heavily use heap allocation for objects anyway. Without a garbage collection support, implementation of the function object pattern is still possible but more involved [Läufer95]. The SELF programming language features two different types of blocks (closures) with a different scope of lifetime [Chambers92a]. This allows avoiding overhead for stack allocatable closures (downward-funargs). Certainly, one closure type only in a language is preferable and the task to replace appropriate occurrences with a stack allocation based version should be left to the compiler.

Although PIZZA supports higher-order functions, partial parameterization is not automatically supported [Odersky & Wadler97]. Chapter 7 on page 93, however, clearly demonstrated the benefits of partial parameterization support for function objects. This decoupling and reusability promoting feature should be available for free without hand coding classes or resorting to curry-functionals. In an investigation about the impact of language features on reusability, one of the criteria used for evaluation was support for adapting components to their use-contexts [Biddle & Tempero96]. The ability to supply arguments to a function (component) in advance makes it possible to adapt it to its calling environment (context). Especially the useful combination of keyword parameters and partial parameterization should receive language support.

Often, for instance, for iteration purposes, only a short function needs to be passed which is more conveniently done by passing an anonymous function rather than defining a class and then passing an instance of that class. SMALLTALK's blocks are used for such purposes and also BETA allows passing pattern definitions, which are implicitly instantiated [Madsen et al.93]. Surprisingly, support for implicit binding of free variables [Breuel88] is not the driving force for language support. On the contrary, it has been shown that explicit passing of parameters better supports reuse of function objects (see section 7.9 on page 105). Even with anonymous function objects, changing a free variable name in the function object context will imply only one change in parameter passing, as supposed to a possibly more involved change to all occurrences of the free variable in the anonymous function definition.

The possibility of using function objects for method simplifications, or for imperative commands that support undoing (see section 7.4), or the idea to exploit inheritance between functions [Schmitz92] all suggest language support in the spirit of patterns in the BETA language [Madsen et al.93]. A pattern can be used as a function or a class definition and, thus, nicely combines both concepts. Although this holistic approach is very useful and aesthetically pleasing, it does not help to resolve the fundamental dichotomy between a functional and an object-oriented decomposition approach (see chapters 4 on page 55 and 12 on page 201). Like physicians have to eventually settle on a particle or wave interpretation of elementary particles in a given experiment, the software designer has to use a BETA pattern as an object or a function. If choosing objects one looses functional extendibility and if choosing functions one looses data extendibility. Research has been carried out for finding guidelines as to which from a multitude of patterns is appropriate in specific design situations [Coplien95]. Ideally however, no commitment would be necessary and either viewpoint would be possible in order to draw from object and function benefits while never be forced to suffer drawbacks. Fortunately, research stimulated by pattern Void Value [Kühne96b] leads to an approach based on so-called tiles, which can be assembled to represent functions or objects depending on the most beneficial interpretation (see section 14.8 on page 252).

Language support for Function Object links to Void Value, Lazy Object and

Value Object. First, automatic initialization of function objects could be elegantly achieved by defining void function objects which return the desired result, i.e., typically a function object instance awaiting further arguments.

Second, a lazily evaluated function could be passed as an argument without demanding clients that they pass a dummy argument in order to evaluate the function when in need of the result. For functions returning a structure that has to be accessed (e.g., a stream) the difference does not matter (because access will trigger the evaluation only) but functions returning unstructured values like integers require language support for lazy evaluation to calculate their values lazily (see section 8.8 on page 122).

Third, a type system supporting the notion of value objects would allow passing a function object producing an integer to a client that expects a function object producing a numeric value. An EIFFEL compiler has to reject such a scenario since there is no way to tell it that, although generally invalid, in this case the use of a more specific actual generic argument is safe. If it were possible to specify that the result type of the passed function can not be written to (i.e., is a value), the compiler would be able to accept the more specific argument (see also sections 9.8 on page 155 and 14.5 on page 243). In other words, covariant output types are sound with regard to subtyping and there should be a means to tell the compiler when a generic parameter occurs in an output position.

Any language aiming at supporting function objects must be equipped with a sufficiently mature type system. That is why, the JAVA language fails to support function objects properly [Kühne97]. Although the EIFFEL languages provides the necessary support for genericity, its type system is not completely satisfactory with regard to function objects. For instance the identity function type-wise transforms any type into exactly the same type, i.e., inherits from the abstract **Function** class, constraining both input and output type to be the same. With EIFFEL we, nevertheless, have to equip class **Identity** with a type parameter, since there is no other way to constrain the input and output type of class **Function** to be the same.

Indeed, the lack to internally introduce universally quantified type variables has practical consequences: The natural place for a function composition method would be class **Function** since two functions are combined to produce a new function. The function composition operator $\circ$,

$$\begin{array}{rcl} f \circ g & = & \lambda x \rightarrow f\ (g\ x), \text{ has type} \\ \circ & :: & (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c, \end{array}$$

i.e., the second function's result type (b) matches the first function's argument type, and the resulting composition function goes from the argument type of the second function (a) to the result type of the first function (c). Hence, we would like to code the method using EIFFEL in class **Function** as follows:

```
class Function[B, C]
...
infix "|o|" (f : Function[A, B]) :
                ComposeFunction[A, B, C] is ...
```

Unfortunately, this is not possible since the generic type A is not known within **Function** and there is no other way to introduce it than by adding it to the generic class parameters. This, however, is not feasible because it would create the necessity to declare all functions with three types, regardless whether they are used for function composition or not. Inevitably, this means that function composition can not be implemented as a method but must be modeled as a function object itself.

One approach to solve the above problem would be to allow the declaration of universally quantified type variables for individual methods, akin to C^{++}'s template methods. Indeed, C^{++} does allow function composition to be implemented as a method and functional programming languages also successfully use parametric polymorphism to statically type function composition and similar cases. Similarly, in contrast to EIFFEL's explicit treatment of genericity, a C^{++} template implementation for **Fold** would not force the user to multiply declare a fold function object for each different type instantiation as in

```
fold1 : Fold[Book, Integer];       -- count books
fold2 : Fold[Book, List[String]];  -- produce titlelist
```

The result type of a fold function varies with the type of the initial argument for folding — effectively a new function type is created by each fold usage — but the type could be inferred from the initial argument without requiring the user to explicitly declare it up-front. Thus, a single fold declaration (just stating to fold over books) could be used for both applications above.

Another approach for greater expressiveness in the type system is to interpret the result type of function composition to be *dependent* on the argument type of function composition. In fact, by means of its covariant method redefinition supporting type system, EIFFEL from the start enabled to express the dependency of argument types on receiver types. For instance,

```
is_equal (other: like Current): Boolean is ...
```

specifies that the argument type (of `other`) varies with the receiver type. Among the innovations of EIFFEL, version three, was the possibility to express result types in terms of argument types. For instance,

```
clone (other: General): like other is ...
```

anchors the result type of `clone` to its argument type. This feature, called "expression conformance" [Meyer92], allows the declaration of `clone` in class **Any**[1] once and for all, while allowing statements like

```
  local a, b : Book;
...
  a:=clone(b);
```

[1]Class **General** to be precise.

In the above code the compiler can type-check the assignment since it does not rely on a static anchor to **General** but can assume the result type of `clone` to be same as the type of *expression* `b`. Unfortunately, it is not possible to use "component types" for expression conformance, e.g.,

```
apply(a : Any; f : Function[like a, R]) : like R is ...
```

does not work although anchoring the argument type of function argument `f` works, anchoring the result type of `apply` to the result type of the function argument (`R`) does not. Again, the generic variable `R` is not known, although, by means of expression conformance, could be taken from the actual argument to `apply`.

The next logical evolutionary step in enhancing EIFFEL's type system, hence, would be to support the specification of dependent types including access to "component types" such as generic arguments or even attributes. The latter option would lead to a powerful type system which coincidentally would provide a satisfying solution for the covariant method redefinition problem in EIFFEL [Shang94, Shang95a][2].

14.3 Translator

The most pressing issue with regard to language support for Translator is the possibility of generic functions failing to find an appropriate specialized function at runtime. Since multi-dispatch is only emulated by using the Generic Function Object pattern, without further support there is no way that type safe application of generic functions can be ensured statically. Although the implementation presented (see section 12.9 on page 214) avoids type-switch statements which require maintenence and even allows the addition of specialized functions at runtime, the introduction of a new type in the generic function domain will cause a runtime error if a generic function encounters an instance of it.

In fact, historically multi-dispatch supporting languages have not been statically typed either [Bobrow et al.86b, Shalit et al.92, Chambers92b] although early work on combining multi-dispatch with static typing exists [Agrawal et al.91, Mugridge et al.91]. Nevertheless, static type checking of multi-methods is highly desirable and possible solutions have been presented in the meantime [Chambers & Leavens95].

Another common counterargument towards multi-methods is that they break encapsulation by accessing internals of multiple abstractions. Especially the incarnation of multi-methods in CLOS with its unprotected data abstractions[3] and freely addable functions exemplifies the softening of an object-centered, encapsulation promoting, single-dispatch approach. The CECIL language tries to maintain encapsulation by conceptually assigning one multi-method to all abstractions that

[2]Unfortunately, the creator of EIFFEL, Bertrand Meyer, can not imagine EIFFEL's type system to develop into that direction [Meyer95] and proposed an alternative that simply and, therefore, restrictively bans the coincidental presence of covariance and polymorphism [Meyer96].

[3]Their is a package mechanism, though.

it dispatches on. In other words, encapsulated abstractions are retained and share multi-dispatching methods with other abstractions. In addition, however, it seems only natural to separate implementation detail access from method lookup. The fact that the dynamic type of an argument is used for method lookup should not imply opening the encapsulation barrier of the formal argument type. It should be possible, to specify arguments to be contributing to dispatch and/or opening encapsulation, independently from each other. Note that while it might be sometimes useful to declare a method in a single-dispatch language to refrain from using any implementation details of its abstraction, i.e., to use public features only, the need for such a change protection is much less needed in single-dispatch languages. Therefore, these, reasonably unify the two aspects.

Language support for generic functions, i.e., functions that dispatch on the dynamic type of their argument(s), would make language support for algebraic datatypes superfluous. The PIZZA language allows the convenient definition of algebraic datatypes and introduces a switch statement for constructor case analysis in functions [Odersky & Wadler97]. Instead of introducing constructor switch statements it seems much more reasonable to code the switch statement as a generic function. Datatype constructors can be represented as concrete subclasses to the abstract datatype interface class and generic functions would automatically do the necessary case analysis. The advantage of this proposed approach over the PIZZA solution shows, when a new constructor is added to a datatype. The PIZZA solutions requires us to change existing code, e.g., the algebraic datatype definition class, to add a further constructor and any function containing a switch statement. With the generic function approach we just add another concrete class to the system and add the corresponding specialized functions for each function on the extended datatype. Since code is added only, without the need to change existing code, the generic function approach to algebraic data types perfectly supports the *open-closed* principle of software engineering [Meyer88].

The initial motivation for algebraic types in PIZZA was to make operation extension as easy as constructor extension. The latter is supported by object-oriented design and, for instance, allows easy addition of language constructs to a language interpreter prototype. Distributing interpreter operations (e.g., print and evaluate) over the constructors makes it easy to add constructs but relies on the fact that the operations do not change and are not extended. If, alternatively, the language is already mature but the aim is to experiment with new and/or alternative operations (e.g., various type checkers), it is more beneficial to regard language constructs as constructors and define functions on them. As already explained above, generic functions cover the second case even better than support for algebraic types. Still, what if the language is immature and the environment experimental? Unequivocally, both above mentioned decomposition strategies are desirable then. When changing an operation one should be able to adopt a functional view and when changing a language construct the object-oriented interpretation is more beneficial. Again, a solution to that dilemma is offered by the tiles approach, presented in section 14.8 on page 252.

14.4 Transfold

The Transfold approach to iteration holds a number of implications for client software and library organization. Viewing data structures as maps (with internal iteration) rather than indexable structures (with external iteration) increases the level of abstraction [Pepper92]. If *ordered* iteration is necessary at all, it is deferred to the internal iterator. Copy semantics for structures (including iteration results) may help to avoid aliasing. For instance, implementing matrix multiplication with external iteration and side effects may not work anymore if the target of the operation happens to be one of the arguments.

The discussion in chapter 10 on page 163 actually developed a general framework for data interpretation, construction, and transformation. The concept of structural interpretations producing streams that are consumed by internal iterators, was found to extend to issues like creating data from manifest constants, transforming structures into each other, and data persistence.

In fact, assuming language support for all other functional patterns, no further support for Transfold is required. This is not a reason, though, to exclude Transfold issues from a language design discussion. The borderline between languages and library organization and frameworks respectively, has become more and more fuzzy. Especially SMALLTALK makes this evidentially clear. While it takes only a short time to learn the language, it is a major achievement to aquire expertise on its rich library. Learning the operations available on collections, for instance, is comparable to learning various loop constructs in other, less library oriented, languages. In the future, more extendible languages will further the trend to shift organization principles from languages to libraries.

The uniformity by which Transfold lets us flexibly interpret data structures suggests to facilitate the "folding" of structures by a suitable library organization. From this point of view, it seems natural to have an **Inductive** class very high in the hierarchy. Data structures of various branching arity (such as linear structures, trees, and graphs) but also abstractions like grammars, could be accessible via this interface.

The general nature of folding allows expressing many operations (see section 10.3 on page 173, *Versatility*) with just one iteration primitive. This is in contrast to EIFFEL's plethora of iteration features (do_all, do_if, do_while, do_until, etc.) in its iteration classes [Meyer94b] that still can not cover all cases. Although EIFFEL supports only one language loop construct, its library approach to iteration apparently does not allow such a minimalistic solution. While folding is not a computationally complete function[4] it is sufficiently general to rely on it as the basic iteration principle. In fact, the combination of creating a structure from a source (unfold) and subsequent processing (fold) — a so-called hylomorphism — is computationally complete [Pardo98b].

Unfold operations are needed when the computation is not driven by the structure of the argument but by the structure of the expected result. For instance, parsing is a tree structured process although the consumed structure is

[4]Not every computable function is expressible as folding [Kieburtz & Lewis95].

linear only [Kieburtz & Lewis95]. The Translator pattern lends itself to implement unfold operations. Indeed, structure explorations can be represented by unfolds [Pardo98a]. If the resulting streams are processed by folds — as realized in the Transfold pattern — the resulting operation is a hylomorphism. Automatic program transformations even allow elimination of the intermediate list between unfold and fold to be constructed at all [Gill et al.93]. Eliminations like this, however, critically rely on regular operations to be used to build and consume lists. In summary, the above observations suggest to organize libraries according to the algebraic properties of their structures and to employ unfold and fold operations for their interpretation.

There are a number of reasons as to why it is beneficial to use a fixed set of prefabricated functional forms rather than encouraging user defined recursion for interpreting structures. This view emerged from the *squiggolist* school that aims at a formalism that allows banning user defined recursion in favor of functional forms that are well controlled and amenable to program transformations[5].

Relying on a set of functional forms (e.g., fold) is advantageous in many ways:

- Algorithms using the forms have a concise, readily understandable structure as well as a determined complexity in time and space. Any programmer familiar with the functional forms will understand the algorithm by just looking at the essential parameterized parts.

- Well-known laws for the functional forms may be exploited to transform programs. For instance,

  ```
  Result:=fold @ sum @ 0 @ (map @ (times @ 2) @ list);
  ```

  can be transformed to

  ```
  Result:=2*(fold @ sum @ 0 @ list);
  ```

  using a free "fold-fusion" theorem that is derivable from the signature of fold [Wadler89]. More laws can be found in [Meijer et al.91].

- The predetermination by the available forms may lead programmers to concise, elegant solutions (see section 10.7 on page 176, *High-level mind-set*).

- Using proven recursion schemes frees the programmer from caring about termination, memory demand issues, etc. Many small, albeit disastrous errors can be avoided this way (see section 10.2.1 on page 163, mentioning the Mariner space-probe accident). The decreased number of structure explorations may even make their formal verification feasible. This would improve the overall reliability of a program considerably, when compared to a scenario of multiple, distributed external iterators.

[5]An idea rooted in John Backus' FP system [Backus78].

- Algorithms written as a combination of parameterized combinators can be easily varied. For instance a *tournament-sort* combinator taking two reduction strategies as parameters can express *InsertSort* [Bird & Wadler88], *TreeSort* [Floyd64], and *ParallelTournamentSort* [Stepanov & Kershenbaum86], just by using different combinations of the two reductions operators [Kershenbaum et al.88]. *ParallelTournamentSort*, actually, has very desirably properties, which demonstrates that using standard combinators does not necessarily imply inefficient solutions but, on the contrary, may help to discover better solutions. Another example supporting this claim is *MergeSort*. The naïve custom implementation, which divides a list into a left and a right part, is far inferior to the solution with combinators based on *parallel-reduce* [Kershenbaum et al.88].

- Since functional forms can be defined on an inductive interface, the addition of a new datatype automatically equips it with the available functional forms.

Fortunately, the iteration framework suggested by Transfold easily allows introducing new iteration schemes or utilize a customized scheme in case, e.g., primitive recursion may achieve better efficiency than folding.

In general, internal iteration is better suited for parallel execution since, unlike a custom external iteration, the iteration process is guaranteed to be encapsulated [Baker93]. In particularly, functional forms such as *map* or *parallel-reduce* allow parallel evaluation. In fact, there is a tradition in the area of parallel computations to employ fixed evaluation forms called algorithmic skeletons [Cole88, Cole89, Darlington et al.91, Boiten et al.93, Grant-Duff94, Darlington et al.95]. Interestingly, an object-oriented language, enhanced with the functional pattern system, fulfills all requirements towards a skeleton supporting host language such as imperative, hence, efficient statements, higher-order functions, partial parameterization, and polymorphism [Botorog96]. Provided the library implements functional forms by exploiting parallel hardware, the algorithms using the forms automatically make good use of available parallelism.

An interesting, competitive iteration approach is set out by the SATHER programming language. Language support for iterators — in the form of a restricted kind of coroutines — allows defining structure exploration within structures but still allows flexible, external iteration style, consumption [Murer et al.93b]. The open questions are which style (passing functions or coroutine-loops) is more expressive and understandable, and whether possible code optimizations by the SATHER compiler justify the requirement for an additional language concept.

With regard to EIFFEL's merit to support the functional pattern system, again the need for dependent types or universally quantified type variables occurred. Whereas **Transfold** in EIFFEL has three generic parameters (see section 10.9 on page 182), only one is actually required. The other two could be inferred from transfold's arguments.

The highly generic nature of transfold also revealed the lack of a subtyping relationship between the numeric types (e.g., integers should be subtypes of reals) and the need for a class that specifies that all its descendents conform to class **Numeric** and class **Comparable**. Using EIFFEL it is not possible to retrofit such a class into the library. Already defined classes like **Integer** qualify as descendants of that class but can not (and should not) be modified to inherit from the new class. This experience provides an argument for SATHER's *superclass* concept that allows specifying which old classes qualify as descendents for a new class.

In the attempt to write functions that operate polymorphically on numeric values, EIFFEL's special treatment of arithmetic types was encountered. Specifying the constant `0` does not work for generic functions that can also be instantiated to be **Double** functions because `0` is always interpreted as an integer and is never promoted to a real or double value. Anyone trying to integrate complex numbers into the library also notices that EIFFEL's *balancing rule* [Meyer92] gives more privileges to the language designer than to its user. In the mentioned case I could actually define the function generically by referring to `number.zero` and declaring `number` to be of the generic type. Nevertheless, such experiences question the maturity of arithmetic type treatment in EIFFEL.

Pattern Transfold strengthens the case for Lazy Object language support. Lazy streams are a prerequisite for efficient data exploration. If methods, in contrast to explicit function objects, actually perform lazily, the need for a parallel hierarchy of iterators (or explorers to be precise) will be made redundant. With lazy methods, structure exploration can be defined at the best possible location, inside and internal to structures. Furthermore, functions passed to transfolds should not notice any difference when stream contents are lazy. There should not be the need to wrap unstructured values with argumentless functions in order to achieve their lazy calculation (see 10.7 on page 176, *Lazy Functions*).

14.5 Value Object

The problems induced by reference semantics are well known. In SMALLTALK the sometimes unwanted and especially unexpected effects of aliasing are said to be a bigger source of problems than the lack of a static type system. For instance, the code

```
aPoint := button getTopLeft.  "get reference location"
aPoint y: (aPoint y - 3).     "move three pixels up"
aToolTip setTopLeft: aPoint.  "set toolbar above button"
```

does not only place `aToolTip` three pixels above a `button`, but also, unintentionally, moves the button [Harris97]. Of course, a more functional version would have been in order:

```
aPoint := button getTopLeft.
aToolTip setTopLeft: (aPoint - 0@3).
```

The latter version works as expected since the minus operator produces a new value.

Surprisingly, the prevention of aliasing, or advice how to manage system state does not seem to be an issue in the major software engineering methodologies. Whenever state is (inadvertently) encapsulated, e.g., by subsystem refactorizations [Wirfs-Brock et al.90], it is for the sake of decoupling interfaces, rather than controlling side effects. The Facade pattern [Gamma et al.94] may also help to control subsystem state but is not advertised as such. Other state patterns [Gamma et al.94, Ran95, Dyson & Anderson96] discuss how to implement state dependent behavior but do not touch the issue of controlling state in object-oriented software.

Several points make aliasing more severe in object-oriented languages:

- A method with two parameters of different types **A** and **B** may still suffer aliasing if **B** is a subtype of **A** or vice versa.

- The state of an object is not only determined by its internal parts but also by the state of all objects it references. Object state, hence, is defined by the transitive closure over object connectivity. Therefore, two objects are effectively aliased if their transitive state closures have a non-empty intersection.

- Object state, which is permanent as opposed to temporarily allocated procedure parameters, allows aliasing to become static rather than dynamic. Standard procedure calls create only temporary aliasing that vanishes after call completion. With objects, references can be captured and interference may happen much later, with no direct indication of the problem source. Attempts to control aliasing, in fact, address static aliasing only because its effects are more severe [Hogg91, Almeida97].

- Ironically, object encapsulation may make it impossible to check for possible aliasing between a variable and an object component. If the object does not allow access to the component of interest, even the aliasing aware client can not prevent it [Hogg et al.92].

Clearly, the type system should support the notion of values. This would increase the opportunity for polymorphism since there are more subtype relationships between values than between mutable objects (see sections 9.8 on page 155, *Subtyping*, and 14.2 on page 233). Pattern Lazy Object could benefit, for instance, by letting an integer stream qualify as a numeric stream. An integer stream can not play the role of a mutable numeric stream since non-integer additions are possible. With a value interface to the numeric stream, however, the integer stream is perfectly suitable. Definitely, the type system should allow treating objects and values uniformly. JAVA "excels" in demonstrating the worst possible case, where basic types must be manually converted to objects and back to values for storing and retrieving them in reference containers, etc. Surely, if basic (value) types are treated with special care for efficiency considerations then this should take place without letting users of the language notice.

Apart from value types other means to control aliasing are available (e.g., “const” declarations) or have been proposed (e.g., unshareable references [Minsky96]). Value types — as consciously used for modeling in the BETA language [Madsen et al.93] — however, seem to be the most natural concept. The difference between values and objects can be argued for philosophically [Eckert & Kempe94] and seems much more natural than, e.g., approaches using the type system to restrict multiple references [Wadler90]. Efficiency is not necessarily endangered by the need to copy values occasionally[6] as compiler optimizations are possible [Hudak & Bloss85, Schmidt85, Hudak92].

Programming languages should eventually overcome the notion of pointers, even when they disguise as object references [Kühne96b]. Obviously, the intuitive semantics for assignment is usually copying and the intention to achieve sharing should be explicitely expressed as such. By the same token, the default semantics for “=” should be value, rather than pointer comparison. Choosing pointer comparison, EIFFEL, JAVA, etc., force their users to bother about the difference between strings and their references when the only reason why references come into play is the language designer’s aim to achieve efficient assignment semantics.

The natural consequence for a language featuring immutable values and mutable objects that aims at safe handling of objects, while allowing sharing when necessary, seems to be the support of two different assignment statements. The default statement preserves copy semantics and a less often used reference assignment achieves sharing. The view that should be adopted on the latter is less reference sharing oriented but rather portrays the notion of establishing a broadcasting relationship between an (publishing) object and all its (subscribing) references. In fact, SIMULA featured two different assignment statements for values and references to signal the difference in semantics [Dahl & Nygaard81]. However, since both were applicable to their own domains only, the value of that concept was very limited. Only the single type **Text** allowed both statements and, therefore, enabled a choice of semantics. For all other types, a syntactic documentation of the semantics was achieved but — since redundant — was conceived rather a curse than a blessing. If both operators are applicable to both domains, however, it would be possible to choose the appropriate semantics and the choice would be clearly documented in program code. Contrast this to, e.g., EIFFEL’s solution of one assignment statement whose meaning changes for expanded types. First, the default (sharing) is often non-intuitive and second, the declaration revealing the actual semantics is most likely not displayed in conjuction with the code in question.

C^{++} does make a syntactical difference between value and reference semantics through the use of pointers. Unfortunately, values are not polymorphic, i.e., late binding for method calls is not enabled. This renders the world of C^{++} values pretty useless in an object-oriented context. The EIFFEL approach to value types in the form of expanded types, has been criticized for not providing constructors and also prohibiting subtyping [Kent & Howse96].

[6]The most economic scheme seems to be copy by demand, i.e., copy when a shared value is modified (see figure 9.5 on page 159).

14.6 Lazy Object

Theoreticians value lazy semantics for the property of yielding at least the same results as eager semantics but possibly more, and for the ability to apply "backwards" substitutions — e.g., replacement of a constant 1 with *if true then* 1 *else f* — which is not possible with eager semantics (see section 1.2.4 on page 14). Chapter 8 on page 115, furthermore, showed a number of practical benefits resulting from lazy semantics:

- The reputation of functional programming languages to enable a declarative programming style, may deceptively explained by pointing out the pattern matching style of function definitions. Yet, the latter is, in effect, syntactic sugar only and true declarative programming style is enabled by
  - not being forced to be explicit about artificial upper bounds,
  - allowing up-front declarations, and
  - being able to specify data dependencies only, without worrying for their resolution (see section 8.4 on page 119).

  Even statically circular definitions are possible as long as a dynamic resolution is possible. Specifying dependencies, rather than implementing aggregation relationships, may also help in achieving memory friendly behavior for parallel programming [Kashiwagi & Wise91].

- Data consumption can be separated from data generation. Still, termination control is available for both producers and consumers. Hence, traditionally opposing design goals, efficiency and modularity, are effectively reconciled since partial evaluation is enabled across module boundaries. Moreover, space efficient, interleaved execution of producer-consumer pipelines is automatic.

- Lazy evaluation allows initialization code to be placed at the best possible position; the corresponding data abstractions [Auer94]. Additionally, lazy attribute evaluation opens up the possibility of overcoming current limitations in dealing with value types. EIFFEL does not allow recursive references in expanded classes, e.g., a **Person** referring to another **Person** as its spouse, since that implies an infinite initialization sequence and space requirement [Meyer92]. Lazy initialization would unfold such potentially infinite structures only as much as necessary. This is particular interesting for Value Object and Void Value language support because of potential self-references.

- A system based on streams is determined by the types of modules and the interconnections (streams) between the modules. Stream processes, such as filters, may be executed in a threaded fashion thanks to the value semantics of streams. The increased modularization achieved by streams can be exploited to distribute development efforts nicely into autonomous groups [Manolescu97].

Lazy evaluation plays one of the key roles in the Transfold pattern. It enables the separation between structure exploration and consuming iterators. Most intriguingly, the combination of two functional concepts (higher-order functions and lazy evaluation) solves an as yet unsuccessfully tackled object-oriented design problem: How to have an iteration cake (safety by internal iteration) and eat it too (flexibility through external iteration)? The answer is given by the Transfold pattern. This pattern shows how lazy exploration streams, actually, use continuations[7], without explicitely involving any peculiar continuation passing style.

Typical objections against lazy evaluation are

- memory management overhead, caused by suspensions and
- aggravated debugging, caused by unexpected evaluation orders.

Both arguments seem to have less weight in an object-oriented context, though. First, garbage collection is needed for objects anyway (see section 14.2 on page 233). Second, the complexity of message sends along a typical network of interacting objects is presumably not any simpler than a lazy resolution of dependencies.

The restriction of laziness to values and functions (see section 4.2.1.1 on page 57) nicely resolves the conflict between destructive updates and lazy evaluation. As a consequence, value semantics for function parameters has been postulated, though. Otherwise, updates to objects after they have been passed as function arguments would affect function results (see section 7.9 on page 105, *Call-by-value*). Research in the area of functional programming, furthermore, shows the way to even aim at lazy effects (see section 4.2.2.1 on page 59).

Especially extensible languages require some means to defer expression evaluation. Lazy semantics would allow programmers, extending the language, defining the evaluation strategy and order for their new language abstractions themselves. This approach seems more aesthetically pleasing than the usual facilities like macros [Winston & Horn84] or meta architectures [Chiba & Masuda93].

With regard to language support for lazy values it was already mentioned that an emulation with a dummy parameter for pattern Function Object, `unit` parameters in ML, and explicit `delay`s in SCHEME make the client of such functions aware of their laziness and forces different versions to be written for non-lazy arguments. Moreover, it is not possible to produce unstructured lazy results since these can not defer their evaluation with access methods, as structured values may do (see section 14.2 on page 233).

Returning to theoretical considerations made at the beginning of this section, it is interesting to observe that both objects [Jacobs98] and lazy streams [Jacobs & Rutten97] can be formally explained by coalgebras. Both objects and streams hide inner details and specify destructors only. To the best of my knowledge, however, objects and lazy semantics, as yet, have never been united in a single language.

[7]Stream elements are continuations to further structure explorations.

14.7 Void Value

Following the virtue of making programs robust, i.e., let them react sensibly in case of unusual circumstances, previously simple code may become cluttered with safety checks (e.g., see code examples 11.1– 11.6 starting on page 192). For this reason, the concept of exception handling was introduced. Interestingly, the reasons that may cause an exception to be raised in EIFFEL are:

(a) Hardware

1. Hardware signal for user requests, such as hitting a “break” key.
2. Memory exhausted, integer overflow, etc.

(b) Software

1. Programmer generated exception.
2. Assertion violation.
3. Applying an operation to an unattached reference [Meyer92].

Taking into consideration that the first item in category “Hardware” and the first two items in category “Software” represent welcome aids, and that there is effectively nothing that can be done against hardware limitations, one is left with just one exception cause: Trying to call a feature on a void reference.

For certain, chapter 11 on page 191 demonstrated a better way to deal with such situations by avoiding them in the first place. A remaining discussion is *how* to avoid void references. There are two dual alternatives:

Void Value	Provide a void class for each[8] class in the system [Kühne96b].
Nullcase	Provide a nullcase for each function in the system [Sargeant93].

An example of using a nullcase in an UFO [Sargeant93] function definition is:

```
length: Int is
  nullcase:  0
  otherwise: 1 + self.tail.length
```

How do nullcases compare to pattern Void Value? As a consequence of the encapsulation breaking properties of pattern matching, function definitions in UFO are sensitive to the introduction and change of null values. For instance, if a set of functions is designed to always operate on non-null lists and later it is discovered that indeed null lists may also occur, then all function definitions must be changed to include the nullcase branch. In contrast, methods of an object do not need to be changed, since they operate on a non-void object per definition. The necessary addition here consists of adding one **VoidClass**.

[8]In analogy to an object root class, there should be a void value root class which is inherited by all void classes. Hence, only deviations actually need to be specified.

Similarly, it may occur that a former null value should be transformed into a proper object, just as enumeration types are sometimes transformed into class types [Sargeant et al.95]. Consider a tree representing terminal leafs as void values. Now, we want to enhance the tree with a display method and therefore add attributes to the leafs, which store the display position[9]. In fact, the example describes the conversion of Void Value to Composite. A void class is easily converted into a standard class, but nullcases will not match object instances and must be repackaged into a class definition.

On the other hand, the introduction of a new function or the change of an existing one is more local with the UFO approach. Nullcase and the regular case are edited at one place (the function definition) whereas one may need to visit both void value and object class, in order to do the change. With both cases at one place it is also easier to see what the function does as a whole.

Nevertheless, the extraction of the nullcases into one class description produces an entity of its own right:

- The void value class acts as a compact documentation of base case-, default-, and error-properties of a particular type.

- When trying to understand a type, looking at the void class immediately communicates the set of methods that create exceptions or implement implicit creation or delegate to other definitions, etc.

- With all null properties concentrated at one place it is easier to check for consistency, e.g., to make sure that a terminal node does not answer count with zero, but leaf with true.

- The void value class provides a place for (private) initialization functions needed and possibly shared by void behavior methods.

- Changing the void-behavior of a set of functions is regarded as providing new or changed void data, without changing function definitions.

- It is possible to have multiple void values and to dynamically choose between them. With the nullcase approach, each time a new set of functions (that must repeat the regular case) must be provided.

An example for the utility of separately defining null- and regular cases is the addition of numbers (assuming a void number shall behave like zero). The void value method simply returns the argument (the number to be added), regardless whether the regular case would have used integer, or complex arithmetics. Void complex values, thus may inherit this method from void integer values.

In turn, one may want to keep the regular behavior of a type, but vary the exception or termination behavior. Folding an empty list may result in an error- or default value object, depending on which was given as the initial argument for folding.

[9] If we do not want to treat leafs as Flyweights.

UFO's lacking ability to redefine null- and regular cases independently, is likely to be removed [Sargeant96b]. However, inheriting null- and regular cases separately, implies the loss of locality of a function definition. As in the case of void values, it will not be possible anymore to look at the full definition at once. Finding the actual definition becomes harder if redefinition in a hierarchy of definitions is allowed[10].

Even if inheriting and overriding of null- and otherwise clauses is possible, still the problem of consistently changing the behavior of null values remains. Consider a list of books found during a library retrieval. An empty result at the end of a non-empty list displays nothing and provides no actions. If we want to change that behavior into printing a help message (e.g., suggesting to broaden the scope of search) we can provide the corresponding action (invoking the help system on clicking) in the same void value class definition. In other words, changes to void behavior that affect more than one function are still local.

Note that void values are not restricted to procedural abstraction only. Either by providing an is_void method or by using RTTI it is possible to check arguments or attributes for being void. That is, as in UFO, it is possible to use the "voidness" of arguments (i.e., inspect their constructor) in order to do optimizations, such as returning the receiver of an append message if the argument to be appended is void[11].

An argument in favor of nullcases is that they do not produce inheritance relationships. In fact, there are good reasons for both approaches. Assuming anything else equal, there is also the motivation to use just one instead of two concepts for dynamically looking up definitions. A void value is simply a hierarchy sibling of standard objects and other base-, default-, and error values.

We have seen that trying to achieve automatic initialization of references to void values involves some additional complexity (see Figure 11.8 on page 198) and introduces both storage (for **DefaultReference**) and computation (for delegation) overhead (see section 11.8 on page 197, *Reference Initialization*). Consequently, language semantics should not initialize references to Nil, but to their associated void values. As a result, it would become easier to force program execution to "stay in the game", i.e., stay within the domain of language semantics without creation of runtime aborts or implementation dependent effects [Appel93]. With regard to the caveat of unnoticed errors, introduced by not stopping on not explicitly initialized data (see section 11.7 on page 196, *Initialization Errors*), one may still provide no, or exception generating behavior, for void values, in order to force a brute but early detection of unintended void values.

Applications for Void Value that go beyond what most languages allow to emulate are "Initialization by declaration" and "Implicit creation". Even if it is possible to tie initialization (assigning useful default values) to creation (attaching an object to a reference), Nil values still exist "between" declaration and initialization. Void

[10] EIFFEL, in particular, provides the *flat* format for classes that could be used to resolve all inherited and redefined methods of void value classes.

[11] With procedural abstraction only, one is forced to add each element of the receiver to the argument in any case.

Value allows transferring the concept of automatic initialization known for value types to reference variables, since even variables of abstract type can be attached to a proper void value. An abstract class definition thus not only provides an interface and template methods, but optionally also a void value description.

Beyond the implicit creation of void values, one may extend the concept to allow automatic initialization to any object, i.e., implicit object creation. In analogy to the creation clause for methods in EIFFEL, one may select a creation subclass to be used for implicit reference initialization. In contrast to void values, the thus created instances, could be used to capture and hold state right away. Instead of forcing the programmer to guarantee the creation of an object instance in advance, it appears very useful to allow implicit creation of object instances. Note that "Initialization by declaration" refers to the automatic attachment of references to void values, while "Implicit creation" refers to the automatic conversion of void values to objects. Whenever, a void value can not handle a message (e.g., state has to be accepted), it replaces itself with an appropriate object. Such an automatic conversion would make explicit creation calls superfluous (see code example 14.1).

```
if array.item(key) = Void then
  !!set.make;
  input_array.put(set, key);
end
array.item(key).put(data);
```

Figure 14.1: Code example: Initialization.

"Initialization by declaration" and "Implicit creation" can work together to implement *Lazy Initialization*, which avoids exposing concrete state and initialization overhead, allows easy resetting, and provides a proper place for initialization code [Auer94]. Ken Auer's solution requires to test variables for Nil values, i.e., for being not yet initialized. Automatic initialization of references, however, allows applying messages to void values (instead of Nil), whereas automatic conversion to objects allows the implicit creation of results on demand. The latter step, may not even be necessary if the required results do not need to hold state. As long as no object functionality is needed, void values suffice and do not create storage overhead for "empty" objects.

Summarizing, Void Value replaces void reference handling through *type abstraction* with providing default cases through *procedural abstraction*. The former is related to functional programming, while the latter is related to object-oriented programming [Reynolds75, Cook90]. While void values are seemingly superior to nullcases, the latter concept has its merits as well. As previous sections in this chapter already observed, in general, both views are equally useful and should be available on request. The solution to this goal is presented in the next, final section.

14.8 Conclusion

A language that doesn't affect
the way you think about programming,
is not worth knowing.
– Alan J. Perlis

The impact of the functional pattern system on language design, as detailed in the previous sections, is summarized in table 14.1.

Name *language concept*	*Motivation / Effect*
Function Object *higher-order functions*	user defined control structures, component programming, functional extensibility.
Translator *multi-dispatch*	heterogeneity problem non-intrusively solved, datatype extensibility in the presence of functional extensibility.
Lazy Object *lazy evaluation*	modularization, declarative programming style, iteration framework, recursive value initialization, user defined control structures.
Transfold *algebraic library organization*	library organization with inductive and coinductive types, emphasis on hylomorphisms (*fold* $\circ$ *unfold* operations), operation implementation with parallel hardware.
Value Object *value semantics*	default value assignment, additional broadcasting assignment, value subtyping.
Void Value *void behavior*	void behavior instead of exceptions, reference initialization.

Table 14.1: Pattern implications on language design

Pattern Function Object adds functional decomposition (i.e., functional extensibility) to object-oriented design (see section 14.2 on page 233). It is an essential means for the construction of components [Jazayeri95] and black-box frameworks [Johnson & Foote88]. Translator makes datatype extensions feasible in the presence of a functional decomposition by means of extensible multi-dispatching functions. Heterogenous data (in a structure or coming from, e.g., a database or network) can be dealt with, without imposing an interface on the respective datatypes (see section 14.3 on page 238).

Pattern Lazy Object fosters a declarative programming style, enables the iteration framework described by Transfold, allows value objects to have recursive references, and gives designers control over the evaluation strategies for new control abstractions (see section 14.6 on page 246).

Pattern Transfold builds upon the above concepts and provides good arguments for an algebraic library organization (see section 14.4 on page 240).

Value Object and Void Value collaboratively argue to abandon the traditional notion of a reference. The default semantics for assignment should be value semantics while sharing must be explicitly achieved with a broadcasting assignment (see section 14.5 on page 243). An ubiquitous Nil value is abandoned in favor of type specific void behavior. References are automatically initialized with their corresponding void values. Hence the asymmetry between values, which have useful initialization values, and references that do not, is removed (see section 14.7 on page 248).

As a general observation, there is a demand for a type system that meets the flexibility which is expressible with the functional pattern system. In particular, support for expressing type constraints and extended subtyping opportunities were found to be necessary in addition to EIFFEL's capabilities.

Furthermore, EIFFEL complicates a functional style by achieving object creation by a statement, rather than creation expressions. Consequently, a single line like

$$\mathit{nonsense}\, x\, y \;\equiv\; (\mathit{real}\,((x,y)*(y,x)),\, \mathit{real}\,((x,-y)*(y,-x)))$$

expands to a small program:

```
nonsense (x : Number; y : Number) : Complex is
  local z1, z2, r1, r2 : Complex;
  do
    !!z1.make(x,y);  !!z2.make(y,x);
    !!z3.make(x,-y); !!z4.make(y,-x);

    !!r1.make(z1*z2);
    !!r2.make(z3*z4);

    !!Result.make(r1.real, r2.real);
  end;
```

The rational for having a creation statement is to accommodate the fact that object creation is a side effect and an expression should not create side effects [Meyer95]. I certainly second this argument but one should note that the "side effect" of a creation expression does not violate any existing invariant in the system. In other words, unlike typical side effects, object creation can do no harm to other clients.

The only unwanted effect could be an exception raised due to insufficient memory. It does not seem convincing to reject creation expression on the basis that such an exception should rather be raised by a statement.

Of course, the above code examples do not only compare creation statements to creation expressions. The notational economy of the functional definition also largely draws from the fact that values can be created with manifest constants. A (tuple representation of a) complex value in the functional definition is created with just five characters, whereas the corresponding EIFFEL creation statement is much more verbose.

This suggests to provide at least one way to express constant values (e.g., syntax for manifest array constants) whose type can be converted into other datatypes (e.g., collections), that is, to adopt the SMALLTALK philosophy [LaLonde94]. While this issue may appear insignificant, it extends to the before mentioned library organization. Linear manifest constants could be used to create any other datatype, even branching ones, like trees. The already mentioned notion of an "unfold" can be used as a creation method in datatypes. It would work as a stream reader that constructs any particular datatype by consuming input from a manifest constant, output of a different datatype, etc. For instance,

```
t := Tree[Integer].unfold({Node:
                                {Node: {Leaf: 1} {Leaf: 2}}
                                {Leaf: 3}
                              }).
```

With this point of view, "unfolds" are parsers, while "folds" are interpreters.

Several of the previous sections in this chapter arrived at the conclusion that neither datatype extensibility (object-orientation) nor functional extensibility (functional programming) are sufficient on their own (see sections 14.2 on page 233, 14.3 on page 238, 14.7 on page 248). The number of attempts on the Visitor pattern [Gamma et al.94, Nordberg96, Martin97, Krishnamurthi et al.98] indicate the importance and difficulty of adding functional extensibility to object-oriented design. However, from the above versions only the last achieves to reconcile datatype extensions with functional extensibility. All other versions require changes to function code when new datatypes are added. This, however, means that one is left with an "either-or" choice. Without Visitor, functional extension is awkward, whereas with Visitor, datatype extension is awkward.

The Extensible Visitor pattern [Krishnamurthi et al.98] avoids this by enabling late binding on function creation. Thus, function creation in existing function code can be adapted to new requirements — i.e., to the new functions needed for the datatype extensions. Using inheritance to override function creation, existing functions can be extended. There is a need to create functions within visitors, and therefore the problem of prematurely fixing the function version, because it is sometimes necessary to pass different arguments to visitors for sub-traversals or one even needs to change the visitor type (e.g., using identifier analysis during type checking). Unfortunately, Extensible Visitor suffers from problems with the original Visitor version [Gamma et al.94], i.e., it implies the pollution of the datatype interface, introduces a dependency cycle, and does not support partial visitations [Nordberg96, Martin97].

I argue that the notion of a multi-dispatching function, used by Translator, is much more natural then the explicit binding of data and functions via a visitor-style Accept method. Translator does not require datatypes to feature an Accept method and, hence, even allows functionally extending datatypes without source code access. Translator-style functional extensions do also cope with datatype extensibility, since the (generic) functions used for traversals never change. New functions are simply introduced by extending the set of specialized functions available to a generic function (see chapter 12 on page 201), that is, the state of the traversal function is changed, not its type. Ergo, no code needs to be updated in case of datatype extensions.

Revising the above scenario, we have datatypes — i.e., an abstract interface plus concrete subclasses for each datatype constructor — which are interpreted by multi-dispatching functions, i.e., generic functions with associated function packages (see chapter 12 on page 201). While this achieves extensibility in both dimensions, (external) functions and datatypes exist in isolation. One can not look at a datatype and see what external functions it supports. Obviously, that implies that external functions can not be changed in an object context, i.e., as pattern matching branches distributed over constructor objects. Conversely, it is also not possible to alter object methods (internal functions) in a functional context, i.e., as a monolithic algorithm. In other words, functions implemented according to the Interpreter pattern (i.e., internally) can not be treated as functions, whereas functions implemented according to the Translator pattern (i.e., externally), can not be treated as being distributed over objects.

So, although we overcame the dichotomy between functional and object-oriented view for extensibility, we still suffer from the same dichotomy in terms of external (type abstraction) versus internal (procedural abstraction) function definition. An internally implemented function should be changeable as a single function without the need to visit all constructor parts. Conversely, the change or even introduction of a datatype should not require to update many existing external functions. One "object view" should gather all relevant pattern matching branches from external functions. Ideally, no commitment to either view, i.e., paradigm, would be necessary and either viewpoint could be adopted when appropriate. In other words, retooling (changing the paradigm) should not be expensive, although it usually is [Kuhn70].

The logical conclusion from the above is to adopt a more general decomposition strategy that allows the dynamic generation of an object-oriented or functional perspective. We already encountered this general view on decomposition in the form of table 4.2 on page 65. We simply construct a matrix (see table 14.2 on the next page) with columns for datatype constructors (c_j) and rows for functions on the datatype (f_i).

Each tile within the matrix defines the result of applying one function to one constructor. Taking a vertical stripe from the table, gives us the object-oriented view (a constructor subclass with functions). A horizontal stripe, gives us the functional view (a function defined on constructors) [Cook90]. Using an analogy from physics, we represent elementary particles with a matrix and let observers take a

		Datatype	
		c_1	c_2
Interface	f_1	$f_1\ c_1$	$f_1\ c_2$
	f_2	$f_2\ c_1$	$f_2\ c_2$
	f_3	$f_3\ c_1$	$f_3\ c_2$

Table 14.2: Tiles with object-oriented and functional dimensions

wave or particle view on the matrix.

The collection of constructors (topmost row) is a datatype, whereas the collection of functions (leftmost column) is an interface. A datatype defines constructors and an interface defines a set of functions.

Given just a datatype, we may add any interface, that is defined on a supertype[12] of it, e.g., provide different function sets (e.g., stack or queue interfaces) on lists. This is a functional view, where we add functions to passive data.

Given just an interface, we may supply any datatype, that is a subtype to that assumed by the interface, e.g., implement a list interface with alternative constructor sets. This is an object-oriented view, were we supply implementations to abstract interfaces.

In the envisioned — tile based — programming language, a programmer could declare datatypes and interfaces independently and rely on late binding of the missing respective part. Hence, both paradigms would be available. The combination of datatype and interface can be viewed from a functional perspective — there is one datatype with an associated set of functions — or from an object-oriented perspective — there is one abstract class with with several, constructor implementing, subclasses. In any way, it seems to be appropriate to speak of a concrete type.

Extensions are now simply a matter of adding a horizontal or vertical stripe to the table. That is, it does not matter whether we filled the table with horizontal strips (functions) or vertical stripes (constructors) before, we simply add a function or constructor. Maybe even more importantly, if we build the table column-wise (e.g., adding language constructs to a language), we can still alter an existing function (e.g., operation on the language, such as compilation or type checking) with a functional view, by locally altering a row. Conversely, if we build the table rowwise (e.g., extending language operations), we may still concentrate on individual language constructs, i.e., alter all functions for one constructor in an object context.

Again, using more conventional terms: The tiles approach makes it feasible to implement an interpretation on an abstract syntax tree as tree member methods since one interpretation could be edited as a single function, created by a func-

[12]The notions of supertype and subtype shall include the type itself.

tional view on a definition distributed over many types. In turn, pattern Translator could be applied even if the visited structure is unstable since a change to one tree member object could be done simultaneously to all interpretations, gathered by an object-oriented view on all interpretations.

All this, of course, depends on a supporting environment. A browser (tiler) must be able to provide each view on demand. Working on actual functions or objects, then, is just an illusion since only a virtual view is manipulated, with changes being appropriately distributed to the corresponding tiles. Functions and objects, however, actually exist in software as declared entities.

I conclude this outline of an approach that uses environment support to generated most appropriate views, with considerations about repeating the tiles concept on datatypes and interfaces respectively (i.e., the row and column types of table 14.2 on the facing page). Previously, tiles defined one function behavior for one constructor. Now, each inner square in table 14.3 defines one interface behavior on one datatype.

		Domain	
		D_1	D_2
Interpretation	I_1	$I_1\,D_1$	$I_1\,D_2$
	I_2	$I_2\,D_1$	$I_2\,D_2$
	I_3	$I_3\,D_1$	$I_3\,D_2$

Table 14.3: Concrete types with implementation and interface views

Alternative interfaces on one datatype (leftmost column) are different views on a datatype (e.g., a person or library member view on a student). In object-oriented terms they can be regarded as multiple abstract superclasses.

Alternative datatypes to one interface (topmost row) are implementation choices for that interface. An interface, therefore, is like an abstract class in object-oriented programming that defines an interface for several concrete implementations. Sets of implementers and views give rise to two concepts: Domains and interpretations. A domain is a collection of datatypes that can implement an interface. An interpretation is a set of views on a datatype. Note, that a square in table 14.3 corresponds to table 14.2 on the preceding page, i.e., is defined by editing and possibly adapting a tiles matrix. Table 14.3, hence, organizes concrete datatypes into combinations of interfaces with their implementations. The same approach that was useful to relate functions to data is now used to add implementations, define new views or roles, find implementations for an interface, change one implementation for several interfaces, etc. If we assume that one of the interfaces is a **State** interface to several **ConcreteState** implementations (see the State pattern in [Gamma et al.94]) then a browser (tiler) invoked on table 14.3 on the page before is an editor for the State pattern. Organizing interfaces and implementations this way may also extend into the area of subject-oriented programming [Harrison & Ossher93], where client-specific interfaces (views) play an important role.

If we further progress on the abstraction level, i.e., again fold the whole matrix to one square in a new matrix, we are in the position to define interpretations on domains (e.g., type-checking or compilation on a programming language). With just one sort of interaction (using a tiler) we, therefore, may ascent or descent abstraction levels and edit individual tiles (see figure 14.2). Note that a functions-constructors matrix on the lowest level corresponds to a class in object-orinted languages. A typical class browser does not even support the organization on the second level. Though traversing inheritance relationships is possible, these may or may not correspond to interface-implementation relationships. It is intriguing how useful the abstraction matrix on the top is additionally. Together, the three levels organize software into a three abstraction level hierarchy, which is amenable by one user interaction paradigm, that is, tile browsing. Note that nothing really prevents us from going further up the hierarchy. A set of interpretations could be called a tool and a set of domains might be a field, and so on. The further we proceed upwards from the bottom, the sparser the matrixes will be filled. All axis labels lend themselves to be used as types for the declaration of software entities in programs.

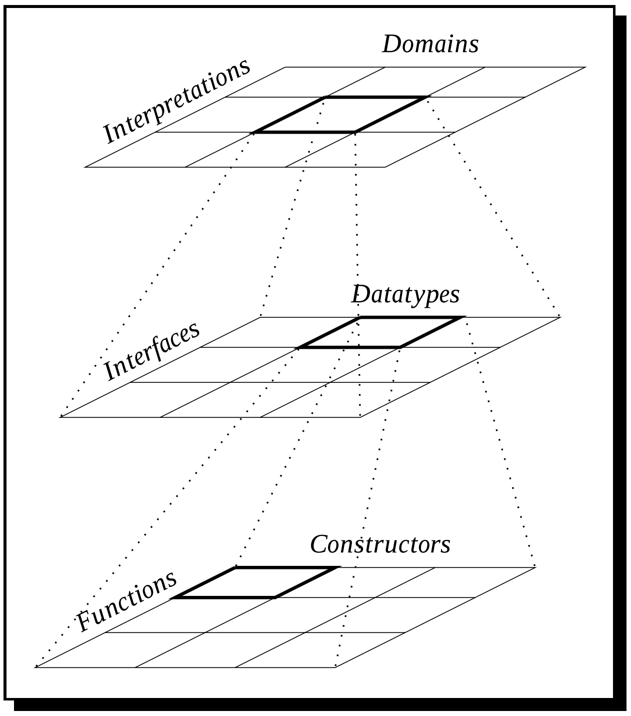


Figure 14.2: Hierarchical tiles editing

The usefulness of the above notions, their details, and the approach in general remains to be researched. Since the main emphasis of this dissertation is the functional pattern system — with a view to its implications on language design — this relatively advanced outline of an envisioned language and environment model must necessarily be incomplete. Though many important details have been left out, I nevertheless hope to have drawn a convincing proposal for future language and environment research.

The idea to lever languages by the use of tools has, of course, been exploited before. For instance, type inference can be added to languages with much simpler type system by means of context relations [Snelting86]. Monomorphic languages (e.g., PASCAL) can be used polymorphically with the help of an additional fragment system [Snelting et al.91, Grosch & Snelting93]. SMALLTALK does not require any syntax to separate class methods from each other since they are edited with a browser only one at a time. SMALLTALK's highly interactive environment is also highly intertwined with its dynamic type system and interpretation approach. Only in conjuction, language and environment provide a paramountly productive development tool[13]. Since SMALLTALK is a single-dispatch language only, a tool has been build to support the maintainance of multi-dispatching arithmetic operations [Hebel & Johnson90]. Industrially supported programming environments, though, merely scratch the surface (e.g., with syntax-highlighting) and do not exploit the available potential at all.

In conclusion, I showed that a well designed language, such as EIFFEL, makes the subsumption of functional programming feasible and partially offers good implementation support. However, I also demonstrated that functional patterns add real value and that there is plenty of room for improving support for functional patterns.

Ultimately, the functional pattern system induces a vision for a very advanced programming language with features such as higher-order functions, multi-dispatch, lazy evaluation, value semantics with broadcasting, and void behavior. The language uses an algebraically organized library and is based on the notion of tiles. Software organization principles are deferred to an associated environment, therefore, allowing dynamic paradigm changes.

[13]Curiously, SMALLTALK — being the first object-oriented language after SIMULA — has many functional characteristics, such as blocks, streams, internal iterations, and value (not reference) comparison for objects being the default.

Epilogue

Not only will men of science have to grapple with the sciences that deal with man,
but — and this is a far more difficult matter — they will have to persuade
the world to listen to what they have discovered. If they cannot succeed
in this difficult enterprise, man will destroy himself
by his halfway cleverness.
– Bertrand Russel

A functional pattern system is valuable in many aspects. The following sections conclude about the many facets involved by the successful attempt to capture the functional programming paradigm with patterns for object-oriented design.

Software design

Each of the six presented patterns is capable of improving today's software designs. Even those patterns that do not represent completely novel approaches, for the first time explicitely describe the applicability and resulting consequences of each technique in a functional pattern system context. To my knowledge, my work represents the first comprehensive attempt to examine functionally inspired techniques in terms of software engineering concepts.

Understanding object-oriented practice

In the context of the functional pattern system, many object-oriented practices appear as special cases of more general functional patterns. For instance, call-back functions according to the Command pattern are function objects[14] that do not take arguments after their creation. The acceptance of arguments after creation opens up a wealth of useful applications (see pattern Function Object on page 93). So-called iterator objects, coming into existence through the need to separate iteration action refinement inheritance from data structures [Meyer94b], may also be regarded as function objects with a non-standard application interface. Likewise, a visitor capable of visiting several node types, is a function object with several entry points.

[14]Procedure objects, to be precise (see section 7.10 on page 107).

Pattern Lazy Object describes what other benefits, apart from exchangeability, may be obtained from a pipes and filters architecture [Buschmann et al.96] (e.g., decoupling). Moreover, knowledge of Lazy Object immediately provides insight into the justification of the presence of streams in the SMALLTALK library. They help to avoid repetition of computations, e.g., when copying structures. In addition, Lazy Object suggests to also use them for partial structure exploration.

The patterns Interpreter [Gamma et al.94] and Translator could be regarded as object-oriented and functional incarnations respectively of the same idea: Abstract syntax is used to reify meaning. Pattern Translator adds a framework for incremental evaluation additionally and — by its non-intrusive way to add meaning to structures — is suitable for more interpretation applications, such as a persistence mechanism.

The Transfold pattern uses a very general scheme (folding) that allows expressing all the usual iteration operations (e.g., mapping, filtering, finding, etc.) in terms of it. Furthermore, it generalizes internal iteration to multiple data structures, thereby providing a remedy for the inflexibility previously associated with internal iteration.

Pattern Void Value widens the scope of using a default implementation [Smith95] to library design and to a safe mechanism of reference initialization.

In sum, the functional pattern system can help to understand existing object-oriented solutions in a broader sense and, hence, enable their generalization and also application to other areas. More fundamentally, it is hoped to make formerly "ingenious" designs understandable as relatively straightforward applications of well-known functional patterns. Patterns in general are, thus, a means to move design from an art towards engineering[15].

Object-oriented fertilization

Although techniques similar to function objects and lazy object have been described before, they gave a partial view only.

Function Object

For instance, descriptions of techniques similar to function objects concentrated on the locality of function definition [Breuel88], discussed a weakly typed way of achieving dynamic dispatch on functions [Coplien92], or used a non-uniform application syntax for algorithm parameterization [Hillegass93]. Apart from discussing the software engineering consequences of first class functions, pattern Function Object introduces innovations such as returning functions as a result, type safe currying of functions, keyword parameters, and generic function objects [Kühne97].

[15]There is no doubt, however, that engineering itself is not a mechanical activity and genius will still shine through exceptional masterpieces of engineering.

Lazy Object

Precursors of Lazy Object descriptions described a "trick" only to avoid unnecessary and repetitive evaluations [Smith95] or concentrated on the architectural aspects of pipes and filters [Buschmann et al.96]. John Hughes' investigation in the benefits of higher-order functions and lazy evaluation [Hughes87] is the only work I know of, which is similar in spirit to the research performed for this thesis. Of course, combined in a functional pattern system context, Function Object and Lazy Object go far beyond the issues discussed by John Hughes.

Transfold

Neither external iterators [Gamma et al.94], nor internal iterators operating on a single structure only [Goldberg & Robson83] provide a satisfactory general iteration framework. The availability of Function Object, however, makes internal iteration feasible even for object-oriented languages without support for closures. The availability of Lazy Object makes the separation of structure exploration and data consumption feasible. Combined with the idea of simultaneously processing multiple structures during one internal iteration, the result called Transfold provides the safety and economic notation of internal iteration while maintaining the flexibility and control of external iteration.

Translator

Without having been designed for this purpose, pattern Translator resolves some problematic issues associated with Visitor [Gamma et al.94]. Other proposals have been made to avoid interface pollution and a dependency cycle [Martin97], to allow partial visitations [Nordberg96], or to provide extensible visitors [Krishnamurthi et al.98]. But none of them introduces the elegant notion of an iteration with a generic function and coincidentally opens up the possibility of incremental evaluation with the concept of homomorphic interpretations [Kühne98].

In sum, patterns Transfold and Translator yield remarkable solutions for previously — with a pure object-oriented mind-set — unsuccessfully tackled problems.

Void Value

While there have been examples of default objects in the spirit of Void Value (e.g., NoController in VISUALWORKS), the idea to completely abandon nil and provide void values as a general principle is without precedent. The nullcases of UFO represent a corresponding idea in a functional programming framework. A comparison between the two diametrical approaches found Void Value to be superior [Kühne96b].

Value Object

Finally, pattern Value Object touches upon a much neglected topic in object-orientation: The taming of state. Software methodologies typically ignore this topic generously and focus on interfaces. Other approaches take reference semantics for granted and try to establish restrictions in order to prevent aliasing [Hogg91, Minsky96, Almeida97]. Value Object is a small, but maybe not that unimportant, reminder of the fact that reference semantics is a natural solution for efficient execution but not for intuitive programming.

In conclusion, functional patterns have been shown to be superior to a number of purely object-oriented solutions, e.g., using (multiple) inheritance and double-dispatch emulation.

The human side of patterns

One part of the pattern community believes that "*patterns are not so much about form and technology as they are about aesthetics and decency, about improving quality of life, about human dignity and decency*" [Coplien96a]. While the effect is indirect, I see a good chance that the presented pattern system will bring more comfort and quality to software users. Pattern Function Object promotes software with dynamically exchangeable strategies. This may induce a change in human computer interface design and the way software can be updated. A user of a word processor might be able to select between formating strategies and combine them with hyphenation algorithms. Software updates might consist of single components only that, e.g., add a formatting strategy to allow text to float around figures. This perspective provides an interesting alternative to the current buy-all-or-nothing choice, where fat software includes many unwanted features but almost certainly misses individually interesting ones. With function-object-like components, users would be empowered to configurate their own systems by buying components from the shelf.

The stream concept described in Lazy Object may help to work towards the same goal. Streams are the foundation of a pipes and filters architecture that allows configuration by the user. As a UNIX user creates new commands by combining existing commands with pipes, an application user may freely combine and arrange components that communicate with streams. This also would allow deferring decisions about application design to the user, creating more tailored and, thus, more useful and comfort providing software.

Transfold provides the most comfort to the programmer using it, but also users benefit from increased safety. However small the risk of introducing errors with external control structures is, it is there (see the Mariner incident in section 10.2.1 on page 163) and internal iteration minimizes that risk to absolutely zero.

The contribution of Value Object to end user comfort is quite indirect, but it may help to produce software containing less bugs (caused by aliasing) and allows programmers investing more time in improving software quality, because they are freed from tracking down tricky, state induced problems.

Pattern Translator offers users the benefit of incremental computations and, therefore, faster system response. An incremental approach may make the difference between annoying latencies and fluid interaction.

Last not least, Void Value can not prevent system malfunctions but allows handling such situations much more gracefully. Instead of frustrating users with unnerving program — or even operating system crashes — Void Value causes initialization induced system malfunctions to emerge as a non-performing system. Users are not delighted by not getting the answers they hoped for but at least they will save the trouble of losing data due to crashes or spending frustrating hours with reboots.

None of the patterns in the functional pattern system are a prerequisite to achieve user friendly software but their inherent software engineering qualities facilitates the creation of such software. Almost inevitably their contributions to flexibility and safety will shine through to improved human computer interactions.

Given a toolkit with such properties, a designer is less urged to emulate flexibility. For instance, user defined macros in a word processor are difficult to provide with an implementation language that does not facilitate dynamic extensions. As a result, inefficient interpretation techniques might be used. Hence, the functional pattern system may help to reduce the phenomenon of fat and slow application software resulting from inefficient emergency routes taken to account for an inflexible implementation language.

Pattern system

Within the functional pattern system, patterns reinforce each other due to their manifold interactions and collaborations. The analysis of pattern relationships revealed the nature of individual patterns and allowed them to be placed in a conceptual space. Pattern Translator was identified as a high-level, client oriented service. Transfold lends itself to be implemented as a library organization principle. Patterns Lazy Object and Value Object vividly claim language support due to their supporting potential, but are also very well able to benefit from other patterns.

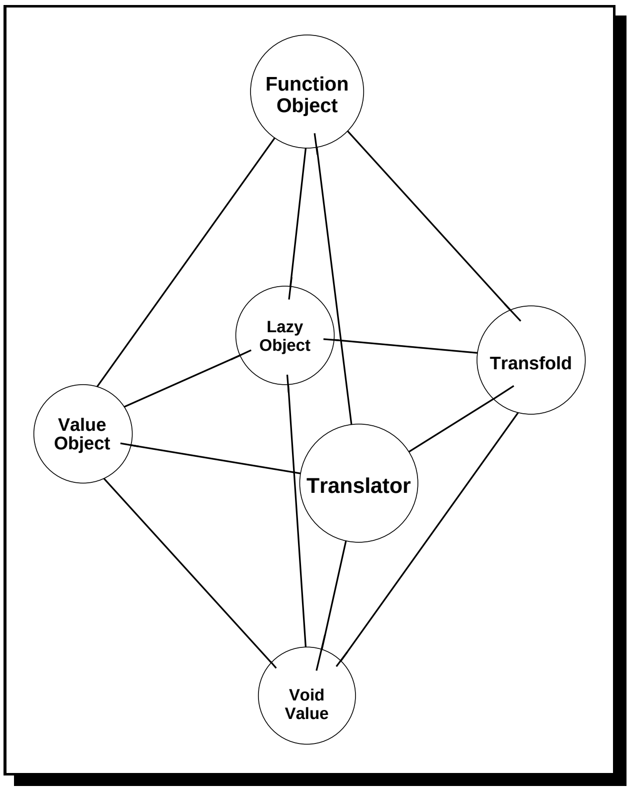


Figure E.1: Pattern Cathedral

The Void Value pattern is clearly the foundation stone for the functional pattern system (see figure E.1) and any language designed according to it. Complementary, Function Object is also fundamental in nature but shapes the pattern system or language from above, as a crucial framework element. As such, it can be regarded as the keystone[16] of the functional pattern system.

I do not claim completeness with the presented six patterns. Especially many functional programming practices wait to be documented. With regard to object-oriented software construction it seems worthwhile to further explore patterns helping to control state in the spirit of so-called *Islands* [Hogg91]. Nevertheless, the most crucial language concepts have been captured and described by the presented functional pattern system.

Drawbacks

The most obvious drawback with respect to the functional ideal is the overhead in description complexity. Some services, such as lazy streams, can be coded once and for all but, for instance, the tedious coding of dedicated classes for function currying must be repeated for every new function object definition. Although the benefits outweigh the required effort, the sometimes apparent need to code an emulation of fundamental mechanisms ultimately calls for language support.

Furthermore, object-oriented emulations of built-in functional concepts may incur efficiency penalties. While most systems will not be as time critical as to disallow functional patterns — and may in fact experience a speed up due to lazy evaluation, for instance — language support could significantly speed up execution and would allow compiler optimizations.

Tool support

Tools could aid in using functional patterns in conventional object-oriented languages from simple support such as automatically generating all necessary classes for a function object, till sophisticated type checking to discover possible failure to find an appropriate special function when a generic function is applied to a polymorphic variable.

The most intriguing impulse for tool support, however, is initiated by pattern Void Value. Research, comparing nullcases to Void Value [Kühne96b], led to the idea of moving responsibilities away from languages towards their associated environments.

Tool support as described in section 14.8 on page 252, frees a language from committing itself to either paradigm. The language just provides the notation for filling in the tiles of figure 4.2 on page 65 (the implementation corresponding to a function and an argument type) while a browser generates either a functional (rows), an object-oriented (columns), or a global (full table) view. This approach opens up the way to a (tile based) language supporting both procedural and type

[16]Or "boss", i.e., the last structural stone in the dome of a cathedral that imparts its grandeur the necessary stability.

abstraction extensions. The dichotomy between procedural and type abstraction is akin to the dual wave and particle interpretations of elementary particles in physics. Wave and particle interpretation are two incompatible views on the same entity. In a sense, hence, tiles are the elementary particles of an integrated functional and object-oriented programming language. The programmer, therefore, is an observer of a software "experiment". The programmer may choose the one interpretation that suits the currently required adjustments most. A programming environment, thus, is not anymore just a facilitator of editing, debugging, project management and class browsing, but actively provides optimal, and in particular, updateable views on a software system's organization (see figure 14.2 on page 258).

While object-oriented successors to LISP [Bobrow et al.86b] also use atomic pieces of constructor behavior (methods) to extend (generic) functions and objects, to my knowledge, the "tiles" approach represents the first attempt to retain an object-centered, encapsulation providing view [Chambers92b], while *coincidentally* allowing a functional perspective. Furthermore, it seems to be the first time that the organization principle or decomposition strategy — in short paradigm — of a language has been made dynamic by deferring it to its associated environment. Although much research into the tiles approach is still necessary, it seems fair to claim that William Cook's conjecture, about multi-dispatch languages being a possible escape from the procedural and type abstraction dichotomy [Cook90], has been validated.

Paradigm integration

When inquired for his opinion about object technology Timothy Budd felt positive about it but, referring to multi-paradigm programming, added:

> "*...the real revolution would arrive when we finally understood how to bring these diverse points of view together into one framework. [Budd95]*"
>
> – Timothy Budd

Without doubt the functional pattern system is a contribution to multi-paradigm software and language design. Its basic idea, to express functional concepts with patterns for object-oriented design with a view to holistic language design was acknowledged to be a contribution against a proliferation of concepts and terminology which would leave everyone with an incomplete picture [Reynolds96]. The functional pattern system advances the integration of the functional and object-oriented paradigms since

- an analysis on calculus level was performed to ensure the adequateness of embedding functions into objects,
- an investigation into the conflicts and cohabitance of the paradigms yielded a new integration approach and provided the stimulus for a selection of functional concepts to be expressed as patterns,

- functional and object-oriented solutions complemented each other leading to synergistic effects (see section 4.2.3 on page 59), and
- each functional pattern integrates smoothly into object-oriented designs.

As a result, I demonstrated the reduction of functional programming to object-oriented programming, at least in John Hughes' sense in that you can not add power to a language by removing concepts (e.g., state). All the virtues of functional programming enabled by adding powerful concepts like higher-order functions or lazy evaluation are feasible with an object-oriented language as well. However, reasoning about programs, e.g., as necessary for functional program transformations, is not possible for object-oriented software, unless one relies on design discipline (self imposed renunciation of state) for parts of the system. Yet, there is no way to define state away anyway. For instance, using monads in functional programming implies that the order of mapping a list suddenly does matter [Meijer & Jeuring95]. A monad used for stateful programming introduces order in calculations and it, in that sense, therefore, does not matter whether state is expressed in a language or supported by a language (see section 1.3.2.2 on page 21).

The object-oriented paradigm on its own was already shown to defeat many critical arguments made by John Backus [Backus78] towards imperative, so-called "von Neumann" languages [Kühne96a]. Supplemented with the presented functional pattern system the object-oriented paradigm truly liberates designers from the von Neumann style without any concession.

Patterns and Paradigms

Patterns and paradigms have an interesting relationship. Just like patterns depend on the level of abstraction, e.g., one could consider subroutines to be a pattern in assembler languages but not for higher level languages, they also depend on the context's paradigm. For instance, in a CLOS like setting based on generic functions, the Visitor pattern makes much less sense, since it mainly shows a technique to achieve double dispatch in single-dispatch languages. Also, most arguments of the Iterator pattern are obsolete for SMALLTALK with its rich iteration interface for collections. Therefore, the occurrence of certain patterns typically denotes weaknesses of their context paradigms, since they provide functionality which is not available at a primitive level. As has been carried out in this thesis, patterns may also be used to describe and characterize a paradigm by capturing its foundations in well documented mini-architectures. So, as a third relationship aspect between patterns and paradigms, patterns may be used to embed one paradigm into another. The

- signaling (what is wrong with a paradigm),
- descriptive (what are the constituents of a paradigm), and
- integrative (paradigm benefit transfer)

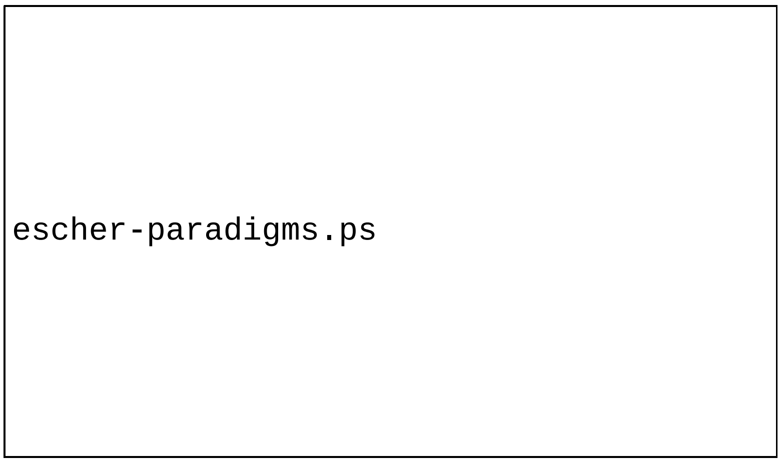


Figure E.2: Paradigm integration with patterns

aspects of patterns with regard to paradigms tie these two terms into a close relationship. I expect to see more evidence of this fruitful combination in the future since multi paradigm programming will become more and more important [Budd95] and patterns will surely continue their success. Patterns may be used to import paradigms, but ultimately will be used as lego pieces for language design.

Summarizing, this thesis created an original approach for paradigm integration. The mechanisms of object-orientation and functional programming have been combined and examined for their suitability to build reusable designs. Coincidentally, an imperative paradigm has been joined with a declarative paradigm (see figure 4.1 on page 56). A designer now has to choose between several competing mechanisms from two paradigms (e.g., inheritance and function objects). While this may appear to be an extra complication to a novice, it is an essential enrichment to the expert. Since the functional concepts are not only documented but formulated as patterns containing applicabilities and consequences, even novices will have guidance as to which tool-set to choose for a task.

Language design

As patterns are hoped to form the engineering handbooks of software development — and, therefore, contribute to make software engineering a science — language concept patterns, as presented here, may as well contribute to make language design a science. A dissection of programming languages into software engineering principles supporting concepts would allow the design of new languages from a known set of pieces. The functional pattern system captures the essence of software engineering properties supported by functional concepts. Along with an analysis of an effective paradigm integration approach (see chapter 4 on page 55) it enables a language designer to take a shopping list approach for the design of a new language. The documentation of the added benefit of a feature (e.g., for multi-dispatching methods see Translator on page on page 201) can be used for a solid justification for integrating it into a language. If the incorporation of features is based on single examples of their usefulness (e.g., support of binary methods through multi-dispatch), the decision may look arbitrary since there are often competing mechanisms that solve the same problem (e.g., a type system based on subsumption [Abadi & Cardelli95]).

Furthermore, an analysis of relationships between patterns in a system (see chapter 13 on page 221) gives rise to an overall language architecture. As indicated in section 14.8 on page 265, patterns Function Object and Void Value autonomously defined their position as two poles in a language design framework. Other patterns like Lazy Object and Value Object indicate a supporting as well as a service-using role and, thus, demand a language designer to facilitate all the rich possible interactions. It is also a welcome effect if a pattern, such as Transfold or Translator, recommends itself to be kept out of the design of a language. Hence, by analyzing pattern relationships, a designer gets hints which patterns are relevant to language design and what their role in a language framework might be.

Related work also recognized the relationship between patterns and languages. Gamma et al. acknowledge that patterns depend on the language context and, e.g., remark on Visitor that it is less needed in multi-dispatch languages [Gamma et al.94]. Peter Norvig investigates into the significance of well-known patterns in the context of dynamic languages [Norvig96]. Seiter et al. propose a new language construct (context relations) to facilitate the expression of a number of known patterns [Seiter et al.96]. Jan Bosch also introduces language support (a layered object model) to ease the implementation of design patterns and retain them as entities in the software [Bosch98]. Baumgartner et al. argue for an orthogonal combination of several well-known language concepts (interfaces, closures, and multi-dispatch) by demonstrating the reduction of effort needed to realize a number of design patterns [Baumgartner et al.96].

The latter work and mine are similar in that they regard particular patterns as symptoms of language deficiencies. Both approaches choose a specific set of patterns to induce a number of language constructs, whereas the above approaches aim at supporting patterns in general by one general language mechanism. My approach is different, though, in that Baumgartner et al. use typical patterns representing common coding practice, whereas my pattern base consists of language concepts from another paradigm. Hence, the discussions take place on different levels and the ultimate consequences of my paradigm integration for language design are much more radical (see section 14.8 on page 252).

Krishnamurthi et al. also aim at synthesizing object-oriented and functional design to allow extensions of both types and processors (functions) [Krishnamurthi et al.98]. They propose the Extensible Visitor pattern, a composite pattern [Riehle97], that uses a combination of Visitor and Factory Method [Gamma et al.94], to allow visitors to cope with extensions to the datatypes they operate on. The relation to language design is a proposed meta-linguistic abstraction to facilitate the implementation of Visitor and Extensible Visitor. However, Extensible Visitor does not overcome the problems of the original Visitor design, i.e., it also implies the pollution of the datatype interface, introduces a dependency cycle, and does not support partial visitations [Martin97]. Most severely, the requirement of a process[17] method in datatypes aggravates or even prohibits the definition of processes on already existing datatypes. In comparison to the tiles approach presented in section 14.8 on page 252, their resulting software structure leaves processor (function) definitions distributed over an inheritance hierarchy. Hence, in case of a required change to a processor, the programmer has to consult multiple classes since processor changes are likely to cross processor variant boundaries. The tile approach avoids this by the notion of multi-dispatching functions that do not rely on inheritance for extensions.

Apart from being a generative set of language design pieces, the functional pattern system was also used as a test case for an already designed language (see chapter 14 on page 233). The obstacles found while trying to implement functional patterns in EIFFEL revealed shortcomings of the language that also will emerge in other attempts to design reusable software. Since the functional pattern system supports flexible designs, it evaluates an implementation language for its ability to support these. Any difficulties encountered with the implementation of the functional pattern system are valuable hints on improving the implementation language.

Previous integration attempts

The functional and object-oriented paradigms have been the target for many integration attempts. However, often only a partial attempt (e.g., just providing functions) has been made [van Rossum91, Klagges93, Dami94, Watt et al.94], or state is completely rejected [Braine94, Rémy & Vouillon97].

Object-oriented extensions to functional languages typical concentrate on typing issues and do not provide dynamic binding [Rémy & Vouillon97]. One dialect of HASKELL uses existential types to enable dynamic binding [Läufer96] but state has to be modeled in programs, i.e., is not supported as a primitive.

CLOS [Bobrow et al.86b], OAKLISP [Lang & Pearlmutter86] and DYLAN [Shalit et al.92] can be regarded as object-oriented extensions to LISP [Winston & Horn84] but they have no strong notion of encapsulation. Classes are modeled with records with no access restrictions. Their generic functions are a very powerful approach but when viewed from a paradigm integration perspective the languages appear unpolished. The frequent use of macros and the meta-level facilities make them suitable for explorative programming but questionable for being software engineering languages[18].

Two languages, UFO [Sargeant93] and LEDA [Budd95], attempt a full integration including state. A comparison between the *nullcases* of UFO and Void Value has been given in [Kühne96b] (see also section 14.7 on page 248). Timothy Budd

[17]The analog method used for Visitor is called Accept.

[18]Indeed, the perceived lack of determinism in CLOS led to the development of SATHER.

shows many examples when a particular paradigm is useful but does not extract applicabilities nor consequences. So, even with a given multi-paradigm language like LEDA the functional pattern system is useful for guidance as to when to use which technique.

A functional pattern system is a promising approach to familiarize object-oriented designers with functional concepts. Another attempt is to take a popular object-oriented language and conservatively extend it with functional features. The PIZZA language adds parametric polymorphism, first-class functions, and algebraic types to JAVA [Odersky & Wadler97]. Any JAVA program is also a correct PIZZA program. Parametric polymorphism is a standard feature in modern languages and should have been available in JAVA already. One nice point about the PIZZA approach though, is that it allows polymorphic methods, i.e., universally quantified variable types for methods that, hence, do not demand their class to feature extra type parameters. First-class functions in PIZZA implement function objects (without keyword parameters and multi-dispatch) and can even be specified anonymously, but fail to support partial parameterization directly. The discussion in chapter 7 on page 93, however, made it very clear that partial parameterization is one of the most beneficial aspects of function objects. While support for algebraic data types in PIZZA introduces a nice, short syntax for constructors (subclasses implementing constructors of a sum type), I prefer Void Value or dynamically extensible multi-dispatch functions over pattern matching. The PIZZA switch statements over type constructors are sensitive to the addition of new constructors. Void Value avoids this issue by promoting procedural abstraction [Kühne96b] (see also section 4.2.1.2 on page 58). Multi-dispatching functions avoid the same problem by allowing the extension of the specialized function set without requiring changes to old code. Conclusively, PIZZA is a vast improvement over JAVA but is not entirely satisfactory with regard to a paradigm integrating language. The design of JAVA and its virtual machine forced the designers of PIZZA several times to compromises [Odersky & Wadler97], and left a number of rough edges. Most prominently, built-in JAVA types do not mix well with user defined classes, for example when trying to define a general container class. Ironically, JAVA's solution to polymorphic arrays, prohibits to adopt the best model of arrays in PIZZA [Odersky & Wadler97].

Although BETA [Madsen et al.93] and TRANSFRAME [Shang95b] have not been designed to integrate paradigms, their unification of classes and functions are the closest approach to a holistic paradigm integrating language, as suggested by the functional pattern system. However, the discussion about tool support in section 14.8 on page 252 seems to indicate that a reductionistic approach appears to be more promising. With a holistic concept like BETA's *pattern*, that can be used as an object or as a function, one, nevertheless, has to decide for either view and is left with the benefits and drawbacks of this decision. Hence, the approach to have a language that allows defining tiles that are dynamically arranged to represent a function or an object by a tool, seems to be the better approach. Also, none of the above mentioned languages attempts to integrate state with lazy evaluation. Interesting research to make effects lazy has been carried out for functional languages [Launchbury93, Launchbury & Jones94], but a language supporting stateful objects with dynamic binding and lazy evaluation remains to be designed. The investigation into paradigm conflicts and possible cohabitance, the impulses from the functional pattern system, and the proposed directions in language design, are hoped to be supporting contributions towards this goal.

"If I have not seen as far as others, then that's because giants were standing on my shoulders." – Hal Abelson

Bibliography

[Abadi & Cardelli94] Martín Abadi and Luca Cardelli. A theory of primitive objects: untyped and first order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320, Japan, 1994.

[Abadi & Cardelli95] Martín Abadi and Luca Cardelli. On subtyping and matching. In W. Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, pages 145–167, Aarhus, Denmark, Springer-Verlag, August 1995.

[Abadi94] Martín Abadi. Baby modula 3 and a theory of objects. *Journal of Functional Programming*, 4:249–283, April 1994.

[Abelson & Sussman87] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, London, 6th edition, 1987.

[Agrawal et al.91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *Proceedings OOPSLA '91*, pages 113–128, November 1991.

[Aho & Ullmann92] Alfred V. Aho and Jeffrey D. Ullmann. *Foundations of Computer Science*. Computer Science Press, New York, 1992.

[Aho et al.86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, March 1986.

[Alexander et al.77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[Alexander64] Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[Alexander79] Christopher Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.

[Allison92]	L. Allison. Lazy dynamic-programming can be eager. *Information Processing Letters*, 43(4):201–212, 1992.
[Almeida97]	Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of ECOOP '97*, 1997.
[America & v. Linden90]	Pierre America and Frank v. Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 161–168, October 1990.
[Appel93]	Andrew W. Appel. A critique of standard ML. *Journal of Functional Programming*, 4(3):391–429, October 1993.
[Auer94]	Ken Auer. Reusability through self-encapsulation. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 505–516. Addison-Wesley, 1994.
[Backus et al.63]	J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 1(6):1–17, January 1963.
[Backus78]	John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
[Baker93]	Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.
[Ballard95]	Fred Ballard. Who discovered Alexander? In *Patterns mailing list*. patterns-discussion@cs.uiuc.edu, October 1995.
[Barendregt84]	Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishing Company, 2^{nd} edition, 1984.

[Baumgartner et al.96]	Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, February 1996.
[Beaudouin-Lafon94]	Micheal Beaudouin-Lafon. *Object-oriented Languages*. Chapman and Hall, 1994.
[Beck & Johnson94]	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.
[Beck96]	Kent Beck. Method object. In *Patterns mailing list Digest*, Patterns Digest. patterns-discussion@cs.uiuc.edu, April 1996.
[Berger91]	Emery D. Berger. FP+OOP=Haskell. Technical Report TR-92-30, The University of Texas at Austin, Department of Computer Science, December 1991.
[Biddle & Tempero96]	Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. Technical Report CS-TR-95/17, Victoria University of Wellington, Department of Computer Science, January 1996.
[Bird & de Moor92]	R. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction*, LNCS 669, pages 45–66. Springer Verlag, June 1992.
[Bird & Wadler88]	Richard Bird and Philip Wadler. *Introduction to Functional Programming*. C.A.R. Hoare Series. Prentice Hall International, 1988.
[Bird84]	Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica 21*, pages 239–250, 1984.
[Bird86]	Richard S. Bird. An Introduction to the Theory of Lists. Technical Report PRG-56, Oxford University, October 1986.
[Bobrow et al.86a]	Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented

programming. In *Proceedings OOPSLA '86*, pages 17–29, November 1986.

[Bobrow et al.86b] Daniel G. Bobrow, Ken Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 17–29, November 1986.

[Boiten et al.93] Eerke A. Boiten, A. Max Geerling, and Helmut A. Partsch. Transformational derivation of (parallel) programs using skeletons. Technical Report 93-20, Katholieke Universiteit Nijmegen, September 1993.

[Booch94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin / Cummings Publishing Company, 2^{nd} edition, 1994.

[Borland94] Borland. *Borland C/C^{++} 4.0 Reference Manual*. Borland, Inc., 1994.

[Bosch98] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2), May 1998.

[Botorog96] Georg Botorog. Strukturierte parallele Programmierung zur Lösung numerischer Probleme. In J. Ebert, editor, *Workshop: Alternative Konzepte für Sprachen und Rechner - Bad Honnef 1995*. Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, May 1996.

[Braine94] Lee Braine. Integrating functional and object-oriented paradigms. Technical report, University College London, Gower Street, London WC1E6BT, UK, December 1994.

[Braine95] Lee Braine. The importance of being lazy, June 1995.

[Breuel88] Thomas M. Breuel. Lexical closures for C++. In *C++ Conference Proceedings*, pages 293–304, October 1988.

[Bruce94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4:127–206, April 1994.

[Buckley95]	Bob Buckley. Common subexpression in parameter patterns. 5380 of comp.lang.functional, August 1995.
[Budd95]	Timothy Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.
[Burstall & Darlington77]	R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), 1977.
[Burstall et al.80]	Ron M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings, 1980 LISP Conference*, pages 136–143, August 1980.
[Burton & Cameron94]	F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programing*, 3(2):171–190, 1994.
[Buschmann et al.96]	Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, July 1996.
[Byte94]	Component software. Byte Magazin, May 1994.
[Cann92]	David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.
[Cardelli & Wegner85]	Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
[Cardelli84]	Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67. Springer Verlag, 1984.
[Carlsson & Hallgren93]	M. Carlsson and T. Hallgren. Chalmers University, May 1993.
[Carré & Geib90]	Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *Proceedings OOPSLA/ECOOP '90*, pages 312–321, October 1990.
[Centaur92]	Centaur documentation, INRIA, 1992.

[Chambers & Leavens95] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *TOPLAS*, 17(6):805–843, November 1995.

[Chambers92a] Craig Chambers. *The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, Stanford University, March 1992.

[Chambers92b] Craig Chambers. Object-oriented multi-methods in cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, Springer-Verlag, June 1992.

[Chiba & Masuda93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In Oscar M. Nierstrasz, editor, *ECOOP '93*, Lecture Notes in Computer Science 707, pages 482–501, July 1993.

[Clack & Myers95] C. Clack and C. Myers. The dys-functional student. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022*, pages 289–309, Nijmegen, The Netherlands, Springer-Verlag, Dec 1995.

[Cleeland et al.96] Chris Cleeland, Douglas C. Schmidt, and Timothy H. Harrison. External polymorphism — An object structural pattern for transparently extending C++ concrete data types. In *Preliminary Proceedings of PLoP '96*, 1996.

[Clinger & Hansen92] W. Clinger and L. T. Hansen. Is explicit deallocation really faster than garbage collection? 1992.

[Cole88] M. I. Cole. A 'skeletal' approach to the exploitation of parallel computation. In *CONPAR 88*, pages 100–107. British Computer Society, London, UK, 1988.

[Cole89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, London, UK, 1989.

[Coleman et al.94] Derek Coleman et al. *Object-oriented development: The Fusion Method*. Object-oriented series. Prentice Hall, 1994.

[Convex93] Convex AVS developer's guide. Convex Computer Corporation, first edition, Richardson, Texas, December 1993.

[Cook89a] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, May 1989.

[Cook89b] W. R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, August 1989.

[Cook90] William R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 151–178. Springer, May 1990.

[Coplien92] James O. Coplien. *Advanced C^{++}: Programming Styles and Idioms*. Addison-Wesley, 1992.

[Coplien95] James O. Coplien. Multi-paradigm design and implementation. Tutorial Notes from the Summer School on Object Orientation in Tampere, Finland, AT&T Bell Laboratories, Naperville, Illinois, USA, August 1995.

[Coplien96a] James O. Coplien. The column without a name: The human side of patterns. *C++ Report*, 8(1):81–85, January 1996.

[Coplien96b] James O. Coplien. Pattern definition. In *Patterns mailing list*. patterns-discussion@cs.uiuc.edu, December 1996.

[Cunningham94] Ward Cunningham. The checks pattern language of information integrity. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 145–156. Addison-Wesley, 1994.

[Dahl & Nygaard81] Ole-Johan Dahl and Kristen Nygaard. The development of the SIMULA language. In Richard L. Wexelblat, editor, *ACM Conference on the History of Programming Languages*, pages 439–493. Academic Press, 1981.

[Dami94] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, University of Geneva, April 1994.

[Darlington et al.91] J. Darlington, A. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. Kelly, K. Sephton, and D. Sharp. Structured parallel functional programming. In H. Glaser and P. H. Hartel, editors, *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 31–52, Southampton, UK, Department of Electronics and Computer Science, University of Southampton, 1991.

[Darlington et al.95] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 19–28, Santa Barbara, California, July 1995.

[Davies & Pfenning96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Proceedings of* 23*rd* *Annual ACM Symposium on Principles of Programming Languages*, January 1996.

[Deegener et al.94] M. Deegener, G. Große, B. Kühnapfel, and H. Wirth. *MuSE* — Multimediale Systementwicklung. In I. Toch, G. Kampe, H. Ecker, and F. Breitenecker, editors, *ASIM-Arbeitskreistreffen Simulation Technischer Systeme und Simulationsmethoden und Sprachen für parallele Prozesse (Wien, 31.1.-1.2. 1994)*. ASIM, 1994.

[DeMichiel & Gabriel87] 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.

[Despeyroux88] Thierry Despeyroux. Typol: a formalism to implement natural semantics. Technical report, INRIA research report 94, 1988.

[Dijkstra76] Edsgar W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976.

[Dosch95] Walter Dosch. Personal communication. Workshop Alternative Konzepte für Sprachen und Rechner in Bad Honnef, May 1995.

[Duke95] Roger Duke. Do formal object-oriented methods have a future? In Christine Mingins and Bertrand

Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 273–280, Prentice Hall International, London, 1995.

[Dyson & Anderson96] Paul Dyson and Bruce Anderson. State patterns. In *The* 1^{st} *Annual European Conference on Pattern Languages of Programming, EuroPLoP '96*, Kloster Irsee, Germany, 1996.

[Ebert87] Jürgen Ebert. Funktionale Programmiersprachen: Ein Überblick. Manuskript, 1987.

[Eckert & Kempe94] Gabriel Eckert and Magnus Kempe. Modeling with objects and values: Issues and perspectives, August 1994.

[Edelson92] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. Technical Report UCSC-CRL-92-27, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, CA 95064, USA, June 1992.

[Edwards95] Stephen Edwards. Streams: A pattern for "pull-driven" processing. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 417–426. Addison-Wesley, 1995.

[Ellis & Stroustrup90] M. Ellis and B. Stroustrup. *The Annotated* C^{++} *Reference Manual*. Addison-Wesley, 1990.

[Feiler & Tichy97] Peter H. Feiler and Walter F. Tichy. Propagator — a family of patterns. In *The* 23^{rd} *TOOLS conference USA '97*, St. Barbara, California, July 1997.

[Felleisen91] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991.

[Field & Harrison88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, Reading, MA, 1988.

[Fisher & Mitchel95] Kathleen Fisher and John C. Mitchel. The development of type systems for object-oriented languages. November 1995.

[Floyd64] R. W. Floyd. Treesort (algorithm 113). *Communications of ACM*, December 1964.

[Floyd87] R. W. Floyd. The paradigms of programming. *ACM Turing Award Lectures: The First Twenty Years*, 1987.

[Freddo96] Freddo. The carving of dragons. In *Patterns mailing list*. patterns-discussion@cs.uiuc.edu, June 1996.

[Friedman & Wise76] D. Friedman and D. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edingburgh University Press, 1976.

[Gabriel96] Richard Gabriel. Pattern definition. In *Patterns mailing list*. patterns-discussion@cs.uiuc.edu, December 1996.

[Gamma et al.93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 406–431, Kaiserslautern, Germany, Springer-Verlag, July 1993.

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

[Gamma91] Erich Gamma. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools (in German)*. PhD thesis, University of Zurich, 1991.

[Gamma95] Erich Gamma. Singular object. In *Patterns mailing list Digest*, Patterns Digest. patterns-discussion@cs.uiuc.edu, March 1995.

[Gamma97] Erich Gamma. The facet pattern. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, Addison-Wesley, 1997.

[Gibbons & Jones93] Jeremy Gibbons and Geraint Jones. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 71, University of Auckland, Department of Computer Science, 1993.

[Gill et al.93] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional*

Programming Languages and Computer Architecture, Copenhagen '93, April 1993.

[Goldberg & Robson83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gordon93a] Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge Computer Laboratory, February 1993.

[Gordon93b] Andrew D. Gordon. An operational semantics for I/O in a lazy functional language. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen*. ACM, 1993.

[Gostanza et al.96] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. *Conference on Functional Programming*, 31(6):110–121, June 1996.

[Grant-Duff94] Zully Grant-Duff. Skeletons, list homomorphisms and parallel program transformation. Technical Report Internal Report 94/14, Department of Computing, Imperial College, July 94.

[Griss95] Martin Griss. Systematic software reuse – objects and frameworks are not enough (panel). *ACM SIGPLAN Notices*, 30(10):281–282, October 1995.

[Grosch & Snelting93] Franz-Josef Grosch and Gregor Snelting. Polymorphic components for monomorphic languages. In *Proceedings of the Second International Workshop on Software Reusability*, pages 47–55. IEEE Comp. Soc. Press, March 1993.

[Hankin et al.97] Chris Hankin, Hanne Riis Nielson, and Jens Palsberg. Position statements on strategic directions for research on programming languages. *ACM SIGPLAN NOTICES*, 32(1):59–65, January 1997.

[Hankin94] Chris Hankin. *Lambda Calculi, A Guide for Computer Scientists*. Graduate Texts in Computer Science. Clarendon Press, Oxford, 1994.

[Harkavy & et al.94] Michael Harkavy and et al., editors. *Webster's new encyclopedic dictionary*. Black Dog & Leventhal publishers Inc., 151 West 19th Street, New York 10011, 1994.

[Harms & Zabinski77] Edward Harms and Michael P. Zabinski. *Introduction to APL and computer programming*. Wiley, 1977.

[Harris97] Dave Harris. Aliasing in Smalltalk. In *Java Wiki Web*. Hillside group, 1997.

[Harrison & Ossher93] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, October 1993.

[Harrison et al.94] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Comparing programming paradigms: An evaluation of functional and object-oriented progams. Technical Report SO171BJ, University of Southhampton, Dept. of Electronincs and Computer Science, UK, August 1994.

[Hebel & Johnson90] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk-80. *Journal of Object-Oriented Programming*, 2(6):40–44, March 1990.

[Henderson & Constantin91] Brian Henderson and L. L. Constantin. Object-oriented development and functional decomposition. *Journal of Object-Oriented Programming*, 3(5):11–17, January 1991.

[Henhapl et al.91] W. Henhapl, W. Bibel, J.L. Encarnação, S. Huss, and E. Neuhold. *MUSE – Multimediale Systementwicklung (DFG-Forschergruppenantrag)*. Technische Hochschule Darmstadt, February 1991.

[Hewitt77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hillegass93] Aaron Hillegass. The design of the Eiffel booch components. *Eiffel Outlook*, 3(3):20–21, December 1993.

[Hirschfeld96] Robert Hirschfeld. Convenience methods. In *The* 1st *Annual European Conference on Pattern Languages of Programming, EuroPLoP '96*, Kloster Irsee, Germany, July 1996.

[Hogg et al.92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *ACM OOPS Messenger*, 2(3):11–16, April 1992.

[Hogg91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, pages 271–285, November 1991.

[Hudak & Bloss85] Paul Hudak and Adriene Bloss. The aggregate update problem in functional programming languages. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985.

[Hudak & Fasel92] Paul Hudak and Joseph Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):Section T, May 1992.

[Hudak & Sundaresh88] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional i/o systems. Technical Report YALEU/DCS/RR-665, Yale University Department of Computer Science, December 1988.

[Hudak89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):361–411, September 1989.

[Hudak92] Paul Hudak. Mutable abstract datatypes. Research Report YALEU/DCS/RR-914, Yale University Department of Computer Science, December 1992.

[Hudak96] Paul Hudak. Building domain-specific embedded languages. *Computing Surveys*, 28(4), December 1996.

[Hughes85] John Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, LNCS 201, pages 129–146. Springer, 1985.

[Hughes87] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, August 1987.

[Hunt & Szymanski77] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[Hutton92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–429, July 1992.

[Ingalls86] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings OOPSLA '86*, pages 347–349, November 1986.

[Jacobs & Rutten97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. In *EATCS Bulletin*, 1997.

[Jacobs98] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In *Electronic Notes in Computer Science 11, Special issue on the workshop Coalgebraic Methods in Computer Science (CMCS 1998)*, 1998.

[Jacobson et al.94] Ivar Jacobson et al. *Object-Oriented Software Engineering: A use case driven approach.* Addison Wesley, 4^{th} edition, 1994.

[Jäger et al.88] M. Jäger, M. Gloger, and S. Kaes. Sampλe — a functional language. LNCS 328, 1988.

[Jager et al.91] P. Jager, W. Jonker, A. Wammes, and J. Wester. On the generation of interactive programming environments. *ESPRIT project 2177: GIPE II*, 1991.

[Jazayeri95] Mehdi Jazayeri. Component programming – a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference, Sitges, Spain*, September 1995.

[Jeschke95] Eric Jeschke. Speed of fp languages, laziness, ”delays”. News article 5745 of comp.lang.functional, November 1995.

[Jézéquel96] J.-M. Jézéquel. *Object Oriented Software Engineering with Eiffel.* Addison-Wesley, March 1996.

[Johnson & Foote88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Johnson92] Ralph E. Johnson. Documenting frameworks using patterns. *ACM SIGPLAN Notices, OOPSLA '92*, 27(10):63–76, October 1992.

[Johnson94] Ralph E. Johnson. How to develop frameworks. In *ECOOP '94 Tutorial Documentation*, July 1994.

[Jones & Wadler93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. *Proceedings of* 20th *Annual ACM Symposium on Principles of Programming Languages*, 4:71–84, January 1993.

[Jones96] Capers Jones. Programming languages table. *Software Productivity Research, Release 8.2, http://www.spr.com/library/langtbl.htm*, March 1996.

[Kashiwagi & Wise91] Yugo Kashiwagi and David S. Wise. Graph algorithms in a lazy functional programming language. In 4th *Int. Symp. on Lucid and Intensional Programming*, pages 35–46, 1991.

[Kay96] Alan Kay. The early history of Smalltalk. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages 2*, pages 511–578. Addison-Wesley, 1996.

[Keller & Sleep86] Robert M. Keller and M. Ronan Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, January 1986.

[Kent & Howse96] Stuart Kent and John Howse. Value types in Eiffel. In *The* 19th *TOOLS conference Europe 96*, Paris, 1996.

[Kent78] W. Kent. Data and reality: Basic assumptions in data processing reconsidered. North-Holland Publishing Company, 1978.

[Kershenbaum et al.88] Aaron Kershenbaum, David Musser, and Alexander Stepanov. Higher-order imperative programming. Technical Report 88–10, Rensselaer Polytechnic Institute Computer Science Department, April 1988.

[Khoshafian & Copeland86] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proceedings OOPSLA '86*, pages 406–416, November 1986.

[Kieburtz & Lewis95] Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 267–307. Springer, 1995.

[King & Launchbury93] David J. King and John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. In John T. O' Donnell and Kevin Hammond, editors, *Glasgow functional programming workshop*, pages 145–155, Ayr, Scotland, Springer-Verlag, 1993.

[Klagges93] 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.

[Knuth74] Donald E. Knuth. Structured programming with go to statments. *Computing Survyes*, 6(4):261–301, December 1974.

[Kofler93] Thomas Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.

[Kozato & Otto93] Y. Kozato and G. Otto. Benchmarking real-life image processing programs in lazy functional languages. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen*. ACM, 1993.

[Krishnamurthi et al.98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, to be published 1998.

[Kuhn70] Thomas Kuhn. *The Structure of Scientific Revolutions*. University of Chicago, Chicago, 2*nd* edition, 1970.

[Kühnapfel & Große94] B. Kühnapfel and G. Große. Specification of Technical Systems by a Combination of Logical and Functional Languages. In Hendrik C. R. Lock, editor, 3*rd* *Workshop on Functional Logic Programming (Schwarzenberg, 10.-14.1. 1994)*, 1994.

[Kühnapfel93] B. Kühnapfel. *Kombinatorsysteme zur Beschreibung paralleler Prozesse in funktionalen Sprachen*. Diplomarbeit am Fachbereich Informatik. Technische Hochschule Darmstadt, March 1993.

[Kühne94] Thomas Kühne. Higher order objects in pure object-oriented languages. *ACM SIGPLAN Notices*, 29(7):15–20, July 1994.

[Kühne95a] Thomas Kühne. Higher order objects in pure object-oriented languages. *ACM OOPS Messenger*, 6(1):1–6, January 1995.

[Kühne95b] Thomas Kühne. Parameterization versus inheritance. In Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Prentice Hall International, London, 1995.

[Kühne96a] Thomas Kühne. Can object-orientation liberate programming from the von Neumann style? In J. Ebert, editor, *Workshop: Alternative Konzepte für Sprachen und Rechner - Bad Honnef 1995*. Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, May 1996.

[Kühne96b] Thomas Kühne. Nil and none considered null and void. In Alexander V. Smolyaninov and Alexei S. Shestialtynov, editors, *Conference Proceedings of WOON'96 in Saint Petersburg*, pages 143–154, June 1996.

[Kühne96c] Thomas Kühne. Recipes to reuse. In *The* 1*st* *Annual European Conference on Pattern Languages of Programming, EuroPLoP '96*, Kloster Irsee, Germany, July 1996.

[Kühne97] Thomas Kühne. The function object pattern. *C++ Report*, 9(9):32–42, October 1997.

[Kühne98] Thomas Kühne. The translator pattern — external functionality with homomorphic mappings. In Raimund Ege, Madhu Sing, and Bertrand Meyer, editors, *The* 23*rd* *TOOLS conference USA '97*, pages 48–62. IEEE Computer Society, July 1998.

[LaLonde & Pugh91] Wilf LaLonde and John Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[LaLonde94] Wilf LaLonde. *Discovering Smalltalk*. Benjamin / Cummings Publishing Company, 1994.

[Landin65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8(2):89–101, February 1965.

[Lang & Pearlmutter86] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented scheme with first class types. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 30–37, November 1986.

[Läufer95] Konstantin Läufer. A framework for higher-order functions in C++. In *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, USENIX, June 1995.

[Läufer96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[Launchbury & Jones94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. *SIGPLAN Notices*, 29(6):24–35, June 1994.

[Launchbury & Jones95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[Launchbury93] John Launchbury. Lazy imperative programming. In *ACM SIGPLAN Workshop on State in Prog. Langs.* University of Copenhagen, June 1993.

[Leavens94] Gary T. Leavens. Fields in physics are like curried functions or Physics for functional programmers. Technical Report TR #94-06b, Department of Computer Science, Iowa State University, 229 Atanasoff Hall, May 1994.

[Limberghen & Mens94] Marc Van Limberghen and Tom Mens. 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.

[Liskov & Guttag86] Barbara Liskov and John Guttag. *Abstraction and Specification in Programm Development*. MIT Press, 1986.

[Liskov & Wing93] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 118–141, Kaiserslautern, Germany, Springer-Verlag, July 1993.

[MacLennan82] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices*, 17(12):70–79, December 1982.

[Madsen et al.93] Ole L. Madsen, Kristen Nygaard, and Birger Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley and ACM Press, 1993.

[Manolescu97] Dragos-Anton Manolescu. A data flow pattern language. In *The 4th Annual Conference on Pattern Languages of Programming, Monticello, Illinois*, 1997.

[Martin94] Robert Martin. Discovering patterns in existing applications. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 365–393. Addison-Wesley, 1994.

[Martin97] Robert C. Martin. Acyclic visitor. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, Addison-Wesley, 1997.

[Maughan96] Glenn Maughan. Restricting access to data structures? comp.lang.eiffel, February 1996.

[Meijer & Jeuring95] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 228–266. Springer, 1995.

[Meijer et al.91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Coomputer Architecture, Cambridge, Massachusetts*, LNCS 523, pages 124–144. Springer Verlag, August 1991.

[Mens et al.94] Tom Mens, Kim Mens, and Patrick Steyaert. Opus: a formal approach to object-orientation. In Tim Denvir Maurice Naftalin and Miquel Bertran, editors, *FME' 94 Proceedings: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Mens et al.95] Tom Mens, Kim Mens, and Patrick Steyaert. Opus: A calculus for modelling object-oriented concepts. In D. Patel, Y. Sun, and S. Patel, editors, *OOIS' 94 Proceedings: International Conference on Object Oriented Information Systems*. Springer-Verlag, 1995.

[Menzies & Haynes95] Tim Menzies and Philip Haynes. The methodology of methodologies, or , evaluating current methodologies: Why and how. In Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 83–92, Prentice Hall International, London, 1995.

[Meunier95a] Jeffrey A. Meunier. Functional programming. Technical report, University of Connecticut, December 1995.

[Meunier95b] Regine Meunier. The pipes and filters architecture. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 427–440. Addison-Wesley, 1995.

[Meyer88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Meyer92] Bertrand Meyer. *EIFFEL the language*. Prentice Hall, Object-Oriented Series, 1992.

[Meyer93] Bertrand Meyer. Toward an object-oriented curriculum. JOOP: Education & Training, May 1993.

[Meyer94a] Bertrand Meyer. Beyond design by contract. Invited presentation at TOOLS 15, December 1994.

[Meyer94b] Bertrand Meyer. *Reusable Software*. Prentice Hall, 1994.

[Meyer95] Bertrand Meyer. The future of Eiffel's type system. Personal communication, March 1995.

[Meyer96] Bertrand Meyer. Static typing and other mysteries of life. *Object Currents*, 1(1), January 1996.

[Meyer97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ 07458, 2nd edition, 1997.

[Meyrowitz87] Norman Meyrowitz, editor. *Proceedings OOPSLA '87*, Orlando, Florida, December 1987.

[Milner et al.90] Robin Milner, M. Tofte, and R. Harper. *The definition of standard ML*. MIT Press, Cambridge, 1990.

[Minsky95] Naftaly H. Minsky. Unshareable dynamic objects – how to resolve a conflict between encapsulation and pointers. *Eiffel Outlook*, 4(5):4–12, June 1995.

[Minsky96] Naftaly H. Minsky. Towards alias-free pointers. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 189–209. Springer-Verlag, June 1996.

[Mogensen95] Torben Ægidius Mogensen. How useful is laziness? News article of comp.lang.functional, August 1995.

[Moynihan94] Tony Moynihan. Objects versus functions in user-validation of requirements: Which paradigm works best? In D. Patel, Y. Sun, and S. Patel, editors, *OOIS '94 Proceedings, 1994 International Conference on Object Oriented Information Systems*, pages 54–73, Dublin City University, Ireland, Springer-Verlag, London, 1995, ISBN 3-540-19927-6, 1994.

[Mugridge et al.91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 307–324, Geneva, Switzerland, Springer-Verlag, July 1991.

[Murer et al.93a] Stephan Murer, Stephen Omohundro, and Clemens Szypersky. Engineering a programming language: The type and class system of Sather. In Joerg Gutknecht, editor, *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993.

[Murer et al.93b] Stephan Murer, Stephen Omohundro, and Clemens Szypersky. Sather Iters: Object-oriented iteration abstraction. Technical Report TR-93-045, ICSI, Berkeley, August 1993.

[Naur69] Peter Naur, editor. *Meeting notes of the NATO Conference in Garmisch.* NATO Science Community, 1969.

[Nelson91] Michael L. Nelson. An object-oriented tower of babel. *ACM OOPS Messenger*, 2(3):3–11, July 1991.

[Neumann86] P.G. Neumann. Risks to the public in computer systems. *ACM Software Engineering Notes 11,* pages 3–28, 1986.

[Nielson & Nielson93] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1993.

[Nierstrasz & Meijler95] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[Nierstrasz89] Oscar Nierstrasz. A survey of object-oriented concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison-Wesley, 1989.

[Nordberg96] Martin E. Nordberg. Variations on the visitor pattern. In *Preliminary Proceedings of PLoP '96*, 1996.

[Norvig96] Peter Norvig. Design patterns in dynamic programming. Presentation at Object World '96, May 1996.

[ObjectExpert97] Subjectivity — a new way to solve your biggest programming challenges. Object Expert issue on Subjectivity, SIGS Publications, March 1997.

[Odersky & Wadler97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Odersky91] Martin Odersky. How to make destructive updates less destructive. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 25–26. ACM Press, January 1991.

[Okasaki95a] Chris Okasaki. Lazy functional sorting. Article 5349 of comp.lang.functional, September 1995.

[Okasaki95b]	Chris Okasaki. Re: Common subexpression in parameter patterns. Article 5385 of comp.lang.functional, August 1995.
[Okasaki95c]	Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.
[Omohundro & Lim92]	Stephen Omohundro and Chu-Cheow Lim. The sather language and libraries. Technical Report TR-92-017, ICSI, Berkeley, March 1992.
[Omohundro94]	Stephen M. Omohundro. The Sather 1.0 specification. Technical report, International Computer Science Institute, Berkeley, December 1994.
[Pardo98a]	Alberto Pardo. *A Calculational Approach to Monadic and Comonadic Programs*. PhD thesis, Darmstadt University of Technology, 1998.
[Pardo98b]	Alberto Pardo. Personal communication, January 1998.
[Parnas72]	D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.
[Pepper92]	Peter Pepper. *Grundlagen der Informatik*. Oldenbourg, 1992.
[Petre89]	Marian Petre. *Finding a Basis for Matching Programming Languages to Programming Tasks*. PhD thesis, Department of Computer Science, University College London, 1989.
[Pierce & Turner94]	Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4:207–247, April 1994.
[Ponder88]	C. G. Ponder. Are applicative languages inefficient? *SIGPLAN Notices*, 23(6):135–139, June 1988.
[Poole et al.98]	Ian Poole, Craig Ewington, Arthur Jones, and Steve Wille. Combining functional and object-oriented programming methodologies in a large commercial application. In *Proceedings of ICCL '98*. IEEE Computer Society, May 1998.

[Posnak et al.96] Edward J. Posnak, R. Greg Lavender, and Harrik M. Vin. Adaptive pipeline: An object structural pattern for adaptive applications. In *The 3rd Annual Conference on Pattern Languages of Programming, Monticello, Illinois,* September 1996.

[Prechel et al.97] Lutz Prechel, Barbara Unger, and Michael Philippsen. Documenting design patterns in code eases program maintenance. Submitted to Proc. Mod. & Emp. Studies of SW Evolution, January 1997.

[Pree94] 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 Verlag, July 1994.

[Quibeldey-Cirkel94] Klaus Quibeldey-Cirkel. Paradigmenwechsel im software-engineering. *Softwaretechnik-Trends*, 14(1):59–62, February 1994.

[Racko94] Roland Racko. Object lessons: In praise of Eiffel. *Software Development*, 2(12), December 1994.

[Ran95] Alexander Ran. Moods: Models for object-oriented design of state. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

[Reddy88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. *ACM Symposium on LISP and Functional Programming,* pages 289–297, July 1988.

[Rémy & Vouillon97] D. Rémy and Jirtme Vouillon. Objective ML: a simple object-oriented extension of ML. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Reynolds75] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages,* pages 157–168. IFIP Working Group 2.1 on Algol, 1975.

[Reynolds96]	John C. Reynolds. Private communication. ACM State of the Art Summer School: Functional and Object Oriented Programming, September 1996.
[Riehle et al.97]	Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, Addison-Wesley, 1997.
[Riehle97]	Dirk Riehle. Composite design patterns. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 218–228, 1997.
[Rist & Terwilliger95]	Robert Rist and Robert Terwilliger. *Object-Oriented Programming in Eiffel*. Object-Oriented Series. Prentice Hall, 1995.
[Ritchie84]	Dennis M. Ritchie. A stream input-output system. *AT&T BGell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.
[Rohlfing78]	Helmut Rohlfing. *Pascal: Eine Einführung*. Hochschultaschenbücher 756. B.I. Wissenschaftsverlag, Mannheim, Wien, Zürich, 1978.
[Rozas93]	Guillermo J. Rozas. *Translucent Procedures, Abstraction without Opacity*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1993.
[Rumbaugh et al.97]	James Rumbaugh, Grady Booch, and Ivar Jacobson. *Unified Modeling Language, version 1.0*. Rational Software Corporation, January 1997.
[Rumbaugh91]	J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, 1991.
[Sargeant et al.95]	John Sargeant, Steve Hooton, and Chris Kirkham. Ufo: Language evolution and consequences of state. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 48–62, April 1995.

[Sargeant93] John Sargeant. Uniting Functional and Object-Oriented Programming. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 1–26. First JSSST International Symposium, November 1993.

[Sargeant95] John Sargeant. How useful is laziness? News article of comp.lang.functional, August 1995.

[Sargeant96a] John Sargeant. ”imperative” and ”declarative”. comp.lang.functional, April 1996.

[Sargeant96b] John Sargeant. Personal communication, April 1996.

[Schmidt85] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[Schmidt86] D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. Allyn and Bacon, Inc, Boston Mass., 1986.

[Schmitz92] Lothar Schmitz. Using inheritance to explore a family of algorithms. *Structured Programming*, 13(2):55–64, February 1992.

[Schneider91] Hans-Jochen Schneider, editor. *Lexikon der Informatik und Datenverarbeitung*. 3. Auflage. Oldenbourg, 1991.

[Schroeder95] Ulrik Schroeder. *Inkrementelle, syntaxbasierte Revisions- und Variantenkontrolle mit interaktiver Konfigurationsunterstützung*. PhD thesis, TH Darmstadt, 1995.

[Seiter et al.96] Linda Seiter, Jens Palsberg, and Karl Lieberherr. Evolution of object behavior using context relations. In *ACM SIGSOFT’96, Fourth Symposium on the Foundations of Software Engineering*. ACM Press, San Francisco, California, October 1996.

[Sestoft88] Peter Sestoft. Replacing function parameters by global variables. Diku, University of Copenhagen, October 1988.

[Shalit et al.92] Andrew Shalit, Jeffrey Piazza, and David Moon. *Dylan — an object-oriented dynamic language*. Apple Computer, Inc., 1992.

[Shan95] Yen-Ping Shan. Smalltalk on the rise. *Communications of the ACM*, 38(10):102–104, October 1995.

[Shang94] David L. Shang. Covariant specification. *ACM SIGPLAN Notices*, 29(12):58–65, December 1994.

[Shang95a] David L. Shang. Covariant deep subtyping reconsidered. *ACM SIGPLAN Notices*, 30(5):21–28, May 1995.

[Shang95b] David L. Shang. Transframe: The language reference, September 1995.

[Shank90] Roger C. Shank. *Tell Me A Story*. Charles Scribner's Sons, NY, 1990.

[Shaw96] Mary Shaw. Some patterns for software architecture. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[Siegel95] David M. Siegel. Pass by reference. In *Patterns mailing list Digest*, Patterns Digest. patterns-discussion@cs.uiuc.edu, March 1995.

[Smith95] David N. Smith. *Dave's Smalltalk FAQ*. D. N. Smith, dnsmith@watson.ibm.com, July 1995.

[Snelting et al.91] Gregor Snelting, Franz-Josef Grosch, and Ulrik Schroeder. Inference-based support for programming in the large. In *Proceedings of the* 3^{rd} *European Software Engineering Conference*, LNCS 5550, pages 396–408, Milano, Italy, October 1991.

[Snelting86] Gregor Snelting. *Inkrementelle semantische Analyse in unvollständigen Programmfragmenten mit Kontextrelationen*. PhD thesis, TH Darmstadt, 1986.

[Snyder86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *Proceedings OOPSLA '86*, 21(11):38–45, November 1986.

[Stepanov & Kershenbaum86] A. Stepanov and A. Kershenbaum. Using tournament trees to sort. Technical Report 86–13, Center for Advanced Technology in Telecommunications, Polytechnic University, 1986.

[Stepanov & Lee94] A. Stepanov and M. Lee. The standard template library. ISO Programming Language C++ Project. Doc. No. X3J16 / 94-0095, WG21 / NO482, May 1994.

[Sun97] Java JDK 1.1 guide, 1997.

[Szyperski92] Clemens A. Szyperski. Import is not inheritance – why we need both: Modules and classes. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 19–32, Utrecht, The Netherlands, Springer-Verlag, July 1992.

[Thierbach96] Dirk Thierbach. Ein kalkülbasierter Vergleich zwischen objektorientiertem und funktionalem Paradigma. Master's thesis, TH Darmstadt, March 1996.

[Thierbach97] Dirk Thierbach. Personal communication, July 1997.

[Thomas & Weedon95] Pete Thomas and Ray Weedon. *Object-Oriented Programming in Eiffel.* International Computer Science Series. Addison-Wesley, 1995.

[Thomas95] David Thomas. Ubiquitous applications: Embedded systems to mainframe. *Communications of the ACM*, 38(10):112–114, October 1995.

[Thompson89] Simon Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13:181–218, 1989.

[van Rossum91] Guido van Rossum. Linking a stub generator to a prototyping language (python). *Proceedings of EurOpen Spring Conference*, 1991.

[van Yzendoor95] Arjan van Yzendoor. Cyclic datastructures in functional programming. Article: 5784 of comp.lang.functional, November 1995.

[Wadler & Blott89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989.

[Wadler85] Philip Wadler. How to replace failure by a list of successes. *Functional Programming Languages and*

Computer Architecture, pages 113–128, September 1985.

[Wadler87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, Germany, January 1987.

[Wadler89] Philip Wadler. Theorems for free! In 4^{th} *International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.

[Wadler90] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[Wadler92] Philip Wadler. The essence of functional programming. *Proceedings of* 19^{th} *Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[Walden & Nerson95] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture: Analysis and Design of reliable Systems*. Prentice Hall, 1995.

[Wampler94] Dean Wampler. Where's the beef? patterns to chew on (first in a series). In *Patterns mailing list*. patterns-discussion@cs.uiuc.edu, December 1994.

[Ward94] Martin P. Ward. Language-oriented programming. *Software Concepts & Tools*, 15(4):147–161, 1994.

[Waters79] Richard C. Waters. A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, 5(3):237–247, January 1979.

[Watt et al.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.

[Wechler92] Wolfgang Wechler. *Universal Algebra for Computer Scientists*. EATCS 25. Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[Wegner87] Peter Wegner. Dimensions of object-based language design. *OOPSLA '87*, 22(12):168–182, December 1987.

[Wegner90] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger*, 1(1):3–11, August 1990.

[Wegner95] Peter Wegner. Models and paradigms of interaction. Technical Report CS-95-21, Deparment of Computer Science, Brown University, Providence, Rhode Island 02912, 1995.

[Weinand & Gamma94] André Weinand and Erich Gamma. Et++ — A portable, homogenous class library and application framework. *Proceedings of the UBILAB '94 Conference*, pages 66–92, September 1994.

[Wikström87] Ake Wikström. *Functional Programming Using Standard ML*. International Series in Computer Science. Prentice Hall, 1987.

[Wilson95] Neil Wilson. Const correctness with eiffel classes. Article: 6749 of comp.lang.eiffel, May 1995.

[Winston & Horn84] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley, 2nd edition, 1984.

[Wirfs-Brock et al.90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Yu & Zhuang95] S. Yu and Q. Zhuang. Software reuse via algorithm abstraction. In Raimund Ege and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 17*, Prentice Hall International, London, 1995.

[Zimmer94] Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1994.

Index