Insight ETHOS
on object-orientation in operating systems

Author(s):
Szyperski, Clemens

Publication Date:
1992

Permanent Link:

https://doi.org/10.3929/ethz-a-000666071 →

Diss. ETH Nr. 9884

Insight ETHOS: On Object-Orientation in Operating Systems

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY (ETH) ZÜRICH

for the degree of
Doctor of Technical Sciences

presented by
CLEMENS ALDEN SZYPERSKI
Dipl.-Ing., Aachen Institute of Technology (RWTH), Germany
born on October 19, 1962
citizen of Germany and the United States of America

accepted on the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. H. Mössenböck, co-examiner

1992

To Bianca, my parents, and all my family ...

Acknowledgements

Wer immer nach dem Zweck der Dinge fragt
wird ihre Schönheit nie verstehen.
(Halldor Kiljan Laxness)

The adventure of writing a book in general, and a thesis in particular is hard to imagine without the support from many people. For encouraging my work I am most indebted to my advisor Prof. N. Wirth. His ability to combine ingenuity, pragmatism, and Occam's razor is hardly paralleled. For a liberal supervision of my project I would like to express my warmest thanks to Prof. H. Mössenböck. He introduced me to the ideas of Object-Orientation and made possible the continuous existence of a friendly atmosphere in our group.

The Oberon project, jointly led by Proff. N. Wirth and J. Gutknecht, was a source of inspiration that never dried up. Oberon's simplicity and rigor were a constant yardstick to measure against. Smooth and productive, the work with the Oberon system and language was never less than pure fun.

All my colleagues at the Institute for Computer Systems were of significant help throughout all the ups and downs. Josef Templ created the starting point for my project by designing the key aspects of the language Oberon-2. Marc Brandis, Robert Griesemer, Cuno Pfister, Josef Templ, and Wolfgang Weck contributed with numerous discussions on all topics of the thesis. Robert Griesemer was always willing to try out new releases of Ethos and ported some of his own applications. Régis Crelier provided the rock-solid Oberon-2 compiler and was always willing to discuss language design and implementation issues. Beat Heeb was a dependable source of profound technical knowledge in all areas of subtle hardware problems. Beat Stamm shared his insights into aspects of font and image rastering. Stefan Ludwig proofread the thesis and encouraged with his enthusiasm for new ideas. Finally, Wolfgang Weck provided the handy user interface for the PAT search engine by Prof. G. Gonnet that was used to retrieve definitions and quotations from the Oxford English Dictionary.

Contents

Abstract

Ethos: The characteristic spirit, prevalent tone of sentiment, of a people or community; the "genius" of an institution or system. (Aristoteles – Rhetorics ii. xii–xiv)

Ethos is both, an Elaborate Thesis on Operating Systems and an Ephemeral Thesis Operating System: It is the name of a project aiming at design principles of Object-Oriented Operating Systems, as well as the name of an actual implementation used as a case study to validate the found design principles. This thesis covers the Ethos project and system.

Results of the *Ethos project* are generalizations of design principles for object-oriented systems, namely the Carrier/Rider Separation, and the Directory Object concept. The former aims at a widely applicable scheme for separating data management from data access tasks. The latter generalizes the notion of prototype objects to manage integration of extensions into the running system. Another result of the Ethos project is the introduction of a flyweight pre-emption concept called *engines*, with the key attribute that engines may share a single stack.

The *Ethos system* is an evolutionary successor of the Oberon system, and is implemented using the language Oberon-2. The Ethos system is based on a strongly typed hierarchy of abstractions. Each abstraction has a default implementation, but multiple implementations of the same abstraction may be used simultaneously. As the extensibility of Ethos goes down to the very machine level, Ethos is well suited to prototype new operating system services.

Ethos contains a finalizing garbage collector with integrated support for identity directories, a facility for dynamic integration of modules into the running system, and a framework for developing extensions. The document editor Write has evolved as part of the Ethos system and is used as the primary medium for user activities. (Write has also been used to prepare this thesis.) A refined version of the Oberon user interface is provided in the standard setting.

Kurzfassung

Ethos ist zugleich der Name eines Projekts und der Name eines Systems. Das Projekt zielt auf die Isolierung und Bestimmung geeigneter Designprinzipien objektorientierter Betriebssysteme ab, während das System diesen Prinzipien folgend entworfen wurde. Als Fallstudie eines konkreten Betriebssystems untermauert das System die grundlegenden Gedanken des Projekts.

Wesentliche Resultate des Projekts sind Generalisierungen bekannter Designprinzipien objektorientierter Systeme. So werden "Model-View-Controller"-artige Konzepte zum *Carrier-Rider* Trennungsprinzip verallgemeinert, das eine breite Anwendung in der systematischen Trennung von Datenzugriff und Datenverwaltung findet. Weiter werden "Prototypes" zum *Directory Object* Konzept verallgemeinert, um eine feine Kontrolle über die Anbindung einer Implementationsvariante an eine gegebene Abstraktion zu ermöglichen. Schliesslich wird ein besonders leichtgewichtiges Konzept, genannt *Engine*, zur Realisierung von preemption-basierten Verfahren eingeführt. Das spezielle Merkmal von Engines ist die gemeinsame Benutzung eines Stacks.

Das Ethos-System, das vollständig in der Sprache Oberon-2 implementiert wurde, kann als evolutionärer Nachfolger des Oberon-Systems verstanden werden. Das Rückgrat von Ethos bildet eine streng typisierte Hierarchie von Abstraktionen. Zu jeder Abstraktion gehört mindestens eine Standardimplementierung – die gleichzeitige Verwendung mehrerer, alternativer Implementierungen der selben Abstraktion ist jedoch stets möglich. Da die Erweiterbarkeit und Austauschbarkeit von Abstraktionen im Ethos-System bis auf die Ebene der eigentlichen Maschine hinab reicht, kann Ethos auch zum Entwickeln von Prototyp-Implementierungen neuer Betriebssystemfunktionen verwendet werden.

Ethos ist mit einem finalisierenden Garbage Collector ausgestattet, der Identitätsverzeichnisse direkt unterstützt. Das dynamische Laden und Integrieren neuer Module in das laufende System ist jederzeit möglich. Die Konstruktion neuer Erweiterungen wird durch vorgezeichnete Rahmenkonstruktionen ("Frameworks") erleichtert. Als primäres Medium für alle Benutzeraktivitäten wird der als Bestandteil des Ethos-Systems entwickelte Dokumenteneditor Write verwendet. (Mit Write wurde auch die vorliegende Arbeit erstellt.) Schliesslich gehört eine Weiterentwicklung der Oberon-Benutzungsoberfläche zur Grundausstattung.

Guide to the Reader

If it were not for the standing danger of having one's head talked off one's shoulders.
(D. Gerald, 1897)

This thesis covers a rather large scope. To reduce the prerequisites on the side of the reader, introductions to principles of object-orientation and operating systems are given, as far as relevant. The current understanding of programming distinguishes *programming in the small* from *programming in the large*. Object-orientation is a technique to deal with problems in the latter area. Operating systems are of a size and complexity that needs to be tackled using such techniques. Parts of this thesis have text-book character, describing the interaction of design principles and system design.

To ease the task of reading, the thesis has been organized to allow for different pathways through its chapters. The preamble may serve for a brief overview, perhaps followed by the conclusions (Chapter 4).

The main text is organized into three chapters. In Chapter 1 basic concepts are explained. A discussion of specialized concepts follows in Chapter 2. Chapter 3 – measured by number of pages and level of detail – is by far the largest. It covers the case study *Ethos*, an object-oriented operating system that has been designed and implemented to explore the concepts covered in the first two chapters. While the path from Chapter 1 to Chapter 3 follows a top-down approach, Chapter 3 itself describes the Ethos system in a bottom-up fashion. This follows the idea that *concepts* are best explained *top-down*, while *systems* – with upper parts depending on lower ones – are better explained *bottom-up*. Extensive conclusions in Chapter 4 consider lessons learned and point at possible directions of future research. The commented Bibliography supports *reference driven access*.

A reader may use the forward thread through the thesis and read chapters one to three in order, following the top-down development from concepts to concrete examples. Otherwise, the first two chapters may be selectively skipped, while concentrating on Chapter 3 in a first pass. This way the concrete examples from the case study may serve as a reading aid when returning to the first two chapters.

To stress the main train of thought, potentially distracting details have been formatted as *insets*. This paragraph is an example for an inset. While such insets may contain interesting information for the curious and patient reader, they may safely be skipped when aiming at an understanding of the "big picture".

Preamble

There is no contradiction between finite extent, and infinite extensibleness.
(E. Caird, 1877)

In the following a brief overview of the thesis is given. It assumes a certain familiarity with the field as it excludes explanations or definitions.

What has been achieved? The main objective of this thesis is two-fold: On the one hand, it was tried to isolate and understand some important design principles applicable when building an object-oriented operating system. On the other hand, the applicability of such principles as well as of the object-orientation principle in general was investigated by designing and implementing the object-oriented operating system Ethos. Ethos is considered a case study exhibiting interesting aspects, but intentionally neglecting others that are less related to the topic of this thesis.

The search for design principles has been fruitful. A generalization of several existing principles led to the new *Carrier/Rider Separation* design principle. Carrier/Rider is both, a generalization of the Model/View concept known from Smalltalk, and of the File/Rider concept found in the Oberon system. Carriers and Riders separate building blocks supporting general forms of data management and data access, respectively. Another important new design principle are so-called *Directory Objects*, which can be seen as a generalization of prototype objects.

The case study led to the successful design and implementation of Ethos, which can be seen as an evolutionary successor of the Oberon system. Ethos has been implemented entirely using the single programming language Oberon-2. Oberon-2 developed out of Oberon during the early stages of the Ethos project. Experiences with Oberon-2 while implementing Ethos led to several adjustments of the language.

Several algorithmic and structural methods have been developed or originally combined in order to implement Ethos. Among the interesting ones are: An integrated support for object finalization, identity directories, and garbage collection; certain meta-programming facilities, e.g. used to support generic object externalization and internalization; the flyweight pre-emption concept of "engines" for certain real-time demands; a way to support various naming models in file systems by means of extensions; and an interesting extension model for document-based applications. The latter has also been used to implement the Write editor for the Oberon system. Write has been used to prepare this thesis.

The Ethos project shows that it is feasible to support run-time extensibility on all levels of a system when using a hybrid object-oriented programming language for its implementation. By relying on a strongly typed language and a statically checked type-safe implementation, reliance on hardware protection support is mostly avoided. It is sometimes claimed that type safety can be a burden when trying to implement something swiftly ("rapid prototyping"). However, the price paid was always found to be worth it: The resulting system can be trusted and locating errors after extending the system was almost always found to be easy.

What is new or different? As mentioned above, Ethos can be viewed as an evolutionary successor of the Oberon system. This is most apparent when looking at the user interface or the typical mode of operation: Both are almost identical in Ethos, although improved in numerous ways. However, looking behind the scenes the differences are obvious. Ethos is entirely built around the concept of exchangeable implementations of defined abstractions. In contrast, the low-level components of Oberon – including the file system – are implemented as abstract data types, and are not extensible.

In Ethos the limits of extensibility are pushed down to the very machine level. For example, it is possible to extend such basic components as the file system, the module loader, or even the memory management. Such flexibility is needed when developing new operating system services: Ethos allows to work with experimental versions of operating system services just the same way as it allows to work with experimental extensions on higher levels of the system. Extensions on all levels can be installed and removed at run-time.

In a sense, the uniform treatment of extensible abstractions on all levels blurs the distinction between application and operating system even more than this was already the case in the Oberon system. The achieved flexibility comes close to what is known from systems like Smalltalk, while providing the safety and comprehensibility of a strongly typed and statically checked language, and the efficiency of a fully compiled system.

What can be learned? Working on proper design principles as well as designing and implementing a working operating system opened many problems and some research areas. Some of the problems have been solved, some have been isolated. Remaining problems of a more general nature can be seen as entry points to new or still open research areas.

Among the lessons that have been learned during the course of this work is the conviction that hardware should be constructed to support safe programming, but otherwise should stay out of the way. Safety in principle cannot be achieved by hardware facilities alone, but must be left to proper software models. For a system to be effectively and efficiently extensible, it is important to guarantee safety properties on a fine granularity, well adjusted to

the implemented abstractions. Languages with expressive and powerful type systems are one step. Another one is the strict implementation of type-safety, where loopholes should not exist.

Moving most (if not all) of the safety issues to the language and its implementation is a significant burden for the language designer. However, another lesson learned from the Ethos project comes in handy: A *single*, sufficiently simple, orthogonal, yet powerful language can be used for all purposes, from the implementation of device drivers, over applications specific and system configuration purposes, up to user-level scripting. While Oberon-2 is not the "final language" achieving all these goals, it comes close enough to demonstrate the feasibility of this approach.

1 Introduction

The theme of this thesis is Object-Orientation in Operating Systems. Hence, this introduction aims at clarifying the terms Object-Orientation and Operating System. As will become apparent, both terms are rather imprecise. Instead of developing precise definitions, the important concepts related to the two terms will be worked out and set into a context suitable for the course of this thesis.

1.1 Basic Concepts of Object–Oriented Programming

Object-Orientation is currently (1992) one of the buzz-words in computer science. Using such a word has the advantage of attracting attention, but the apparent disadvantage of causing many, hardly controllable associations. To provide some stable ground for the rest of the thesis, this section summarizes the important aspects of object-orientation, as far as they are relevant to the presented work. Building on the basic definitions, a motivation is given showing that it is useful to build something in an object-oriented fashion. Thereafter, remarks concerning the structure of object-oriented systems follow. Finally, criteria are given for programming languages that are supposed to support object-oriented programming.

1.1.1 What is Object–Orientation?

The concept of object-orientation can be captured as a list of more basic concepts:

Object-Orientation :=
Encapsulation & Polymorphism & Late Binding [& Inheritance]

This "equation" is critical since no commonly accepted definition of object-orientation exists (e.g. [Weg90]). However, all essentials are included that should not be left out. (Code-)Inheritance is an exception: Its historical relation to object-oriented languages makes it important, yet it can be considered optional (see below). Instead of a philosophical discourse on the meaning of terms like *object*, some concise characterizations of the essentials are given. In this subsection, no attempt is made to motivate, judge, or justify

any of the mentioned concepts. Also, it is not claimed that all important aspects of an object-oriented language are covered, as many otherwise important aspects are not necessary to make a language object-oriented.

Encapsulation. The term object is based on the distinction of *inside* and *outside*. An object is fully encapsulated if its state (inside) can be observed and changed (from the outside) *only* by means of operations defined by the object. Such operations are called *methods*, and invoking a method of an object is termed *sending a message* to that object. The object is then called the *receiver* of the message. Sometimes a set of objects should co-exist in a tight manner, where the innards of each member of the set should be directly accessible to the other members. Hence, it may be useful to encapsulate on a coarser granularity than that of objects, e.g. on a class or module level. (Classes are introduced below, a discussion on modules follows in Chapter 2.)

Polymorphism. A thing is called *polymorphic* if it can assume several forms, e.g. a polymorphic variable may hold values of different types. If a parameter of an operation is polymorphic, the operation is called polymorphic.

> For operations, one has to distinguish between the types of *formal* and of *actual parameters*, where the former types are those given in the signature declaration of the operation, and the latter types are those of the arguments the operation is actually applied to. In the following, formal parameters are simply called parameters, and actual parameters are called arguments.

If the types of values that may be assigned to a variable are unrestricted (e.g. Smalltalk [GR83]), the resulting polymorphism is called *ad-hoc*. In this case it is in general undecidable whether a certain operation may be applied to a variable – i.e. whether the operation is defined for the current value of the variable. Hence, checking validity of operations when using ad-hoc polymorphism requires run-time checking, and therefore the association of a type tag with every value.

To increase the static semantics of a program, it is useful to declare *types*. A type restricts the values that may be assigned to a particular variable or used with a particular operation. In traditional languages (e.g. Modula-2 [Wir82]), value types must exactly match variable or parameter types, making all constructs *monomorphic*.

Often it is more useful to allow use of all value types that somehow *conform* to a given type, where a type *T1* conforms to a type *T* if values of type *T1* can be used wherever values of type *T* are expected. Languages like Simula-67 [DMN68], Object-Pascal [Tes85], or Oberon [Wir88b] define a *partial ordering* on types: For certain types (*classes* in Simula-67, *objects* in Object-Pascal, *records* in Oberon), a type can be defined to be a *direct subtype* of another type (its *direct base type*). The resulting polymorphism is called *inclusion polymorphism*, as a variable of a certain type *T* may hold values that

are of type *T* or of any subtype of *T*.

The relation of true subtypes must be acyclic. If a type may have at most one direct base type, the resulting ordering forms a *forest* (e.g. Oberon). If all types (except one, having no base type) have exactly one direct base type the ordering forms a *tree*. A general form of type conformance results if a type may have multiple direct base types: Then the type graph can be completed to form a general *lattice* (e.g. C++ [ES90]). Besides being enforced by declaration, a language may define type conformance on the basis of signature subsets (e.g. Quest [Car89]). For example, a Quest record type (a:A, b:B) is automatically a subtype of the types (a:A) and (b:B) since it includes the latter signatures.

Finally, the types of a cluster of polymorphic variables may be coupled in a way that all values assigned to variables of such a cluster must be of the same type. This restricted form of polymorphism is called *genericity*. Common schemes to couple the types of variables are class templates (e.g. C++) or generic modules (e.g. Ada [DoD80]).

Besides using inclusion polymorphism, polymorphic *operations* may be constrained using more refined concepts. This is done by turning the type constraints of parameters themselves into (type-)parameters of the operation, e.g. *f(Type::Kind; param:Type)*, where *Type* is a type-parameter of kind *Kind* and *param* is a normal parameter of type *Type*. The resulting operation is said to support *parametric polymorphism*, which can be split further into simple (unquantified) *parametric*, *bounded*, and *f-bounded quantification* [CCH*89] [Har91]. Bounded quantification requires the type arguments to conform to a certain given type, while f-bounded quantification additionally requires the implementations of the type arguments to inherit from a certain class (see below). General forms of parametric polymorphism are not available in most of the current object-oriented languages. An example for a restricted form is the *like current* construct present in Eiffel [Mey92]. While being a topic of ongoing research (e.g. [CHO92]), aspects of special forms of polymorphisms are not pursued any further here.

The type of an operation *B* is said to conform to the type of another operation *A*, if all in-parameters of *B* are of equal or more general types than the corresponding parameters of *A*, and all out-parameters of *B* are of equal or more special type than the corresponding parameters of *A*. Hence, specializing (subtyping) an operation supports specializing its out-parameters, but requires generalizing its in-parameters! If a parameter of a more specialized operation has a more specialized type, the parameter is called *covariant*; if a parameter of a more specialized operation has a more general type it is called *contravariant*. Languages supporting forms of parametric polymorphism (e.g. Eiffel's *like current*) are in danger of violating the contravariance rule of operator conformance and hence risk being type-unsafe.

A simple example results from looking at special operators that are functions of the form **f**: $A \rightarrow B$. Another function defined as **f'**: $A' \rightarrow B'$ can be used wherever **f** could be used, if $A \subseteq A'$ (A' is a generalization of A) and $B' \subseteq B$ (B' is a specialization of B) holds, i.e. if **f'** can accept the domain of **f** and does not exceed the range of **f**. Figure 1.1 (taken from [Car89]) illustrates the situation.

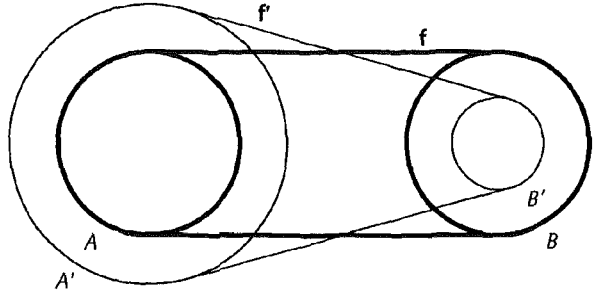


Figure 1.1 – Covariance of function domains vs. Contravariance of function ranges.

From the variance rules for in- and out-parameters follows that the type of in-out-parameters must not be varied when subtyping an operator.

Late Binding. Often a set of polymorphic operations exists, where each of the operations could be applied to a given set of arguments. Then the selection of the actual operation can be based on the values or types of the arguments. For example, if two operations *increment* are defined for parameter types *Real* and *Integer*, respectively, then for an argument of type *Integer* the increment for integers should be selected.

An operation returning another operation based on its arguments is commonly termed a *higher-order function.* In the specific case, where an operation is chosen *at run-time*, the deferred binding of the actual operation to the point (time and locus) of invocation is called *late binding* (or *dynamic binding*).

If the argument *types* determine the choice of binding, the operation is called *type-driven.* If a single, distinguished argument type is used for selection purposes, the operation is called a *method.* If a language distinguishes between types and classes (see below), operations may instead be *class-driven.* This is the case in languages like Smalltalk, where types are not explicitly present at all. If several argument types are evaluated for selection purposes, the operation is called a *multi-method.* Multi-methods are found in CLOS [DG87] or in Cecil [Cha92]. To respect object encapsulation, a method should avoid direct side-effects in objects other than the receiver. Hence, multi-methods should have a single, distinguished "receiver" parameter.

Since methods are chosen based on the type of their receiver, the receiver type can change in a covariant way. This is important, as it allows a method to directly operate on values of the appropriate subtype. When using multi-methods, contravariance problems can be avoided by providing an individual method implementation for the Cartesian product of all sets of allowed argument types.

If the argument *values* are used to select the operation, the operation is called *value-driven*. A typical form is the use of a procedure variable as part of one of the arguments. Then the operation currently bound to the variable is used at invocation time. Value-driven operations allow for a higher degree of flexibility, but weaken the static properties of a given program. Examples for value-driven approaches are the handlers in Oberon [WG89], or the method slots in Self [US87].

Inheritance. The concept of inheritance is not well defined in the literature on object-orientation. In the context of this thesis, inheritance is understood as *code inheritance* or – synonymously – *subclassing*. The code that implements the methods of a certain type is called a *class*. If a class builds on the code of another class, the former is called a *subclass* of the latter. (Vice versa, the latter is called a *base class* or a *superclass* of the former.) If a class may have at most one base class, the relation is called *single inheritance*, otherwise *multiple inheritance*. A typical form of subclassing is the automatic inclusion of all methods implemented in the base classes. Such methods are called *inherited*. A subclass may *override* an inherited method by providing a new implementation for it, and the overriding method may invoke the overridden method using a so-called *super-call*.

Certain languages replace the static subclassing hierarchy by a dynamic relation between objects and *parent objects*. Then, an object *forwards* all messages that it cannot fully handle to one of its parent objects.

For methods, the receiver may be used to invoke another method of the same receiver. This way, the call-graph becomes recursive, where the recursion structure is dynamic, depends on the class of the receiver, and is defined by the inheritance structure of the participating classes. Since the receiver is often termed *self*, the resulting recursion is called *self-recursion* (and the resulting polymorphism *f-bounded*). If forwarding is used while preserving the identity of the original receiver, the concept of *delegation* results. This is used in Self to replace subclassing. Delegation can be simulated in languages not directly supporting it [JZ91]. Also, subclassing and delegation can be simulated by each other [Ste87].

Code inheritance is one of the most original aspects of early object-oriented languages, but in this thesis it is *not* considered a defining feature of object-orientation. Indeed, object-oriented languages exist that do not support automatic code inheritance. For example, Oberon supports the object-oriented

style by resorting to object-driven operations, yet Oberon has no explicit language features for classes or subclassing. Oberon-2 adds the option of using type-driven operations, making it very similar to Simula-67.

Subtyping versus Subclassing. Types and subtyping are used to express conformance relations, while classes and subclassing are used to package implementations and express code inheritance relations. In principle, these are two very different concepts, and examples are given in the literature that show that sometimes the subclass and subtype hierarchies contradict each other, if both are chosen to be optimal for the problem at hand [CHC90].

Still, many object-oriented languages with a strong typing scheme identify subtyping with subclassing; examples are Simula-67 and Oberon-2. However, languages exist that carefully separate types and classes; an example being Emerald [Hut87][Bla91]. There, the idea is that a class implements a type, and that the same class may implement many types. Likewise, a type may have many alternative implementations in form of many classes implementing the same type.

> In a sense, the language Oberon separates types and classes, since multiple implementations of an object-driven handler may exist, and one of them may be chosen on a per-object basis. However, Oberon has no explicit constructs to describe classes or subclassing, weakening its static expressiveness.

An issue related to the proper use of subtyping is the often cited *is-a* relation, where it is meant that *T' is-a T* if objects of type *T'* can be used wherever type *T* is expected. The *is-a* relation is stronger than the type conformance relation, since it covers aspects that are not decidable, and hence cannot be specified using a type system. However, a strong type system can help to express significant parts of the *is-a* relation.

1.1.2 Why Build Something in an Object-Oriented Fashion?

Having described the various concepts related to object-orientation, the next question to pursue is the relevance of the object-oriented approach: Why at all is it useful to build something in an object-oriented fashion? This subsection covers some of the general advantages, split into static, dynamic, and evolutionary aspects. Additionally, *extensibility* is introduced as a major motivation for object-orientation. However, object-orientation in the sense defined in the previous subsection does not automatically guarantee extensibility. Hence, a new term is defined, capturing the combination of object-orientation and extensibility. Finally, a few warnings are given that hint at the limits of object-orientation as a problem solving approach.

Static Aspect. Using *polymorphic constructs* one can often avoid explicit *case analysis*, i.e. case switches distributed all over a program. Thereby a program can be better organized and thus be easier to understand and to maintain. The potential for static program checking is increased if polymorphisms are used instead of scattered case switches. Also, new variants can be added later without having to modify arbitrary portions of the system. The latter is a prerequisite for extensible systems.

Dynamic Aspect. Late binding allows "old" code to invoke "new" one: A program can invoke something that did not even exist when the program was implemented. Hence, code can be made extensible without requiring changes to the extended code. For a system to be safely extensible, an appropriate type system is crucial.

Also, extending a system is not really possible if separate compilation (on some level of granularity) is impossible. For example, the system-level type-checking introduced in the current Eiffel version [Mey92] "solves" the covariant typing loophole of previous Eiffel versions (cf. 1.1.1, [Coo89]). As a result, an Eiffel *library* may fail to type-check in the context of a program just because a new library *client* has been added!

To capture the additional needs of an extensible system, the term *Extensibly Object-Oriented* is introduced:

Extensibly Object-Orientation :=
Object-Orientation & Separate Compilation & Type Safety

Requiring the possibility of separate compilation *in principle* imposes certain restrictions on a programming language and its implementation. In general, it requires that a *compilation unit need not be re-compiled, just because a new compilation unit has been added* to the system.

In particular, a language that relies on global program analysis to type-check a program does not fulfill the separate compilation requirement. This is often overlooked in the literature when discussing typing issues of object-oriented languages. Many "solutions" are based on type reconstruction, where the compiler reconstructs type information not or only partially given by the programmer (e.g. [PS90]). However, correct type reconstruction usually requires global program analysis.

A particular implementation may of course use global analysis to improve code quality. Likewise, run-time compilation techniques may be used (e.g. [DS84][HCU91]). However, such *optimizations* must be carefully distinguished from the requirement of separate compilation *in principle*. Also, the granularity of separate compilation (classes, modules, etc.) is left to the language design.

Evolutionary Aspect. Whether a system is extensible in the sense defined above can be tested easily: It must be possible to compile a new extension without affecting the correctness of previously existing code. An important aspect that is often neglected is the *practical* support for *dynamic integration*, i.e. the possibility to indeed load and bind extensions at run-time, augmenting or partially replacing existing functionality. A system supporting dynamic integration can grow with the needs of its users. The recent addition and increasing popularity of *dynamic link libraries* in traditional systems like UNIX serves as an example.

If a *system* combines object-orientation with dynamic integration, it supports extensible object-orientation: Previously loaded code uses dynamic integration and binding to invoke code that was not even *known* when loading the invoking code. (To support version updates, extensions should also be removable at run-time, as is possible in Oberon [WG89]. This has been "forgotten" in Cedar [Tei84] at the price of frequent re-booting.)

It has been noted above that the use of multi-methods may ease the covariance problem [Cha92]. However, multi-methods contrast with dynamic integration: If several extensions are created independently, then it is hard or impossible to fill the resulting Cartesian product. For example, the multi-method *less(x, y: Number)* needs to be specialized in four different ways when separately introducing *Integer* $\subseteq$ *Number* and *Real* $\subseteq$ *Number*, i.e. by defining the *less* for all parameter types of the Cartesian product $\{Integer, Real\} \times \{Integer, Real\} = \{(Integer, Integer), (Integer, Real), (Real, Integer), (Real, Real)\}$.

Some Warnings. Object-orientation is not a panacea for the general problem of programming, i.e. of designing and implementing programs solving specific problems. In fact, there are many cases where the object-oriented approach taken to an extreme destroys the comprehensibility of a system by introducing a far to high degree of freedom: It seems unwise to make everything – every "bit" – object-oriented.

1.1.3 Design Aspects of an Extensible Object-Oriented System

Assuming that a system should be implemented in an extensible object-oriented (EOO) fashion: What is the best way to design it? This subsection lists a few rather general principles, while more detailed discussions may be found in Chapter 2.

Simplicity. There is no doubt about the power of simplicity. However, it is difficult to find simple yet adequate designs even for small systems. For larger systems, this is almost impossible if an ad-hoc approach is followed. Instead, certain design principles should be followed, and it may even be fruitful to

begin by creating new design principles to support the designing task. The application of design principles helps in achieving uniform and orthogonal designs. Both, uniformity and orthogonality are especially important when designing an extensible system.

Extensibility. Making a system extensible introduces the requirement of specifying duties, i.e. policies are needed to describe what providers and clients must do and what they should not do. This is already a necessity when using traditional library code. However, the heavy use of late binding in object-oriented designs tends to cause "library" parts to invoke parts of "clients". Thus the interaction between existing parts and extensions becomes much more complex, and a specification of what can be expected becomes more important. A typical approach is "programming by contract" [Mey88], where the use of an interface is compared to the signing of a contract. A "contract" includes typing constraints, preconditions, postconditions, and invariants. Formal contracts are beyond the scope of this thesis. However, the issue of strong typing as a part of a proper specification is stressed repeatedly when discussing particular problems.

Static Safety. To control the correctness of a program it is useful to allow for static checking of as many aspects as possible. In principle, a program should be checked against its complete specification. Since in the general case this is undecidable, at least the important decidable aspects of the specification should be isolated and checked statically. This is the main purpose of a type system. In addition to static checking, an implementation may need to resort to run-time checks to close security breaches in undecidable cases. Typical examples are run-time index and NIL checks.

To strengthen the static checking, it is useful to allow to distinguish between known (and therefore trusted) and client–added relationships. For example, a language may support modules and impose stronger barriers across module boundaries than it does inside of a module [Szy92b].

Often, static safety is ignored in favor of run-time hardware checks. However, for a system supporting extensibility on all levels it is impossible to do so: The degree of interaction between trusted and non-trusted parts of the system becomes too fine-grained to use primitive concepts like memory protection using a memory management unit.

Dynamic Safety. Run-time checks must be predictable. For example, a program which is about to index an array can guard the operation by checking the index range. If such a guard is omitted and the index is out of range anyway, a programming error is present and it is reasonable to have a run-time check abort program execution. However, approaches exist where run-time checks are in fact unpredictable. An example are "solutions" to the covariance problem (1.1.1) based on added type guards in overriding methods. Due to

late binding, a type guard applied to a method parameter introduces a run-time check *within the callee* that in general cannot be predicted at the point of method invocation. (A possible way to solve this problem is sketched in 4.4.)

Efficiency. Of course, a program needs to be *effective*, i.e. correct with respect to its specifications. However, it also needs to be *efficient*, i.e. free of unnecessary activities in time and space. The most common myth about efficiency is that the increasing efficiency of hardware makes the efficiency of software less and less important. Simple studies of non-linear algorithmic complexities suffice to reject this idea.

In the context of object-orientation the choice of the right granularity is critical. While certain language implementations make it quite efficient to turn everything into an object, they rely on the fact that most fine-grained objects are actually never extended. Thus, such systems effectively remove late binding in most places where fine-grained objects are used. Consequentially, designing a system to express everything – every "bit" – in terms of objects puts a heavy burden on the language implementation.

Often – and certainly with current languages and language implementations – it is preferable to design a system using a more modest object granularity. This is the basic motivation for supporting hybrid object-oriented languages, where the distinction between object-oriented and traditional parts is easy.

> It is noteworthy that the efficient implementations of pure object-oriented languages – where "efficient" is still only 50% of the performance achieved with more traditional languages [CU91] – even rely on the assumption that certain parts of the system are never extended! For example, [DS84] assumes that basic control structures – which in Smalltalk are expressed in terms of message invocations – are never overridden and thus can be compiled into efficient code. Other approaches, like the one for Self [CU91] tend to consume enormous amounts of memory.

1.1.4 What an EOO Supporting Language Should Provide

The introduction of the term Extensibly Object-Oriented (EOO) leads to the question what a language should provide to support EOO. In the following various language requirements are derived.

Extensibility. The definition of EOO requires the language to support separate compilation in principle. Usually, it cannot be expected this is done on an arbitrarily fine granularity. For examples, type definitions standing in a cyclic relationship usually cannot be compiled separately. Thus, a useful unit of compilation need be defined. Typically, *modules* are introduced into the

language and are explicitly required to be separately compilable [Szy92b].

Expression of Strong Constraints. For an extensible system the expression of constraints is important: Especially on the level of interfaces open to arbitrary extensions, it must be possible to precisely express constraints. A long way towards this goal can be gone with a sufficiently strong *type system.* (The meaning of "sufficiently strong" is an open research issue.) A type system should be decidable: A type requiring an integer (all operations allowed) to be prime does not fulfill this condition. Expressing minor details using a verbose type system should be avoided. Doing so subverts the intention that a well typed interface should be easy to read and easy to understand, as it forms an important part of a system's documentation. Especially in the OO context, it is not yet clear what constraints should be expressible within a type system.

Safety. Having a strong and decidable type system alone does not improve program quality. It is important that all type assertions are checked, i.e. that the language *implementation is type-safe.* Additionally, a few assertions are important yet undecidable. For example, an array should never be indexed outside of its index bounds, a pointer should never be dereferenced when it is NIL, and a type guard should only be passed if the guarded object is of an appropriate (run-time dependent) type. A certain amount of run-time checking is thus required. In particular cases, program analysis may allow for optimization of these. However, analysis must be limited to the granularity of compilation units to maintain the requirement of separate compilability in principle.

Efficiency. Using an object-oriented approach on all levels of abstraction can become quite costly. Even modern and complex compilation techniques (e.g. [CU91]) cause programs to execute only about half as fast as similar programs written in a style that is not purely object-oriented. Another issue is the static semantics that a program has: If everything is – at least in principle – based on object abstractions and late binding, then the static meaning of program fragments gets very weak. Hence, efficiency and comprehensibility reasons stand for a more modest approach, where object-orientation is deliberately used by the designer of a program to indicate planned and intentional degrees of freedom.

Often, it is claimed that the structural advantages of the object-oriented approach dominate the weakening of static semantics. In other words, the grouping of data structures with associated operations is felt important enough to use object-oriented structures even in the cases where static binding can be used. The idea is, that the compiler can detect such cases and generate static calls anyway. However, this contrasts strongly with the requirement for extensibility, as a separately compilable language cannot be implemented to remove late bindings crossing compilation units. For

example, once a method definition gets exported, it is never known whether an extension added later overrides the method, and thus requires late binding of invocations of this method. (However, run-time compilation techniques together with invalidation of generated code upon integration of new extensions can be used to optimize globally. The result is a significant increase in compiler and system complexity.)

Thus, the argument that late binding can be *statically* optimized by the compiler is only true for implementations that are fully encapsulated within a compilation unit, e.g. a module. Such encapsulated implementations are not extensible, and the advantage of using object-oriented approaches to structure the contents of a *single* module is relatively small. Or, to put it differently, if the structuring requirements are such that object-orientation seems to help, the language should provide for adequate static structuring schemes usable within, say, a module. Of course, such structures can well be similar to classes, except that the programmer makes the intended static binding *explicit* instead of relying on a smart compiler. (In C++, static binding of methods is even the default!)

To conclude, it seems unwise to express every "bit" in terms of objects. Instead, the nature of objects, i.e. the separation of inside and outside, should be kept in mind. The designer has to trade off, whether a certain abstraction requires to be encapsulated into a class, instead being expressed as an abstract data type, an abstract data structure, or even not being encapsulated at all. The language can support this design decision by providing a spectrum of abstraction tools that make the intended degree of freedom (of abstraction) explicit.

1.2 What is an Operating System?

The term *Operating System* is to be understood in an historical framework. The first computing systems loaded a single program at a time, executed it to its completion, and produced some output. The loading of a program and the return of the output to the programmer was done by human operators. With the introduction of *batch processing* services this was automated by means of a constantly loaded *monitor* program, which in turn loaded and executed a sequence (a batch) of programs.

Later, the demand for a machine per user grew, while prohibitive hardware costs made it impossible to install enough machines. To overcome this situation *time-sharing* services were introduced, based on a *multi-programming* model: Each user was provided with the illusion of working on a private (virtual) machine of his own. This significantly increased the functional requirements of the rather primitive monitoring program: It now had to

allocate shared resources (primary and secondary memory, input/output devices) and to *schedule programs*. The resulting control programs were termed *Operating Systems*, and they had to solve a bunch of complicated problems, including those related to deadlocks and starvation. Instead of going into the details here, the reader is refered the extensive literature on classical operating system problems and designs (e.g. [Dij68][PS83][Tan87] [Tan92]).

For protection reasons the traditional operating systems were entirely separated from the application programs they executed. On modern workstations, assigned to individual users, this need not be so. In fact, the definition of what belongs to an operating system and what not, got more and more blurred. This section looks at various OS models and the characteristics of the resulting operating systems.

1.2.1 Older Operating System Models

Traditional (time-sharing) operating systems have to guarantee inter-user protection, where different users may have different access rights (permissions) to the computing resources. An especially successful architecture for such systems was originally proposed by Dijkstra [Dij68]: The separation of the design into *layers*, where a layer is restricted to the abstractions and services provided by lower layers. For example, device specific code (often called *drivers*) was located in the lowest layers, protecting critical resources against misuse.

The layered architecture suggests that everything up to a certain layer belongs to the operating system. This leads to an implicit definition of what an operating system is, i.e. by exclusion of things that an application cannot do. More advanced systems allow *extensions of OS layers* to be installed, e.g. new device drivers. However, even if this is possible, such extensions are placed on the level of a certain layer and hence it is still clear whether a particular extension extends the OS or an application.

The layered design is easily supported by the hardware. Even if the language implementation is unsafe, or if assembler programming is permitted, it is possible to inform the hardware about the *protection level* a certain program is executing in. Then, memory areas containing code and data can be protected against access from within programs executing on the wrong protection level. It is common to support just two levels termed *user mode* and *supervisor mode*. More refined schemes support *protection rings*, i.e. multiple levels arranged under a total ordering.

The interrelation of layered software and hardware protection levels allows for another definition of the traditional operating system term: Everything

executing in a level of sufficient privilege belongs to the operating system. This is easy to understand but somewhat discomforting, as it relates the definition to certain hardware facilities.

1.2.2 Open Operating System Models

An operating system is a collection of things that don't fit into a language. There shouldn't be one.
(Daniel H. H. Ingalls [Ing81])

In an open system extensions should be possible on all levels. To enable extensions all interfaces of abstractions should be available, and all abstractions should be extensible. If this is carried through to the lowest levels, the distinction between OS services and other services gets blurred. Several alternatives follow:

A. In an open system, everything belongs to the OS – the notions of "application" and "operating system" vanish.

B. The OS shrinks to a minimal kernel, i.e. almost everything becomes "application".

C. A less strict and more pragmatic position is assumed and thus a blurred definition is accepted.

Obviously, alternative *A* is not very helpful. *B* follows the architecture oriented approach, but somehow predicts the OS to vanish in the future. Thus, *C* is followed to have a name for the things discussed. To arrive at a pragmatic characterization of "OS", several approaches can be tried.

First trial: The *OS manages all shared resources*, i.e. the ones used by more than one application. What precisely is a resource? If abstract services are included, this definition makes every shared piece of code part of the OS, including all libraries. If resource means hardware resource, then higher-level abstractions that traditionally belong to an OS, are excluded. For example, inter-program communication mechanisms are entirely abstract, yet traditionally belong to the OS.

Second trial (due to Josef Templ): The OS is what is there after booting the machine. This is perhaps the most unprecise definition, as it gets shaky as soon as startup configuration mechanisms exist. Then, everything that happens to be part of the current startup configuration is part of the OS.

Third trial: The OS provides a virtual machine view hiding away hardware details. This definition gets close to alternative *B* above. All that remains within the OS is the minimal software required to complete the basic hardware. Often, such an "OS" is termed *micro-kernel* [Gie90] and all that it contains are means to safely implement most OS functionality on the

application level. If the definition of operating systems is reduced to that of micro-kernels, the OS gets emptied out by pushing most problems into the application domain.

For the scope of this thesis the following simple compromise is pursued:

> *An operating system is a collection of application-independent abstractions and services that are of general use to applications.*

Of course, the term "of general use" is fuzzy and refers to the typical use of the complete system. For example, a math library that is part of the average installation and that is used by many applications, will be considered part of the OS.

As a general conclusion it may be added that the strict distinction between OS and application is not wanted in the setting of fully extensible systems. Introducing the term operating system anyway is helpful when a name is needed for "the part of the average installation of a system that supports most of its typical uses", as this is the part that creates the overall flavor of the system.

2 Object–Orientation in Operating Systems

The two primary notions isolated in the introduction are Extensible Object-Oriented programming and Operating System. In the following the combination of these two terms is motivated (2.1) and a short survey of other work is given (2.2).

For large systems the design should follow a small set of generally applicable design principles. This helps to avoid bundling a large number of ad-hoc solutions into a heterogeneous system. Instead, clearly arranged systems can be built that are easy to understand and reasonably easy to maintain once the underlying principles are understood. As the whole idea of bringing object-orientation into operating systems is an increase of flexibility and extensibility, the application of general design principles becomes even more important. For this thesis, a significant portion of the work went into the development of a set of sufficiently powerful design principles (2.3).

Following the general issues of design principles, more concrete details on how to structure an object-oriented operating system will be discussed (2.4). This covers two levels of system structure: Modularization and Type Hierarchy.

To conclude this chapter, specific operating system features required to support Extensible Object-Orientation are treated (2.5), and other, more traditional operating system features are considered (2.6).

2.1 Motivation and Goals

2.1.1 Two Orthogonal Possibilities

Object-orientation in operating systems can be considered for two rather distinct purposes. On the one hand, an operating system, like any other program, may *itself* be written in an object-oriented fashion (2.1.2). On the other hand, an operating system can be designed to smoothly *support* applications written in an object-oriented fashion. For an open operating system, i.e. one which can be extended and adopted to varying needs, the distinction between supporting object-orientation and being object-oriented becomes blurred (2.1.3). Finally, certain facilities are required to effectively support object-oriented applications (2.1.4).

2.1.2 Why Write a Program (an OS) in an EOO fashion?

The simplest reason for writing a program in an object-oriented fashion is the possibility to improve system structure. Of course, this is already possible by resorting to abstract data types (ADT). However, it is important to understand that the object-oriented approach is a superset of the ADT approach, in that a program implemented using ADT's can readily and automatically be converted into one using objects. The transformed program will contain a class definition for each ADT, but will not use any inheritance or subtyping relationships.

By adding a type hierarchy the various "ADT's" can be set into relation. This allows for uniform treatment of variants: By coding generic parts of the program to operate on a common base type, redundant coding can be avoided. In a certain sense subtyping replaces the need for variant types (sometimes called union types). However, the set of variants need not be known when coding the handling of the base type. New variants (subtypes) can be added freely and at any time.

Finally, if case specific operations need be executed, instead of coding a case switch which explicitly distinguishes between possible variants, bound procedures are used. The case analysis is then moved to the dynamic binding and the set of variants is left open.

The object-oriented style adds an important possibility. As the actual variants of a certain base type need never be fixed, it becomes possible to dynamically load and bind new variants at run-time. This is possible, if code is written in a way that avoids explicit type case analysis and instead resorts to late binding: The newly loaded implementation merely adds a new variant to the known base type and introduces objects of the new type. Old code continues to work, as it will refer to the known base type of the new objects, only. Dynamic binding then takes care of dispatching control to the implementation of the new type variant.

Subtyping leaves the set of variants (subtypes) of a base type open. Also, and equally important, it leaves the number of subvariant levels open: With every newly introduced type it becomes possible to define subtypes of it. Hence, having subvariants of variants is possible at any time.

The potential opened by allowing to add subtypes (and their implementations) at run-time is only explorable if programs can be split into separately compiled parts. This point is often underestimated. Therefore, in this thesis, it was decided to distinguish between object-orientation and extensible object-orientation. A program written in an extensible object-oriented fashion is never complete in the sense that extensions are always to be expected. Therefore, the EOO style has severe consequences for

the programming style adopted: In order to be safe, an EOO program has to draw strong boundaries between what may be changed by an extension and what has to stay untouched. As an EOO program can never be considered complete, resorting to global checking techniques to introduce safety as an afterthought is doomed to fail.

To conclude, using the extensible object-oriented style a system can be built in an evolutionary fashion: Starting with a small initial system more functionality is added when needed. Further functionality may even be added to an already running system without disruption of service.

2.1.3 Why should an OS Support EOO Programming?

For an operating system extensions are vital: Every application loaded and executed by a traditional OS is just an extension of the OS itself. Hence, it is a conceptually small step to consider extensions on all levels of an operating system. However, the traditional program oriented approach is not robust enough to allow for that. Hence, traditional OS's drew a strong safety barrier between their intrinsic functions and those implemented in an application program. For true extensions to be applied to inner parts of an OS, a clean and safe integration of extensions is required. The extensible object-oriented style is a way in this direction.

If it is accepted that it makes sense to structure an OS in an extensible object-oriented way, then the question follows whether or not an OS should also be enriched to *support* EOO programming. For a traditional operating system, this is indeed an option. (For example, several object-oriented operating systems exist that implement the UNIX kernel interface, thus not supporting EOO programming of OS clients.) For an extensible object-oriented operating system it is not an option! In fact, an extensible object-oriented OS *must* support EOO programming, as the extensions of the OS itself are supposed to be implemented in the EOO way. Since extensions of the OS should be executed under the OS the EOO support is indeed mandatory.

While the argumentation above is imprecise in a technical sense, the next section will clarify the situation by analyzing what it means to support EOO programming.

2.1.4 What means "Supporting EOO Programming" for an OS?

What follows is a brief investigation what supporting EOO programming means for an OS. Details of how to implement such supportive facilities are ignored (cf. 2.5).

Generic Object Manipulation. The idea of object-oriented programming is centered around the term *object*. An object is meant to encapsulate part of a system's state space and to provide a set of operations in order to manipulate its state. The use of objects becomes particularly powerful, if generic handling of objects of a common base type is possible. Most implementations of generic operations on objects are expected to be provided by the extensions. For example, if a base type *T* requires an operation *PrintOn*, then each implementation of a subtype of *T* is expected to provide a suitable implementation of *PrintOn*. Then, a generic piece of code may invoke *PrintOn* on some object referred to by a variable *x* of base type *T*. This causes invocation of the proper implementation associated with the actual type of the object referred to by *x*.

For some operations genericity requires system support, though. This is generally the case when the machine-dependent representation of objects is involved. Typical examples are the generic copying of objects, and the generic mapping (e.g. loading and storing) of objects to and from sequentialized representations (e.g. files).

While an object's implementation can take care of copying or mapping its components, it cannot do so for its class membership. To do so, an object needs run-time access to information about its own class. This can be achieved by adding certain *meta-programming* facilities to a system or a language: Facilities that allow a program to acquire and, in principle, to change information about its own execution environment.

The dilemma becomes apparent when considering loading an object from an external representation, say a file. Since the object is not yet available, its own implementation cannot help in doing the first step: Finding out what class implements the object. In a traditional system the variants to be expected on the file are known in advance and a simple coding scheme plus a case switch suffice. If the system is extensible this does no longer work. The variants to be expected *cannot* be covered by a central loading service. Instead, a more general service is required that can map class information (e.g. the class name) to a unique external representation, and that can take such a representation to retrieve the class again. Finally, it must be possible to create a new instance of the retrieved class.

Dynamic Loading. To make use of *extensible* object-orientation, it is crucial to be able to load extensions on demand. At first glance, this seems orthogonal to the generic object support discussed above. However, this is not so. The example of loading an object from a file is again helpful to understand this interrelation. Before internalizing the object it is necessary to read the external representation of the class implementing the object. If the class (it's implementation) is present in the system, a new instance of that class is created and the class controls the actual object loading. Things are more complicated if the class is not present. Then, the system has to try to *dynamically load and link* the implementation of the missing class. If dynamic loading is not supported, a system must be configured to have all class implementations available *before* trying to load an object from some file! This stands in sharp contrast to the requirement that the system should be extensible and grow to its needs.

Dynamic loading and system extensibility require separate compilation; otherwise the loadable extensions must have been compiled before running the system at all. For an operating system used to support program development the latter is unacceptable. Hence, the dynamic loading facility is likely to be based on the units of separate compilation, e.g. *modules.* (This will be discussed in more detail in section 2.5.2.)

Garbage Collection. If a system grows by means of dynamic loading of extensions it becomes hard to maintain central knowledge of all references that exist for a particular object. Even worse, unless the used programming language introduces severe restrictions on the passing and copying of references, it becomes *impossible* for the programmer to keep track of references in a safe way. Either an object has to be hidden from extensions, making its whole purpose of being an object questionable. Or the space allocated to the object can never be reclaimed without risking the existence of *dangling pointers,* i.e. references that survived the object deallocation.

A possibility is to explicitly overload the reference-creating operations (as may be done when using C++). Then, the risk of deallocating an object while references remain can be avoided, but instead the risk of having *memory leaks* is introduced, i.e. it becomes possible and often unavoidable that object deallocation is not triggered, although the object became unreachable. (One should note that an object becomes unreachable either if no reference to it remains, or if all remaining references are located in unreachable objects. It is well possible to have circular references among objects that jointly became unreachable.)

In other words, for a dynamically extensible system explicit deallocation *is not feasible.* Since finite memory resources force space to be reclaimed eventually, the only viable alternative is to introduce garbage collection, i.e. a mechanism to safely determine if an object became unreachable, and to

deallocate it thereafter. This point is very important. Often, garbage collection is considered a matter of *convenience*, relieving the programmer from the error-prone task of explicitly deallocating memory. While this is a true aspect, it is not the whole truth. Far more important is the *impossibility* for the programmer to solve this problem in a safe way when programming a dynamically extensible system.

> Garbage collection in turn is possible only in an environment with centralized resource management. For example, it must be clear where to look for references to objects, and it must be clear where to look for unreachable objects.

Finalization. Another aspect closely related to garbage collection is object finalization. Object finalization refers to the possibility for an object to take some final actions just before it gets removed by the garbage collector. As the time of removal cannot be determined by the programmer (it depends on the state of the extensible system and its interaction with the used garbage collection strategy), an object implementation cannot control the state of some external resource in a safe way. For example, an object may encapsulate the state of an external device, say one of several attached lasers. Then, before being removed from the system, the object needs to ensure certain safety conditions, e.g. by turning the laser off. Another example is an object representing the state of a file provided by some (stateful) remote file system. To release resources it is assumed that the object needs to close the remote file eventually. Again, as for the laser example, the object could provide an appropriate method for this purpose, but it cannot *guarantee* that the method gets invoked before the object becomes unreachable.

A third example deals with *identity directories.* Often it is useful to have some directory service return the *same* object repeatedly if the same search key is presented repeatedly. For example, to preserve space a font directory should return the same rastered font whenever the same name is presented. An identity directory implementation maintains the set of objects retrieved so far, and consults this set whenever a new request is made. The problem is that such a set interacts badly with garbage collection: Since the directory has no means to decide when an object in its set became otherwise unreachable, i.e. no references *besides* the one from the set structure remain, it cannot remove objects from the set. Therefore, the garbage collector will *never* collect such objects again. While this problem cannot be solved by means of finalization it is closely related to it: The solution requires an action (set exclusion) to be performed when all but one particular reference to an object vanished. This will become apparent in the section introducing the finalizing garbage collector facility (2.5.3).

Exception Handling. Exceptions are unexpected behaviors observed when invoking a certain operation. In most cases exceptions can (and should) be avoided. For example, instead of performing the operation

$x := y / z$

and therefore risking to raise a division-by-zero exception, a guard should be added:

if $z \neq 0 \rightarrow x := y / z$ [] $z = 0 \rightarrow$ ... fi

where the case $z=0$ is caught and dealt with separately. This is possible whenever the guard is feasible. However, this assumption may not hold in certain cases. For example, when using a matrix inversion operation to compute A^{-1}, the check whether $|A| \neq 0$ holds is almost as expensive to compute as is the operation A^{-1} itself. In such a case, instead of coding

if $|A| \neq 0 \rightarrow B := A^{-1}$ [] $|A| = 0 \rightarrow$ ... fi

it may be better to write

Invert(A, B, res); if res = 0 → skip [] res ≠ 0 → ... fi

where *res* is a result status of the operation Invert.

Hence, exceptions can be avoided in all cases considered so far. A system that discovers an exception anyway should therefore terminate the offending program, as the occurrence of an exception is considered a programming error. However, care must be taken when generalizing this result to extensible systems. What does it mean to prevent exception in the following case:

x.f

where x is a variable of some type T that defines a type-bound procedure f? As the actual class of the object referenced by x is statically unknown, the appropriate guard for this operation is unknown too. In principle, the definition of T should include the semantics defined for f. Then the caller can supply appropriate guards to guarantee preconditions of f and the implementation of f can take additional precautions and perhaps return some result status. Hence, it seems that the situation is the same as before, and exceptions that occur anyway should lead to program termination.

The rationale for forcing program termination upon exception detection was the concept that a program causing an exception is considered erroneous. This rigor implies that an extensible system is erroneous if *any* of its extensions are erroneous! For an extensible operating system, this means that the operating system is erroneous if any of its extensions, e.g. applications, is erroneous. This, of course, is intolerable, as the operating system should definitely not be terminated if an exception in one of the dynamically loaded extensions occurs. Instead, it should be possible to force the system back into a consistent state by taking control from the erroneous extension without leaving inconsistent states in other parts of the system. Hence, for an extensible system, some sort of exception handling is crucial.

2.2 Other work

Before going into deeper details, a short survey of other work may be in order. Since the thesis text contains references to the literature whenever some concept is introduced, this section will be kept short. Also, it is not expected to cover all existing systems that may somehow relate to this work. The systems that will be considered have been selected to cover a broad range of approaches and to isolate the main differences to approaches taken in the Ethos project. Two classes of systems are looked at: Systems that are object-oriented or support object-orientation, and systems that are extensible.

Cedar, Oberon. Cedar [Tei84] is the conceptual predecessor of the Oberon system [WG89][WG92] and thereby the Ethos system. Cedar has been developed at the Xerox Palo Alto Research Center (Xerox PARC), California. Its highlights are extensibility on an application level, a relatively high degree of inter-application integration using a common model of texts, a novel user interface based on tiled windows, and the use of the strongly typed language Cedar [Lam83] that evolved out of Mesa [MMS79]. However, the more operating system specific features, like support for multiple processes and inter-process communication are rather traditional, as is the type system of Cedar, which does not support true subtyping. Both led to an increase in complexity and a decrease in comprehensibility of the Cedar system.

Oberon in turn follows the most important ideas of the Cedar user interface, while introducing an entirely different programming model. While Cedar is based on multiple processes, each executing for some application, Oberon is a single-threaded system with no notion of applications. Instead, the whole Oberon system can be considered a single application that is extensible to a certain degree. Also, Oberon is based on the language Oberon [Wir88b][RW92], which fully supports subtyping. Ethos carries these ideas through, but largely increases the potential for extensibility. Ethos is implemented using the language Oberon-2 [MW91].

Smalltalk. The classical extensible object-oriented system. Building on the experience with fully integrated Lisp Systems, the Smalltalk designers dropped the separation of operating system and application [Gol83]. Instead, Smalltalk is based on a virtual machine predefining some primitives that in turn are used to construct a running, interactive system. The language Smalltalk is quite small and relies on a large standard class library [GR83]. The set of all class definitions and all objects coexisting in a system at a time is called the Smalltalk *image*. The system is programmed and used by successively extending the image. Extensions and modifications to all parts of the system can be made at any time.

The basic philosophy is that source code of the system is always available.

This is more important than it might appear. The system is large and rather complex. Everything is modeled using objects communicating by means of messages, but the language has no manifest types. Hence expressions have *no* static meaning, and almost all programming errors end in a run-time error of the kind "message not understood". Although it is considered good programming practice to use identifiers like *anInteger* to somehow add type information by convention, the system is hard to understand without browsing through its source code. Hence, having all source code at hand is a must for the Smalltalk programmer: It largely replaces documentation!

Other than Smalltalk, the Ethos project is based on the strongly typed and compiled language Oberon-2. Like Modula-2 and Oberon, Oberon-2 provides modules for system structuring. This was found helpful to avoid clutter; even more, an argument can be made that a useful language should provide both, modules and classes [Szy92b].

Choices. Choices is a research project at the University of Illinois at Urbana-Champaign [CRJ87]. From the outside Choices mimics the interfaces of various standard operating systems, e.g. MS/DOS or UNIX. However, the components of Choices are radically different in following object-oriented *frameworks* to achieve easily configurable and adaptable designs [CJM*89]. In this context, a framework is defined as the common design of (a family of) similar applications or subsystems [JF88]. If such a design can be expressed by means of classes, then specialization by means of subclassing can be used to derive a concrete design adapted to special needs. The framework that has been explored most thoroughly in the Choices project is that of the file system [MCR*89][Mad91]. The file system framework has been used successfully to implement a large variety of file systems on different machines.

A major difference between Choices and Ethos is that Choices looks like an ordinary OS, but is structured in an object-oriented fashion inside. Ethos, on the other hand does not try to look like any existing OS (except for resembling the Oberon system), and is structured in an object-oriented fashion from the outside. Several components of the Ethos system have not been structured in an object-oriented fashion, but instead fully encapsulated into single, closed modules. Using Choices-like frameworks to re-engineer such modules would be useful.

Choices has been implemented using C++, hence aiming at a performance comparable to Ethos and certainly superior to Smalltalk. Choices suffers significantly from the lack of run-time type information and garbage collection support in C++. As admitted by the designers of Choices [Cam92], the inability of changing the C++ compiler to support run-time type information was quite a hindrance. (The Choices project is partially funded by the industry, strong compatibility requirements – with the mistakes of the past – where thus mandatory.)

Clouds. The Clouds operating system resulted from a research project at Georgia Institute of Technology [DLA88]. Clouds is object-oriented and concentrates on the support for *reliable objects* in low levels of the system. The key idea is to provide automatic means for fault-tolerance and fault-recovery on the level of objects [Spa86].

Clouds objects tend to be large-grained, as each executes in its own address space. In this sense, a Clouds object is closer to a UNIX process than to Ethos objects. The Ethos system contains no provisions for fault-tolerance based on distribution. Extensions in this direction would be interesting to pursue, in which case the meta-programming mechanisms present in Ethos (3.4.3) are expected to be helpful.

Guide/Commandos. The Commandos system [MG89] emerged as part of a European ESPRIT project and is the result of a multi-institutional collaboration. In Commandos all objects reside within one or more address-spaces, called *domains.* Each Commandos domain may span multiple machines. Accessing an object on a remote machine causes the domain holding the requesting object to extend to the remote machine and include the requested object. This way object communication over machine boundaries is entirely transparent without requiring all objects to reside within a single world-wide address space.

Commandos exists in tight combination with the programming language Guide [KMN*90]. The tight coupling of system and language makes it comparable to systems like Smalltalk, Oberon, or Ethos. As mentioned above, Ethos does not yet cover aspects of distribution. The Commandos/Guide approach would be one possible way to add support for distributed architectures to Ethos, possibly running a complete Ethos system in each Commandos-like domain.

SOS. The SOS system [Sha89] started as an application and language independent layer supporting creation, deletion, migration, storage, localization, and invocation of objects. Due to the sufficiently general approach this layer can be seen as an object-oriented operating system. SOS has been developed at the Institut National de Recherche en Informatique et en Automatique (INRIA), France, as part of the European ESPRIT project SOMIW. SOS is written in C++ and prototyped on top of UNIX. The key concept of SOS are fragmented objects, i.e. objects residing on multiple machines at the same time. A fragmented object may internally be aware of the distribution to allow for optimizations, while from the outside (i.e. as seen from object clients), fragmentation is transparent.

SOS concentrates on the *support* for object-orientation. Implementations of SOS services are based on traditional (UNIX) systems and are not object-oriented, although this could be changed by re-implementing SOS for

bare machines. The fragmented object approach is another way that could be explored when adding support for distributed systems to Ethos.

Amoeba. Amoeba is a distributed operating system based on objects communicating using remote procedure calls (RPCs). Amoeba is the outcome of a research project started at the Vrije Universiteit Amsterdam, Netherlands [MT86], now spanning several other universities. A key idea in Amoeba is the location transparency of objects by means of system-provided name services and access via RPC. Protection against illegal invocations is achieved using encrypted *capabilities* [TMR86]. Amoeba itself is language independent and has been implemented to execute on bare machines or under host operating systems like UNIX.

Other than many of the systems described in this section, Amoeba has been used to support many users, and has proved to perform efficiently [RST89]. The interesting concept of hiding remote invocations by means of remote procedure calls and at the same time introducing protection by means of capabilities might be yet another way to add distribution support to Ethos.

Mach/NextStep. The Mach project [ABB*86][RBF*89] at Carnegie-Mellon University, Pittsburgh, Pennsylvania, aimed at minimizing the operating system subset that must stay in a protected kernel. The resulting *micro-kernel* basically contains address space and thread management, as well as inter-thread communication by message passing. Thereby the kernel is limited to the features required to establish protection barriers plus safe communication across such barriers. Otherwise, the Mach system uses user-level servers to provide for virtual memory (paging), file systems, network protocols, and the like.

NextStep is a user-interface framework and toolkit built on top of Mach [Web89] that is fully object-oriented and implemented using Objective-C [Cox87]. However, NextStep has its limits as it cannot fully hide the fact that the underlying UNIX-like (Mach-based) operating system is not object-oriented and not really geared to support object-orientation. For example, there is no concept of garbage collection on the level of the operating system – data structures shared by multiple processes are thus not collectable. Also, the Mach message mechanism is not transparently integrated, as it differs largely from the finer grained message mechanism used for the communication between Objective-C objects.

The Mach micro-kernel would be an interesting extension to the Ethos kernel, allowing for a more rigid protection mechanism. However, other protection mechanisms going in conjunction with support for distribution may be more adequate. Also, it is not clear that the Mach micro-kernel is indeed minimal (it still contains quite some functionality). The NextStep framework parallels the user-interface framework found in Ethos, but is

significantly richer. In principle, there is no reason why such a richer set of building blocks cannot be provided for Ethos in the future.

V Kernel. The V System has been developed in the course of a research project [Che84] at Stanford University, California. Similar to Mach, but oriented towards transparent distribution, the V kernel contains only a few standard features: A process abstraction with operations for interprocess communication, process management, and memory management. Other than Mach, V also contains access to raw devices, like disk drives. Higher-level abstractions, e.g. a UNIX-like file system, are then built as standard services running in user mode. Coupling *service modules* using the distribution-transparent communication facilities of the V kernel a distributed system can be formed.

The V project was particularly concerned with performance of remote operations; one of the interesting results is the development of the VMTP transport protocol [Che86]. These results could be useful when considering a kernel to be used for Ethos that masks aspects of distribution. However, it is not clear whether this approach of transparency at a very low level carries through to more demanding applications that need to be aware of actual locations to perform best.

Chorus. Chorus is a commercial object-oriented operating system based on a micro-kernel architecture [HAR*88][RAA*88]. The Chorus project aims at a foundation for distributed operating systems that fulfill practical performance needs. The micro-kernel, called Nucleus, integrates distributed processing and communication. On top of the Nucleus subsystem servers are installed. The subsystem Chorus/Mix implements UNIX System V, while other subsystems such as object-oriented ones are planned.

Plan 9. Plan 9 is an operating system project at AT&T Bell Labs, Murray Hill, New Jersey [PPT*90]. In a sense, Plan 9 is more like UNIX than UNIX: *everything* in Plan 9 is either a process or a file. The Plan 9 system itself is rather traditional in that it is based on a closed operating system running applications. However, the extreme overloading of the file concept leads to surprising application interfaces. For example, the Plan 9 window system – called Plan 8½ – uses a set of files (e.g. /dev/cons, /dev/mouse, /dev/bitblt, /dev/screen) for its interface [Pik91]. This approach causes all aspects of distribution to be concentrated in the abstract file system, making things like remote window servers very easy to realize. On the other hand, the concept of well-typed, signature-oriented interfaces is given up: All requests to a service, including all arguments and result values, are coded into byte streams. Unless one introduces an entirely new programming language that allows definition of stream patterns (Plan 9 uses C), there is no way to statically check correctness of interface related operations.

The structure of Plan 9 and its applications may be considered object-oriented in a basic sense: Objects are active (implemented as processes) and communicate by means of stream coded messages (implemented using generalized files). The abstract stream-based interface makes replacement of individual components (objects) easy. However, there is no notion of type, no notion of class, and especially none of type or class hierarchy. Also, Plan 9 objects tend to be applications, i.e. rather heavy weight, as the stream coded communication is too expensive to support a fine granularity on the object level. However, an interesting concept in Plan 9 are process-local name spaces. These could be a useful approach to generalize the Ethos configuration concept based on directory objects (2.3.3) in order to support multiple users.

PenPoint. PenPoint is a proprietary commercial operating system [CS91]. It is built using a rather traditional multitasking kernel. Consequentially, important low-level concepts, like memory management, multitasking support, and program loading are quite conventional. However, on top of these services PenPoint is object-oriented and extensible, where a centralized Class Manager is used to maintain a list of available classes. Classes, objects, and messages are *not* supported by means of language constructs, but as a system service, making PenPoint rather inefficient. Also, PenPoint does not include program development support. Instead, cross development under MS/DOS is expected.

A unique and interesting concept of PenPoint is its user interface, entitled *Notebook Metaphor.* The entire system is organized into a (typically small) set of notebooks. Each notebook contains pages, which again are used to organize all documents. Documents are composed of hierarchically nestable components, where each component may be controlled by a different application. The latter is termed "recursive live embedding of applications" and is similar to the concepts found in the Write editor [Szy92a], which is a part of Oberon and Ethos. The notebook analogy could serve as an alternative interface for Ethos in order to get rid of the more traditional organization using a file directory (that is visible to the user).

Vamos/Overview. Vamos is the result of a research project at the Swiss Federal Institute of Technology (ETH Zurich) [Pes89]. Vamos is not an object-oriented system, but develops some strength in its extension concept by allowing new services to be integrated dynamically. Especially the handling of orderly and of exceptional terminations of extensions has been solved cleanly. Also, Vamos is a multi-threaded system, providing and supporting multiple pre-emptive threads by design. The key concept of Vamos are *domains*, where a domain can be seen as an address space. A domain typically holds the combination of a data space associated with the data manipulating code. Domains and

threads are completely orthogonal Domains are fully static. (The combination of a domain and a thread forms something comparable to a UNIX process.)

The Vamos thread and domain concepts are mostly orthogonal to the Ethos system design. Adding these concepts to a re-engineered version of Ethos, perhaps using a micro-kernel would be interesting, as it would allow versions of Ethos executing multiple, potentially hostile threads.

As an extension to the Vamos kernel system, the window system Overview has been built [Wil89]. The Ethos system already contains a standard user-interface providing for multiple windows (viewers).

2.3 Design Principles

It has been stressed in previous sections and is repeated here: The design of a large system should be organized around a set of general design principles. Otherwise it is likely that the resulting design resembles a conglomerate of possibly inconsistent (or interferring) ad-hoc solutions. In the course of the Ethos project (Chapter 3) it was discovered that sometimes a sufficiently general principle may cover areas for which it originally was not intended. It is felt that this is just due to the general nature of an abstract principle: It helps during concept formation in a way probably leading to unifications of seemingly disjoint aspects.

This section looks at some important principles of design usable to build an extensible object-oriented system. The principles are general enough to avoid a tight coupling to operating system specific topics. However, if a principle was felt too specific for some application domain other than the design of an extensible operating system it has been omitted from this section.

The application of the discussed design principles will be covered in section 2.4.

2.3.1 Before Decomposing a System

When designing a system one typically proceeds by first stating the requirements, then listing applicable design principles, and finally applying these principles to formulate a design. The requirements for the system at hand, an extensible object-oriented operating system, have been stated in previous sections and will be refined in Chapter 3 for the case study Ethos.

The set of design principles has to be examined carefully. First of all, design principles have to be checked for applicability. This can be done by following through several and sufficiently different examples. However, the ultimate test

of validity and usefulness is the application of a design principle in a larger case study. All the principles discussed in the next sections have been applied to the design of the Ethos system. Hence, convincing examples for the various principles are postponed to Chapter 3.

Several known principles have been found useful and applicable (2.3.2). However, in the course of the Ethos project it became apparent that some new or generalized principles were also required (2.3.3).

Before considering individual design principles a common requirement can be stated: The application of principles should lead to *safe* designs. The term safe design is a little vague. In the context of an extensible operating system safety refers to protection of certain system-wide invariants. A typical example are storage invariants that may be subverted by malfunctioning components: Writing to a dangling pointer causes unforeseeable side effects.

In a more general setting: A safe design should be based on the notion of safe components. Then a new component using only a certain set of the existing components – and relying on the correctness of these – should not be able to break invariants established and guaranteed by the used components. Clearly, this is a requirement extending to the used realization tools. It definitely has effects on the set of allowable programming languages and adds certain requirements to the hardware specifications. Requirements imposed on the language have been dealt with in Section 1.1.4, additional items – for hardware and language support – are on a "wish-list" and follow in the conclusions (4.4). The technical details of safe designs based on safe components are covered in Section 2.4.1.

2.3.2 Known Principles

This section concentrates on known design principles that are of major significance for the theme of this thesis. To summarize, these are: Typing and Type Safety, Modularization and Separation of Concerns, Hierarchical Layering of Abstractions, and the Model/View Separation. The first three are fairly general, while the latter refers to a rather specific yet crucial problem area.

Typing and Type Safety.
A fundamental principle is the restriction of operation applications to arguments of proper type. For example, it should not be permissible to multiply two characters. To achieve such a guarantee requires the introduction of a *type system*, where a type basically defines a set of values, but may also imply certain semantics by means of its name. A system is typesafe if application of operations to illegal operands is eventually detected. In principle, the earlier such an error is detected, the better. Therefore, languages

with static type checking are preferable over systems which perform such checks at run-time. Static checking means checking by static program analysis, typically done during program compilation. In principle, only *decidable* properties (and decidable at an acceptable expense) can be checked in a static fashion. Hence, certain properties must be checked at run-time for a system to be typesafe. An example is the application of the division operator to a divisor of value zero. In a general setting it is not decidable whether a divisor in a particular program will ever have the value zero. Hence, a run-time check for divisions by zero must be applied.

A type checker may first *reconstruct* type information and then check it for consistency. This is done in many functional languages [FH88], where type information can be left out. Recently, it has been shown that the approach is also feasible for object-oriented languages [OPS92]. This has some advantages when defining general functions where the range and domain should not be overly restrictive. A major disadvantage of type reconstruction is the requirement that all source code must be available to deduce the most general still admissible types. Another disadvantage of this approach is the reduced readability of programs: since type *names* are missing, the implied semantics of the names are not present. Also, types covering the same value sets, say the integers, cannot be distinguished, even if their semantics are different and a combination using a certain operator should not be allowed. Finally, some type reconstruction methods (called system-level type checking) have the disadvantage that an extension can introduce type errors into previously existing code, which is clearly unacceptable.

To maximize static checkability a programming language should have manifest types, i.e. types that are explicitly declared. Also, all constants, variables, and operations should be explicitly associated with some type. Then type errors are easily detectable, the information required for type checking can be made strictly local, and type errors cannot affect previously accepted parts. The latter two conditions are essential prerequisites for separate compilation (1.1.2).

Often it is useful to specify a subtype relationship, i.e. to specify that some type denotes a subset of another one. A subtype is said to *conform* to its base type, if its value set is indeed a subset of the value set of its base type, and if the semantics implied by the subtype is a strict specialization of the semantics implied by the base type. The former can be checked in a static manner, while the latter would require a fully formal specification of the implied semantics, where the specialization property would possibly be undecidable.

For the distinction between types and classes, and the related distinction between subtyping and subclassing refer to Subsection 1.1.1. Problems with this distinction are likely to occur if the conceptual framework is a language that does not distinguish between type and class hierarchies, as is typically

done in the Simula-subtree of programming languages. In the context of design principles one should note that subclassing is an implementation technique, but not really a design principle, unless utmost code re-use is a primary design goal. Also, whether the used implementation language identifies subclassing with subtyping or not should not affect the design principle: A design should always be type and subtyping oriented to express proper concept relations (like *is-a*). Otherwise one may easily end up in insane subtype relationships. Or, in the framework of a language without manifest typing, say Smalltalk, one easily ends up in designs that *cannot* be typed in a static manner.

Modularization and Separation of Concerns.
One of the most important design principles is the *separation of concerns*. This issue has originally been discussed by Parnas [Par72], leading to the introduction of software *modules*. The principle requires a system to be structured into subsystems, each covering a separate concern. The individual subsystems should be orthogonal in the sense that it is always clear where to look for some functionality. The dependencies between subsystems should be minimal and follow some orderly policies, i.e. only inherent dependencies should be there.

Module inter-dependencies are expressed in terms of module couplings, which can be explicit or implicit. *Explicit coupling* requires a module to *export* parts of it, i.e. to provide an interface, which then can be used by another module to use the former module. Reducing the number of dependencies between modules means making module interfaces slim and general, hiding implementation details from module clients. *Implicit coupling* refers to common assumptions made in more than one module that are not manifest in an interface used by these modules. An example might be the shared assumption of the value of some constant. Implicit couplings should be avoided as they significantly hinder maintenance and updating. In the example, if the value of the "constant" is to be changed during a system modification, all occurrences of the implicitly assumed value have to be tracked down and changed, risking an inconsistent update.

If the functionality of a module is accessed only by means of its interface, and if the module's interface is general enough to hide most or all of the implementation details, then the module follows the refined modularization principle called *information hiding*. Hiding information to an extreme leads to a system of (almost) disjoint modules that can be changed and replaced individually without taking too much care of other modules in the same system. Furthermore, and perhaps even more important, if most details of a certain module are hidden, the module can establish and *guarantee invariants* for its internal functions and state. Since all modifications to the module's internal state can be guarded by means of access functions in the module

interface, client modules *cannot* invalidate such invariants, despite potential programming errors in such clients. This special aspect of information hiding is usually captured by the terms *encapsulation* or, more specific, *abstract data type*. These are useful goals to strive for. However, in the context of extensible systems the issues of information hiding and constructing abstract data types become far more complex, as the goals of hiding and making extensible tend to conflict with each other.

If a system is planned to be extended later, extensions should add new modules instead of modifying existing ones. This is a stability policy, as it reduces the risk of uncontrolled effects caused by an extension. Therefore, modules must be designed in a way that their interfaces are *complete*, i.e. that it is unlikely that the module contains unexported parts that actually need be imported by an upcoming extension. This makes the design of effective module interfaces a difficult engineering problem. One way to ease this problem is to look for slim sets of orthogonal functions to be provided by a module. If the module encapsulates a clearly specified service, this might be easy. It is far more difficult, if the set of functions intentionally is a selection out of a too large "complete" set. One way to attack this situation is to give up information hiding: If a module exports all its essential parts, regardless of implementation details and safety breaches, then it is simply not possible that an upcoming extension misses something in the module's interface.

Another possibility, perhaps the preferable one, is to design a module to export *extensible abstractions*. Instead of relying on the completeness of an abstraction provided by a module, a module provides some basic abstraction that can be extended by upcoming modules. When adding a new module, it is important to distinguish between introducing a new abstraction and deriving an extended abstraction based on an existing one in an existing module. As long as clients of the existing module should stay with the functionality provided by the existing module and only new clients (of the new module) can profit from the new module, there is no difference. However, if existing clients, necessarily unaware of the new module, should profit from the functionality of the new module, things become more complex and the difference shows.

For example, a module Files may provide a standard interface to files, say Create, Lookup, Read, Write, and Close. The provided abstraction File may well be organized as an abstract data type, and the module Files may contain an implementation of this abstraction based on a disk device. Then a later extension adds a new kind of file, say files based on a new kind of device, or files with an enhanced semantics, e.g. new operations for positioning in the file. Assume that the new files are based on the implementation of the old files, e.g. by adding a buffer mechanism to support positioning. Nevertheless, clients of the old Files module *cannot* use files created using the new module,

as they statically refer to operations defined in module Files.

Extensible abstractions are based on the notion of *binding operations to abstract entities* instead of binding operations to modules. The most primitive form of binding an operation to an entity is by means of *procedure variables*. In the example of Files above, if Create or Lookup bind appropriate procedures to variables *read*, *write*, and *close* of a file, then clients of the old Files module could properly deal with a file provided by the new module. Of course, an old client will not be able to use functionality that was not at all anticipated by the original interface: It will operate on the conceptual projection of the extended to the original abstraction.

Typically, an abstraction will be represented by an object of a certain type. For extensions to be possible, new information should be bindable to an existing type (not to objects of an existing type!). The appropriate concept is called type extension [Wir88a] and allows a new type to be based on an existing type. Then, reference variables of a base type may actually refer to objects of a derived type. In the Files example, this allows a new file to contain buffer information, although being used and passed around by clients of the original Files module.

The use of procedure variables to attach semantics to types is in a certain sense unsafe, at least as long as there are no means to export "read-only" variables. The unsafety comes from the fact that the procedure variables can be modified in any client module, possibly installing procedures inconsistent with the actual type, or inconsistent among each other. Accepting the usefulness of binding procedures to modules in the case of plain abstract data types, one might proceed and bind procedures to types (or classes) in the case of extensible abstractions. As a result the combination of an object's type and its semantics defined by the set of bound procedures is fixed. Such bound procedures are called *methods*. The implementation of a type together with its methods is called a *class*.

Hierarchical Layering of Abstractions.
Often a system design can be cleaned up by imposing a partial ordering on the provided abstractions. Then the abstractions can be grouped into layers such that no abstraction of a lower layer depends on abstractions of a higher layer. One of the first systems built using a layered structure was THE [Dij68]. It is useful to associate types with abstractions. Then a set of modules – containing type definitions and class implementations – forms a layer. The import relation among modules reflects the layering and a proper layering therefore implies an acyclic import graph.

If a set of modules resides in the same layer, cyclic import among these modules could be allowed. On the other hand all the modules in a cycle function only if synchronously available. Hence, one might argue that modules contained in an import cycle should better be merged into a single module to make the tight coupling explicit. However,

sometimes the sheer size of modules created this way may be a practical argument to have multiple modules and thereby cyclic imports anyway.

In conjunction with layering of abstractions a warning may be in order. Often it is useful to introduce a relatively fine-grained layering when studying a concept space. The ISO OSI (International Standards Organization / Open Systems Interconnect) *reference model* is a typical example for such a referential framework. When turning such a conceptual framework into the design for an implementation great care must be taken not to map every conceptual layer straight into a design layer. The resulting design is in danger of having far too many levels of implementations separated by interfaces that are likely to be overly general and costly. Again, the typical implementation of protocol stacks following the ISO OSI model is a good example for how not to do it. Instead, one should analyze the true needs for layering, i.e. by analyzing where extensions are likely to go in. Only these layers should be found in the resulting design, where each of these layers will typically cover several of the original concept layers.

However, another warning of the opposite kind is perhaps also in order. Enclosing too many conceptual layers into a single design layer may make extensions hard or impossible, often requiring large parts of the system to be redesigned when the need for an extension emerges.

The two opposite warnings stated for hierarchical layering indicate that a proper layering of a system design is, just as the definition of proper module interfaces, one of the hard engineering problems.

Separating Models and Views.

One of the major ideas behind the original Smalltalk class library is the *Model-View-Controller (MVC)* [KP88] design principle (often called MVC "paradigm"). MVC can be seen as a design principle, but is significantly more specialized in its application range than the before mentioned principles of typing and modularization.

The MVC design principle is based on a simple observation. In an interactive computing environment there are three fundamentally different components. Firstly, some components implement computational *models*. Examples are numerical objects, like vectors or matrices, symbolic objects, like formulas, graphics, or texts. Secondly, rather different components are used to visualize objects of the first kind, i.e. to display or print aspects of models. These are called *views*, and examples are visible representations of the examples given above. Thirdly, *controller* components handle user interaction by interpreting user commands. Controllers cause modifications of models or view-specific settings (e.g. the view point or a zoom factor). Modified models notify their controllers and views after changes took place such that controllers and views can stay up-to-date. Figure 2.1 illustrates the MVC components, where multiple controller / view pairs represent the same

model. Also, an external component ("other component") is shown that modifies the model without knowing anything about views.

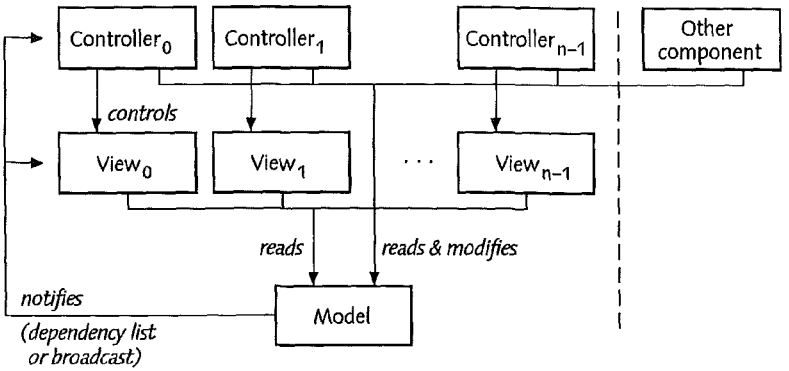


Figure 2.1 – The Model/View/Controller Separation.

Separating models and views decouples two rather different concerns, namely management and visualization of models. The result is a layering where views know about their concrete models, while models at most know that some abstract views may exist that visualize them. Hence, concrete views may be placed in layers above the corresponding model layers. The decoupling makes it easy to maintain multiple and even different views onto the same model.

A model needs to know little about existing views, but it needs to notify views if changes occured. There are two ways to achieve this coupling of models to views: Using broadcasts or using direct notification. In the former case the model merely needs to broadcast a message to *all* views that currently exist in the system. Hence, all that models need to know about is a single notification broadcast mechanism. In the case of direct notification a model maintains a list of (abstract) views that are supposed to visualize the model. This has performance advantages as only relevant views are notified. However, maintaining dependency lists introduces certain complexity to a system design. (In Ethos a single centralized broadcasting service exists.)

Separating controllers from views is in a sense more difficult, as controllers (input) are tightly coupled to their views (output): The view is affected by controller activities, but the controller gets control by activation of certain view features. For example, a controller may be used to scroll a view: The direct manipulation is provided by the controller, but the feedback is provided by the view. Often, it is a good choice to merge view and controller. In other cases a view may allow for installation of a controller component. (Examples for both possibilities are present in Ethos.)

2.3.3 New Principles / Generalizations of Known Principles

In the course of the Ethos project some new design principles emerged that are considered important results of the overall project.

2.3.3.1 Separating Carriers and Riders

For sufficiently complex data structures the separation of data access from data management is of interest. This is especially the case if auxiliary state is required to support the access mechanism. For example, a file system may provide multiple access paths to the same file. Then, the current position in the file – perhaps split into a sector number and a sector offset – is associated with each of the access paths. Another example is a complicated data structure, say a B-Tree, where iteration over the entries requires maintenance of a recursion state. This can be done by means of a recursive function performing up-calls for each element found. However, this kind of structure enumeration makes the combined traversal of multiple structures impractical. For example, traversing two B-Trees simultaneously to find all entries that share certain criteria becomes difficult. As for file access paths, special access objects, often called *cursors*, can be used to access and traverse the B–Tree. Of course, multiple cursors can be set onto the same B-Tree. A cursor can be used to read the current, next, or previous entry from a B-Tree based on a current position. To do so the cursor maintains the recursion state of the B-Tree traversal. As for files special care must be taken to specify the consistency semantics in the case of reader/writer and writer/writer conflicts. A possibility is the use of timestamps to detect updates of the structure. A cursor with an outdated timestamp may then be resynchronized upon its next use. This avoids maintaining a list of all cursors active on a certain structure.

The concepts of file access paths and cursors on data structures can be unified and generalized into the principle of *Carrier/Rider Separation* [Szy90b]. Carriers are the structures holding data, while riders are the structures used to provide (formatted) access. The crucial requirement is that carriers and riders are *independently* extensible and that multiple riders may simultaneously be connected to the same carrier. For example, a rider defined to access an abstract stream of bytes should be usable with every concrete stream of bytes, whether the stream is implemented as a file, a network connection, or a memory-based data structure. Likewise, an abstract stream rider may be specialized into various riders performing different transformations on the accessed data. Finally, operations exist that take an abstract rider parameter set to a corresponding carrier. (For an example, cf. 3.4.5.2.) Therefore, a *Cartesian product* of carrier and rider variants has to be supported, where the

lists of possible carriers and riders are both open to extensions.

The key idea is to define a *bottleneck interface* to interconnect carriers and riders. The bottleneck is particularly hard to design: As it is shared by two separate extension hierarchies it is *not* extensible. It is helpful to consider the stream example given above. Adding a new operation to the bottleneck interface, say InvertByte to invert every bit of a byte in-place (something strange enough that it is likely not present in the original bottleneck interface), would require a new carrier and a new rider base class. Hence, the new functionality could neither be used by existing stream carriers nor by existing stream riders. In other words, modifying the bottleneck interface always affects carrier *and* rider extension hierarchies.

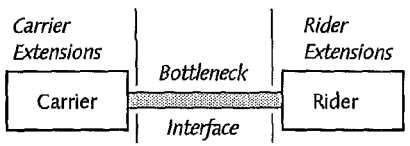


Figure 2.2 – The Carrier/Rider Bottleneck Interface.

The concrete selection of operations for a bottleneck interface depends significantly on the supported carriers and riders. For the example of byte streams it consists of two primitive operations for reading and writing bytes, respectively. (For optimization purposes, it may be enriched by adding two operations that read or write entire blocks of bytes.)

Additionally, an operation is needed that connects a rider to a carrier. The interface of the rider typically adds further operations to support reading and writing other data types. Different rider implementations may then apply different mappings of the rider supported types to and from byte streams supported by the bottleneck interface. Also, a rider may contain auxiliary fields that are used by the carrier to reflect the status of the most recent operation.

Below, the simplified interface of a stream/stream rider pair is given. (The distinction of records and pointers is not important at this point. However, it needs to be considered carefully when designing a concrete system.)

```
TYPE
   Stream = POINTER TO RECORD
      PROCEDURE (S: Stream) Set (R: StreamRider; pos: INTEGER);
   (*the bottleneck interface*)
      PROCEDURE (S: Stream) Read (R: StreamRider; VAR x: BYTE);
      PROCEDURE (S: Stream) Write (R: StreamRider; x: BYTE);
   END;
```

```
StreamRider = RECORD
   eos: BOOLEAN;   (*end of stream reached*)
   res: INTEGER;   (*res ≠ 0 ⇒ unsuccessful completion of most recent operation*)
   PROCEDURE (VAR R: StreamRider) Read (VAR ch: CHAR);
   PROCEDURE (VAR R: StreamRider) ReadInt (VAR n: INTEGER);
   PROCEDURE (VAR R: StreamRider) Write (ch: CHAR);
   PROCEDURE (VAR R: StreamRider) WriteInt (n: INTEGER);
END;
```

The next aspect to consider is the state space of a rider. It can be split into the product of two separate state spaces, one defined by the carrier and one by the rider itself. For example, a stream formatting rider may maintain some statistics on the data processed so far. These statistical data belong to the rider defined state space. On the other hand, the stream may associate some positional hint information with the rider, which then is part of the carrier defined state space. Obviously, the rider cannot take care of the carrier defined, but rider associated state, as this would significantly hinder extension of carriers. On the other hand, the carrier should not directly maintain such per-rider information, as it would require the carrier to maintain a list of attached riders. This is undesirable for three reasons. Firstly, it reduces the efficiency of the carrier implementation, as the rider-specific information has to be retrieved from the list indexed by the rider pointer, instead of directly from the rider. Secondly, the rider can no longer be allocated statically, as it must be linkable into carrier lists. Thirdly, the rider can no longer be collected by the garbage collector, without requiring clients to explicitly inform carriers to close a rider, i.e. to remove it from their lists.

This dilemma can be solved by introducing a moderating object, called *link*, implementing part of the bottleneck interface. To do so the bottleneck interface is split into operations that depend on carrier defined rider state, and others that do not. Usually, almost all bottleneck operations belong to the first category. (An important and necessary exception are operations to attach new riders to a carrier.) Then all rider state dependent operations are implemented by a link, while the remaining operations are implemented by the carrier itself. When attaching a rider to a carrier, the carrier creates a suitable link object. The carrier connects the link object to the rider which then may start to invoke bottleneck operations implemented by the link. Since the link implementation is fully under control of the used carrier, it does not hinder carrier extensions.

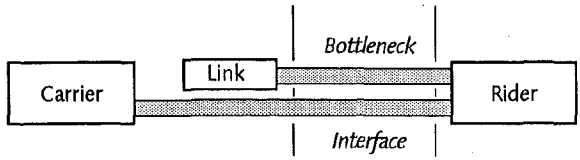


Figure 2.3 – Refined Bottleneck Interface using Link Objects.

The effect of the introduction of links may be seen when re-considering the simple stream example.

```
TYPE
   Stream = POINTER TO RECORD
      PROCEDURE (S: Stream) Set (R: Rider; pos: INTEGER);
   END;

   StreamLink = POINTER TO RECORD   (*the bottleneck interface*)
      PROCEDURE (L: StreamLink) Read (R: Rider; VAR x: BYTE);
      PROCEDURE (L: StreamLink) Write (R: Rider; x: BYTE);
   END;

   StreamRider = RECORD
      base–: Stream;
      link–: StreamLink;
      eos: BOOLEAN;
      res: INTEGER;
      PROCEDURE (VAR R: StreamRider) Connect (base: Stream; link: StreamLink);
      PROCEDURE (VAR R: StreamRider) Read (VAR ch: CHAR);
      PROCEDURE (VAR R: StreamRider) ReadInt (VAR n: INTEGER);
      PROCEDURE (VAR R: StreamRider) Write (ch: CHAR);
      PROCEDURE (VAR R: StreamRider) WriteInt (n: INTEGER);
   END;
```

The pointer connecting the rider to its link is part of the rider interface. This is important since rider extensions need to call the bottleneck operations implemented by the link. The pointer connecting the link to its carrier is kept private, though. To do so is possible since the link is created by the carrier and therefore the link and carrier implementations can be coupled tightly. A private carrier/link connection increases safety, as it is not possible to connect a link to a mismatching carrier. Otherwise an inconsistent bottleneck interface – composed of a mismatched carrier/link pair – could result. Since the rider cannot contain carrier (or link) specific state – except for the feedback information stored in some of its public fields –, there is no such safety problem for the rider/link connection.

The steps happening when setting a stream rider to a stream are:

```
VAR r: MyStreamRider; s: MyStream; x: INTEGER;
s.Set(r, 42);
   VAR u: MyStreamLink;
   create link object u, or re-use r.link if (r.link ≠ NIL) ∧ (r.link IS MyStreamLink);
   r.Connect(s, u)
      if necessary, adapt to change of stream;
      call method in super class
         r.base := s; r.link := u
r.ReadInt(x);
   VAR lo, hi: CHAR;
   r.link.Read(lo); r.link.Read(hi);   (*may set r.eos*)
   x := INT(hi)*256 + ORD(lo)   (*assumes signed conversion INT: CHAR → SHORTINT*)
IF ~r.eos THEN ... x ... END
```

Figure 2.4 shows the minimal pointer relations, and Figure 2.5 the duties of carriers, links, and riders. To make things more clear the situation is shown for two potentially different riders attached to the same carrier. The grey areas in Figure 2.5 correspond to possibly distinct modules implementing the components within each area, i.e. carriers and links are always implemented together.

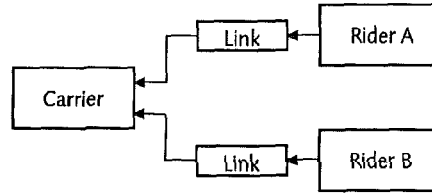


Figure 2.4 – Referential Relations of Carriers, Links, and Riders.

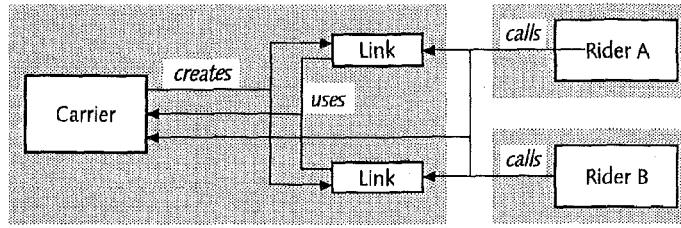


Figure 2.5 – Functional Relations of Carriers, Links, and Riders.

The design of Ethos contains many applications of the Carrier/Rider Separation, as will be discussed in Chapter 3. For the time being it should suffice to note that the range of applications is surprisingly large. The examples cover model spaces with a single dimension (stream devices, files: 3.4.5, texts: 3.4.6), spaces with two dimensions (raster devices, pixelmaps: 3.4.7.1), and abstract spaces (task spaces: 3.4.4). Further examples covering higher dimensions (e.g. voxelmaps to handle volume-pixel models), or other abstract spaces (data structures) are easily derivable.

The Carrier/Rider Separation may be compared to the Model/View Separation. Both separations handle 1-to-N relations, and both support independent extensibility of the separated components. However, Model/View does not expect a view to carry model-defined state information. In fact, it is even assumed that models are (almost) unaware of the existence of views. Thus, the Carrier/Rider principle may be considered a generalization of the Model/View principle. The generalization is mainly expressed by the support of carrier specific rider-state, which motivates the introduction of link objects.

2.3.3.2 *Directory Objects.*

The discussion of separation principles like Model/View or Carrier/Rider leads to a more fundamental issue: That of object creation. Separation principles are mainly motivated by the wish to support independently extensible but combinable type hierarchies. However, a hierarchy of types produces a number of choices among which one has to select when an object of a certain type is needed: In principle, an instance of any subtype of the required type will do. Obviously, there must be *some* place in the system where for every such situation the choice is made what type to use.

The most basic approach is *service-level object creation.* Here, a service directly creates and initializes the objects it provides services for. For example, a file system may offer a function New that returns new file objects. The major disadvantage of this approach is its complete lack of extensibility. Since the concrete type used to instantiate objects is burned into the service, there is no way to have it work on objects that are instances of a subtype of the original type.

The situation may be improved using *client-level object creation.* Instead of calling a service-provided New function, a client creates an object of the required subtype itself. The service provides proper initialization operations such that the newly created object can be supported.

For some programming languages, such initialization operations are automatically executed upon object creation. Using an object-oriented approach, the initialization operations are methods of the created object, perhaps to be called explicitly after object creation. This allows redefinition of initialization code in subclasses and avoids the problem of erroneously initializing an object of a subclass using an initialization

operation defined for a base class.

When using client-level object creation, the types/classes used to instantiate objects are burned into client instead of service code. This makes it easy to add new clients that are based on extended services. However, it still blocks the extension of existing *clients*. A third approach called *prototype cloning* can be used to add another degree of freedom: Client code uses a generic object cloning facility to get a copy of some object of the wished type. The source object is called a *prototype*, in the sense that it shows *prototypical* behavior: It is of some exchangable subtype/subclass and pre-initialized to accommodate exchangeable requirements.

The use of prototype cloning leads to significant flexibility but still exposes a major problem: The client code needs to know which prototype object to use for cloning. Typically, this choice depends on certain parameters, and the mapping of parameter values to prototype objects should not be burned into client code. For example, a file system that implements files as objects would have to provide a prototype object for files. A client wishing to open a file then needs to clone the prototype and perform an open operation on the newly created file object. If there are several file systems implementing different file objects, the choice which prototype to use may depend on the file's path name. However, the mapping of file name parts to file prototypes should not be redone in every client, as it makes it impossible to add a new file system, or a new naming convention.

All the object creation problems considered so far are caused by too early bindings of creation information to creation locus. All these problems can be avoided by following a new design principle: The use of *Directory Objects*. A directory object provides operations to create objects of a certain type. These operations often take parameters that further specify the kind of object required. For example, a directory object of a file system creates file objects based on a file path name. Application code only knows about a variable which holds a directory object creating objects of the wished type.

Even while the system is executing, a new directory object may be installed. All forthcoming requests directed to the variable where the directory object got installed will then be serviced by the new directory object. As a result, existing code will be able to utilize newly installed extensions immediately without any needs for recompilation. A directory object corresponding to the file system example is illustrated below. (Note that an abstract file is just a positionable stream.)

```
TYPE
   FileDirectory = POINTER TO FileDirectoryDesc;
   FileDirectoryDesc = RECORD
      PROCEDURE (F: FileDirectory) Old (name: ARRAY OF CHAR): Stream;
      PROCEDURE (F: FileDirectory) New (name: ARRAY OF CHAR): Stream;
   END;

VAR
   fileDir: FileDirectory;
```

Each service category provided in a system has a corresponding directory object. Extensions that refine an existing service implement a directory object that makes the extensions accessible. A *system configuration* then consists of the current assignment of directory objects to directory variables. Often, an extended directory object does not fully replace a previously existing one, but handles only certain requests while forwarding others to a default directory object. Then, the system configuration also contains the *default relations*, i.e. the assignment of some directory objects as defaults for others. (It is important that new directory objects for a certain abstraction – perhaps as part of other directory objects – can be introduced freely: The global directory object is merely an anchor for configuration purposes, while different applications may indeed use different directory objects for the same abstraction.)

> The Trestle Window System makes use of so-called *oracles* to adapt high-level abstractions to screen-dependent resources [MN91]. In a sense, oracles are special directory objects.

Consequences for the Extension Model. The use of directory objects has radical consequences for the extension model of the system. If it is assumed that a concrete class is implemented as a subclass of another concrete class, then the natural extension model is based on inheriting methods from the base class, and selectively overriding some of these methods in the subclass. Then, the overriding methods can refer to the implementation of the base class by calling overridden methods.

In this case it is *statically* determined from which class a subclass is derived. Sometimes, it is preferable that an instance of a subclass forwards unresolved requests more *dynamically*. In a general setting, requests should be directed to another object that is an instance of (another subclass of) the base class. Usually, the object to forward to is created by means of a request to a directory object associated with the base class. Thereby the system configuration can *dynamically* affect to which class unresolved operations are forwarded. In other words, *the subclass hierarchy, other than the subtype hierarchy, is not static but subject to run-time configuration.*

While it seems a bit complicated, strictly following this rule leads to a uniform and powerful extension model. Its applicability will be demonstrated in Chapter 3. In more detail, the extension model is explained in the next section. Also in 2.4.3, a natural and clean way to structure a system using this design principle and extension model is introduced.

2.4 System Structuring

2.4.1 Primary System Structure Given by a Module Hierarchy

Subsection 2.3.2 motivated the importance of modularizing a system and summarized the most important aspects of modularization. This section picks three essential criteria – module coupling, safety of modules and interfaces, and modularization criteria – and adds more technical details relevant to the discussion of Ethos in Chapter 3.

Module Coupling.
The notion of module coupling and the distinction between implicit and explicit coupling have been introduced above (2.3.2). Module coupling may be understood as the sharing of assumptions and conventions across module boundaries. If coupling is implicit, shared assumptions and conventions are not expressed in the module interfaces.

A general design rule is to reduce module coupling to a minimum while at the same time making it as explicit as possible. The former maximizes the independence of the modules, improving the comprehensibility of the modules as well as of the system composed thereof. The latter aims at capturing all interdependencies of a set of modules in a static fashion: If this is done successfully, a module modification affecting another module is statically detectable, e.g. by a compiler. Both design rules interact with the task of decomposing a system into modules. A good decomposition following both design rules is difficult to achieve, especially if the system is to be extensible.

The used programming language should contain a module concept that allows to define export relations, i.e. to distinguish among exported and private parts of a module. Further, a language may allow to classify exports, e.g. to mark an exported variable to be *read-only* outside of the module, or to mark an exported type or class to be *non-extensible* outside of the module. Modules with defined export relations may be imported by other modules, e.g. by listing imported modules in an importing module.

Module coupling is fully explicit if all assumptions about an imported module *M* are manifest in the export relation of *M*. For example, if the module exports a variable of a certain type, then it should be valid to set the variable to any value allowed by the type, and likewise one must expect to find any such value in the variable. If the value needs to be constrained, i.e. a system invariant should be maintained, the variable should be exported at most in a read-only fashion, while modifications should be controlled by an exported procedure.

Often, implicit coupling is implemented by means of unsafe features of the used programming language. For example, it may be possible to modify variables that are not exported by directly accessing memory locations. This may extend to the parallel definition of entire data structures within separate modules. In general, such techniques should be avoided. By introducing a new module implicit couplings can always be removed. By declaring the new module to be restricted, i.e. by not providing its interface for general use, the safety of the original modules is not affected.

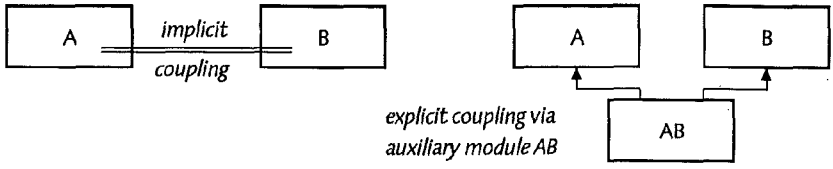


Figure 2.6 – Using an Auxiliary Module to Remove Implicit Couplings.

It should be noted that the introduction of an auxiliary module may not only make a coupling explicit but may also make it less efficient. While this may be considered a deficiency of the used language, it holds for most languages currently available and is especially true if modules are used as units of separate compilation without supporting inter-module optimizations. Hence, at limited but critical places in a concrete system, the controlled use of implicit couplings may be acceptable. (For example, certain assumptions about the storage layout may be shared implicitly among the implementations of the garbage collector and the module loader.) In any case, implicit couplings require great care and sufficient documentation, especially as they are outside of the "self documenting" part of the used programming language.

Safe Modules vs. Safe Module Interfaces.

A quality measure for the modular decomposition of a system is the *safety* of the involved module interfaces. The idea of having safe module interfaces relates to Cardelli's definition of safe modules [Car89].

In the following, only problems of safe modules and safe module interfaces expressed in the language Oberon (or Oberon-2) are considered. Before analyzing the concept of safe module interfaces it is important to understand that Oberon can be split into a safe and an unsafe sub-language. By design of the language, all unsafe features have been moved to a pseudo-module SYSTEM that is known to the compiler. Therefore, a module using one of the unsafe features needs to import SYSTEM. A module using the unsafe part of the language is considered *unsafe*. A module using unsafe parts of the language in its interface also has an *unsafe interface*. A module importing a module with an unsafe interface is itself unsafe.

The designer of a module may also *declare* a module interface to be unsafe. The latter form is a convention that is not part of the language Oberon-2. For

example, a module may be declared to be a private part of a subsystem that ought not be used outside, e.g. a disk driver module that allows to initialize a system's disk. Although the interface of this module is safe in a formal sense, it is not so within the context of the system. (The file system implemented on top of the disk driver module does not expect the disk to be initialized while being used as a file store.)

On the other hand, an unsafe module may well have a *safe interface.* This is the typical case when a module maps a low-level device to a standard abstraction, say a network link to an abstract stream of bytes. The idea behind safe interfaces is that when the corresponding implementation is *correct, no safe client module* can possibly invalidate system invariants. (Of course, a module with a safe interface but an unsafe implementation may still be incorrect; then no guarantees can be made.)

To understand the implications of interface safety, the Oberon file system may serve as an example. There, module *Files* exports a block read operation called *Files.ReadBytes*:

```
Files.ReadBytes (VAR r: Files.Rider; VAR x: ARRAY OF SYSTEM.BYTE; n: LONGINT);
```

The language allows the actual parameter substituted for a formal reference parameter of type ARRAY OF SYSTEM.BYTE to be of *any type.* An early language revision of Oberon moved type BYTE to module SYSTEM. This forces the interface of module Files to import SYSTEM, but the clients of Files *need not import SYSTEM* in order to use ReadBytes! However, passing a pointer to ReadBytes allows *any* module to violate an important system invariant, i.e. that pointers always point to a valid object in the heap or be NIL. For example, the code fragment

```
VAR p: POINTER TO Something;
Files.ReadBytes(r, p, 4)
```

over-writes the pointer variable *p* with an arbitrary value. The consequences are disastrous if *p* in turn does not point to a valid object in the heap and the garbage collector is run. Thus, the interface of Files, a first class module of the Oberon system, must be considered *unsafe*, making *all* implementations of clients of Files unsafe, too!

It is crucial to distinguish between an *unsafe module* which *could* be harmful, and a *incorrect module* which indeed *is* harmful if it is also unsafe. The point is that a wrong but safe module cannot be harmful to other modules. Hence, trying to have almost no unsafe modules in a system is a significant contribution to a systems overall robustness.

There are two possible solutions to the problem of Files.ReadBytes. On the one hand, one might change the language to remove the special compatibility clause for reference parameters of type ARRAY OF SYSTEM.BYTE. On the other hand, one might strictly refrain from using such a parameter declaration in the interface of modules that are supposed to have safe interfaces.

The former approach is cleaner, but leads to a language revision. The latter has been taken in Ethos. For example, block access procedures in Ethos have reference parameters of type ARRAY OF CHAR. This is less intuitive, as it forces every byte to be interpreted as a character, but is safe. (For a client to perform an unsafe mapping of some data structure to an array of characters, the *client* has to import SYSTEM and use SYSTEM.VAL to *explicitly* cause the unsafe type cast. This makes the unsafe implementation of the *client* explicit.)

As a late addition, the standard Oberon compiler has been changed to emit a warning if a module makes use of the unsafe compatibility clause for reference parameters.

What belongs into a single module?

Generally, definitions that together form an abstraction belong into a single module. In particular, definitions standing in a cyclic relation should be kept within a single module. Otherwise, cyclic module imports result, making the set of modules that contains the cycle practically unseparable.

On the other hand, it is important to achieve a certain balance: The size and complexity of the involved modules should be of similar magnitude. Both, having many almost empty modules and having seriously over-stuffed modules invalidates much of the structuring power of modules. However, achieving a proper balance is a subtle design issue. Often, it is a good test to check if the main meaning of a module can be explained in a single or at most a few sentences. This test may equally well be applied to other levels of structuring, e.g. to constants, types, variables, and procedures. However, the larger the structural unit – modules being the largest –, the more complicated is the successful passing of this simple test.

An interesting question is whether a module can ensure invariants spanning instances of multiple exported types. Even more, the question is if this can be done in the presence of extensions of these types. A thorough investigation of this problem may be found in [Szy92b]. The key point is the *read-only* access restriction a module may impose on exported types in Oberon-2. As explained in [Szy92b], this allows to hardwire certain interrelations into the defining module. For example, a module *M* exports two types List and ListMember where instances of ListMember are supposed to invariantly belong to *at most one* instance of type List. To increase readability and efficiency of client code, it is assumed that a pointer field *next* of ListMember is exported by *M*. Even when extending List or ListMember, this invariant can be maintained by *M* if the pointer field is exported in a read-only fashion. A safe client module of *M* has to use the (normal or type-bound) procedures provided by *M* in order to manipulate these pointers. Hence, there is no way for a *safe* module to break the invariant guaranteed by *M* for lists and list members.

2.4.2 The Type Hierarchy: A Secondary System Structure

In Section 2.3.2 the importance of modularization and typing has been stressed. It is interesting to consider the dual role on system structure that these two principles have. The module hierarchy forms the outermost structure of the system, but the actual decomposition of a system into modules is often driven by the type hierarchy which essentially constitutes the hierarchy of abstractions. A useful modularization scheme requires every type to be defined in exactly one module. Also a module that defines a subtype needs to define the corresponding base type, or import the module that contains this definition. Hence, the design of the type hierarchy directly affects the module hierarchy, and vice versa. A module is a subsystem potentially containing multiple interrelated types and classes.

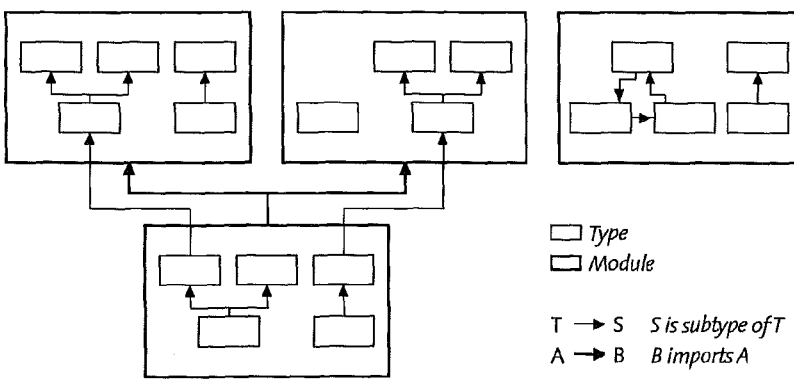


Figure 2.7 – Primary and Secondary Structuring using Modules and Types.

The structural similarity or type and module graphs led to many misconceptions in earlier language designs. For example, the role of modules was loaded onto the type scheme, or modules were treated as instances of special module types. In Modula-2 [Wir82] and even more in Modula-90 [Ode89], modules were identified with abstract data types. In Simula [DMN68], Eiffel [Mey92] and Sather [Omo91] classes are also used for modularization. All these misconceptions led to complications of the type or class constructs and to less clean language designs. (For a clear distinction of modules and types/classes cf. [Szy92b]).

Sometimes the type graph cannot be strictly hierarchical, i.e. cyclic dependencies cannot always be avoided. If this is the case, the types belonging to such a cycle are strongly interrelated and therefore should belong to the same module. In general, it is considered good engineering

practice to avoid cycles on the module level. Indeed, the language Oberon forbids cyclic import.

Based on the type hierarchy a module hierarchy may be derived by clustering all essentials belonging to an important type into a single module. A type is considered important if it is useful in its own right; if it can be used only in conjunction with some other type, the two types should be packaged into the same module.

2.4.3 The Canonical Module Structure

Often a module, say *M*, defines a single new *abstraction* by introducing a new type *T*. Further, *T* may be a subtype of another type, say *T0*. Finally, various *implementations* of *T* may exist or will exist in the course of system extension. To capture this relation between abstractions and implementations it is worthwhile to distinguish subtyping and subclassing (cf. 1.1.1). Figure 2.8 illustrates a possible constellation of types and classes. (In the figure, subclassing follows the direction of subtyping. In principle however, subclassing may also go in the opposite direction of subtyping.)

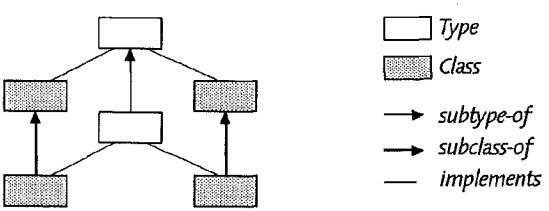


Figure 2.8 – Types and Classes in Separate Hierarchies.

In a language identifying subtyping with subclassing (e.g. Oberon-2), the separation of classes from types can be approximated quite intuitively by requiring classes to be leaf-nodes of the type hierarchy. In other words, all inner nodes of the type hierarchy are (semi-)abstract classes, cf. Figure 2.9. Then, abstract classes are extended to concrete classes, providing alternative implementations. A formal type system based on this use of abstract classes is given in [Dod92]. An abstract class may be made semi-abstract by providing some of the method implementations. The important point is that *(semi-)abstract classes must not be instantiated*, but merely serve to express subtype and subclass relationships. This should be supported by the language: It should be possible to express that a certain class is meant to be abstract, i.e. that creation of direct instances of that class is a programming error.

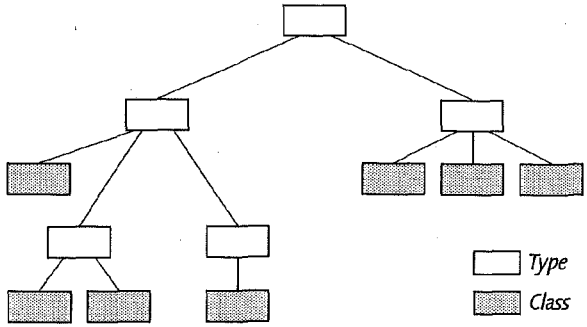


Figure 2.9 – Types and Classes in a Single Hierarchy.

Following this strategy reduces the potential for *code re-use* by means of code inheritance, which is often cited as one of the most important benefits of object-oriented programming. By its very nature, the code inherited from an abstract class tends to be almost empty. It may serve to establish certain abstraction-related defaults, but it usually does not contain significant contributions to the implementation of a subclass. However, a subclass hierarchy supporting code inheritance is less important than one might think at the first glance. The following simple calculation illustrates this point.

Let $|c|$ indicate the code size of a class c, including inherited code. Let two classes C and C' be of a form that allows C' to be expressed as a subclass of C in a way that the coding effort of C' gets almost negligible. Then the code reused when introducing C' is in the order of $|C|$. Let C and C' implement a type T. Next, let T have k clients, i.e. k different components that make use of the interface provided by T. Each of these components has code size $|x_k|$. If C' is introduced then all clients of T can potentially make use of C'. *If* the clients of T are coded in a way that they can be configured to use any implementation of T without being modified, *then* the effective code re-use is in the order of $k\,|x_k|$.

Clearly, for extensions introduced as quick fixes subclassing is useful: It re-uses the code of the base class, and the number of clients using the fix is expected to be small (as it is a fix, not a first-class extension). However, if the number of clients is significant and perhaps uncontrollable, which is normally the case for an inherently extensible system, the re-used client code easily outweighs the re-used base class code. For a language that does not support separate type and class constructs, it may well be preferable to emphasize the type hierarchy (as outlined above, cf. Figure 2.9), instead of risking a hardly documentable mix of type and class hierarchies (by emulating Figure 2.8).

To make clients largely independent of the class used to implement a certain type, clients should not directly create objects, e.g. should not call NEW. Instead, each major type should have a directory object (cf. 2.3.3) associated with it. By exchanging directory objects associated with certain types, existing client code can be re-configured to use different implementations (classes) of these types.

An important objection might be that it is sometimes not possible (or at least not feasible) to re-implement a class in order to change some details. This is the case if the base class contains proprietary and intricate code. However, in such a case it is usually possible to use forwarding: If an object of the new class cannot handle a certain request it forwards the request to an object it contains. The helping object is typically an instance of the class to be extended. This way there is no need to have direct access to the class requests are forwarded to. An important benefit is the gained flexibility: The new class can base its behavior on any object conforming to the required type. Hence, it is possible to compose various extensions on an object by object basis. (In the case of code inheritance this would require dynamic binding of superclasses.)

The ideas developed above lead to a uniform module structure, where a typical module

- exports a new abstraction of type *T* that is perhaps a subtype of an existing abstraction,
- contains a *non-exported* default implementation *C(T)* of that abstraction,
- exports the type of associated directory objects,
- contains a *non-exported* default directory object implementation,
- exports a variable holding a directory object returning (and/or retrieving) objects of type *T*, and
- exports a variable holding a standard directory object returning (and/or retrieving) objects of the default implementation.

Figure 2.10 illustrates the canonical module structure. The standard directory is held by a variable exported *read-only* to prevent external destruction of this reference. Read-only export is indicated by a dash following the name, e.g. *stdDir–*.

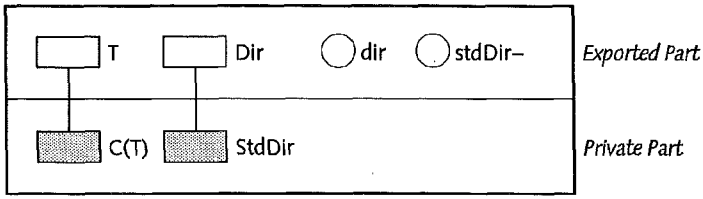


Figure 2.10 – Canonical Module Structure.

The following module interface skeleton results:

```
DEFINITION Ts;
   TYPE
      T = POINTER TO RECORD ... END;
      Dir = POINTER TO RECORD
         PROCEDURE (d: Dir) New0 (...): T;
         PROCEDURE (d: Dir) New1 (...): T;
         ...
      END;

   VAR
      dir, stdDir–: Dir;   (*configurable and fixed (standard) directory objects*)
END Ts.
```

The *New* procedures defined for the directory object are supposed to perform different initializations of the returned object and/or to retrieve the returned object at different places. For example, a file system's directory object would at least provide procedures for creating new files and for retrieving existing files. Often, a refined directory object is supposed to forward unresolved requests to a more basic directory object. For example, a file directory object additionally supporting access to remote files may forward requests for local files to the standard file directory object. In general, it is a configuration issue to which directory object unresolved requests are forwarded. Configurations often resort to eventually forwarding to the default implementation provided by the defining module. Therefore, a canonical module also exports a standard directory object, and typically does so using a read-only variable to prevent external destruction of this configuration anchor.

The module name is often chosen to take the plural form of the name (e.g. Texts) of the defined main abstraction (e.g. Text). In [Nel91] it has been proposed to use the name of the main abstraction as module name (e.g. Text) and to uniformly name the type of that abstraction T. This convention was adopted in [Wil89], but leads to less readable code within the defining module, and is irritating if more than one type is exported.

2.4.4 System Structuring: Summary and Consequences

Several current type-safe object-oriented languages (e.g. Simula, Oberon-2) identify subtyping with subclassing. Separating the two issues anyway leads to a type hierarchy formed using abstract classes, where the concrete classes are attached as leaf nodes. Hence, the merged type/class hierarchy is mainly a hierarchy of abstractions (types), where the leaf nodes provide implementations (classes).

This approach leads to an easily understood hierarchy: Complicated interaction of classes via self-recursion is reduced and indirect recursions spanning multiple subclassing levels are mostly eliminated.

Another argument against extension of concrete classes is the negative effect on extensibility. If two separate extensions of a concrete class exist, they form branches that are hard to combine. The reason is the implicit forwarding to inherited code (subclassing) instead of the explicit forwarding to another object. Figures 2.11 and 2.12 illustrate this: In the case of explicit forwarding it is easy to combine two extensions by configuring the forwarding relation appropriately (a). In the case of inheritance from concrete classes such a combination is hard to do (b). To solve the problem in this case it is necessary to introduce a glue class (c) that does the forwarding to an auxiliary object.

A system designed to be extensible must be expected to be extended by different users in parallel, hence creating separate extension branches. Sometimes such extensions are incompatible anyway, but if they are not it is a good idea to support re-combination of separate extensions to achieve the combined effect of the extensions for a single object. Again, it would be useful if the forwarding style is more explicitly supported by the programming language (cf. 4.4).

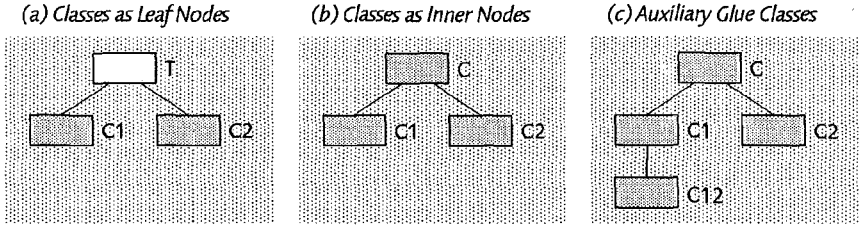


Figure 2.11 – Subclassing of Abstract Classes and of Concrete Classes – Introducing Glue Classes.

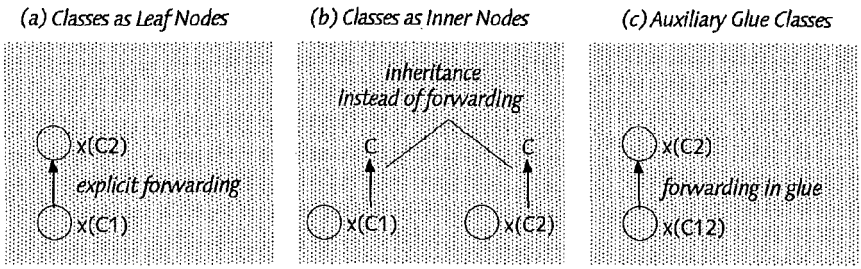


Figure 2.12 – Forwarding Relations in Cases (a), (b), and (c) of Figure 2.11.

To summarize, the use of code inheritance to implement extensions is of limited use. In fact, the use of inheritance to maximize code re-use has been found to be a useless design principle, e.g. [Mag91], just as the introduction of procedures to maximize code factoring is.

However, there are places where code inheritance is recommended and in fact essential, an important example being abstract classes defining and maintaining certain invariants. Such invariants are established by appropriate concrete methods. These are usually overridden in subclasses to add further invariants. For example, an abstract class may need to guarantee that one of its fields is always within a certain value range. First of all, the field can be exported in a read-only fashion. Secondly, an update procedure can be provided that checks the range. If it is expected that a subclass should behave in a specific way when the field gets updated, it is important to use a type-bound procedure.

2.5 Supporting EOO

The term Extensible Object-Oriented (EOO) programming has been introduced in Subsection 1.1.2. In Subsection 2.1.3 the need for supporting EOO in an operating system has been stressed. Finally, in Subsection 2.1.4 required operating system features have been listed. For each of these features, this section looks at detailed requirements. Details on possible implementations of such features follow in Chapter 3.

2.5.1 Generic Object Manipulation

Generic object manipulation is the key feature of object-oriented systems and becomes manifest by means of polymorphic variables. Usually genericity is achieved by binding procedures to variables, based on the type or class of the

object held by the variable at invocation time. For some operations genericity requires special support, though. This can be achieved by adding certain *meta-programming* facilities to a system or a language: Facilities that allow a program to acquire and, in principle, to change information about another program's execution environment [KRB*91]. If a program operates on its own execution environment, meta-programming is called reflection. (A general treatment of reflection features may be found in [GWB91].)

For example, a generic object internalization mechanism needs to retrieve a class by its name and then create an instance of that class. As the number of cases to cover generically is not fixed, the object creation cannot be based on a case switch. Instead, the system (or the language) needs to provide means to get an externally unique name for a class, and to retrieve a class by its name again. Also, the system (or the language) needs to support object creation parameterized by a class. The following pseudo code sketches the situation:

```
WriteObject (s: Stream; o: Object);
   VAR n: ClassName; c: Class;
[[ c := ClassOf(o); n := ClassNameOf(c); WriteName(s, n); o.Write(s) ]]

ReadObject (s: Stream): Object;
   VAR n: ClassName; c: Class; o: Object;
[[ ReadName(s, n); c := ThisClass(n); o := NewOf(c); o.Read(s); RETURN o ]]
```

Obviously, generic object creation, externalization, and internalization requires meta-programming, i.e. run-time introspection of (part of) the type system. At least the four operations (example above) ClassOf, ClassNameOf, ThisClass, and NewOf need be provided. Further introspection facilities, e.g. to traverse the fields of an object in a generic fashion, may also be provided. In general, the need to load modules on demand occurs (cf. 2.5.2), which requires meta-programming support for modules. Typical operations are ModuleNameOf and ThisModule, returning the name of a module and retrieving a module by name, respectively. The latter invokes the module loader if the requested module has not yet been loaded.

System support for meta-programming can be avoided by adding meta-capabilities to the used language. For example by using a *second-level type system*, as in the language Quest [Car89]. Then, every type is an element of a second-level type, a so-called *kind*, just as normal values are elements of normal (first-level) types. This is sometimes called "*making types first-class objects of the language*". A variable t of kind K may hold all types that are elements of K. Let $T \in K$ be such a type. Then an instance of T may be created by assigning T to t and requesting creation of an object of the type given by t. As an analogy, kinds are for types, what higher-order functions are for functions.

When introducing kinds to a language, it is natural to ask for the type of kinds, i.e. to ask for third-level types, and so on. The Metaclass concept of Smalltalk [GR83] models this kind of multi-level meta-programming, where the behavior of classes is modelled

by making all classes instances of metaclasses. While Smalltalk is untyped, it does provide a strong class concept and meta-programming is supported on the class level. The seemingly infinite regress of meta-metaclasses in Smalltalk is broken by introducing a circular construct which identifies the metaclass of all metaclasses, i.e. which collects all meta levels above the second one into a single abstraction.

Instead of providing the illusion of an n-level type (or class) system, or providing a type system with exactly two levels, one may use a conventional (single level) type system and support all higher meta levels by means of system services. This still requires the extraction of type information, and is typically done by the compiler. However, it relies on using low-level features of the language to support meta-programming outside of the language. This way, types are represented by objects encapsulating compile-time information. Such objects are of a normal type, say a pointer to an opaque record, defined using the normal type system.

2.5.2 Dynamic Loading

The notion of *extensible* object-orientation opposed to (plain) object-orientation (cf. 1.1.2) requires the possibility of separate compilation. If units of separate compilation are called *modules*, then this requirement can be made more specific in terms of modules. To do so, the requirements developed in 1.1.2 are quickly recapitulated.

First of all, a module should be compilable without requiring the source code of imported modules. This is essential for extensible systems, as otherwise *all* sources must be present at any one time (which is the case in Smalltalk).

Secondly, compiling a module should have no effect on the correctness of imported modules. Surprisingly, this is violated in otherwise rather modern languages. For example, Eiffel [Mey88] was known for certain breaches in its type system (unsafe covariant typing), rendering type correct programs unsafe. To fix this situation, the newest version of Eiffel (version 3) [Mey92] adds a second level of type checking, called *system-level type checking*. The idea is to check an entire system for potential typing problems. However, this makes it possible that adding a new component to an existing system may exhibit a typing problem in one of the previously existing components! Such a situation is clearly unacceptable in an EOO setting: Adding a new component should by no means cause the *static* checking of a previously accepted component to fail.

For separately compilable modules, the next step is module loading on demand. This requires references to an unloaded module to be detected, the missing module to be retrieved from some secondary store, recursive loading of all imported but not yet loaded modules, and linking of the loaded modules with the imported ones. Finally, the newly loaded modules need to be initialized to establish initial invariants. A module should be initialized after

initializing all of its imports. Care must be taken if cyclic imports are supported: For modules contained in an import cycle there is no natural initialization order, which is another good reason for avoiding cyclic module dependencies.

The demand for loading a module occurs either directly or indirectly. The indirect case is straight-forward and happens when an imported module needs to be loaded. The direct case occurs when the *name* of a module somehow becomes available and the module corresponding to that name is asked for. Examples for the use of module names are user commands and generic internalizations. A user command may be identified by a pair (*module name, command name*); commands are explained in more detail in 3.4.8.2. As explained above (2.5.1), generic internalization is based on using the name of a class to retrieve the class. This requires retrieving the module that defines the named class. Hence, the pair (*module name, class name*) is used to actually identify a class. In both cases, retrieving the named module may cause its loading.

Once a module is loaded, meta-inspection of its features may become necessary. For example, the exported classes (or types) need be retrieved by name. The same may hold for other entities like user commands, where a command is just an exported parameterless procedure, again retrievable by name. Also, it may be useful to reflect on the import graph spanned by the loaded modules.

2.5.3 Garbage Collection and Finalization

Garbage collection and (generalized) finalization support has been motivated above (2.1.4). To summarize, garbage collection is *required* in an extensible system since explicit deallocation always has the potential of introducing dangling pointers or memory leaks. However, the implicit deallocation of objects using a garbage collector also prevents secondary actions to be taken as soon as an object gets collected. If the garbage collector calls a special method of an object just before it gets collected, the object is said to be *finalized*. Finally, if an object is ever to be collected it first must become unreachable. This stands in contrast to *identity directories*. An identity directory maintains at most one copy of objects of a certain kind. Thus an identity directory holds references to all objects it contains, effectively preventing garbage collection of these objects. However, it is usually safe and expected that objects that are held by an identity directory but are otherwise unreachable first get deleted from that directory and then get collected.

The whole idea of supporting garbage collection, finalization, and identity directories for extensible object-oriented systems is based on safety concerns.

Assuming perfect correctness of all components (including all upcoming extensions), these problems can be solved more directly and more efficiently. However, perfect correctness – at least of upcoming extensions – cannot be expected, and safety is a key issue. Therefore the garbage collection, finalization, and identity directory services should be safe, i.e. should themselves be robust to ill behavior of client code (as long as clients are implemented using the safe subset of the language, cf. 2.4.1).

For finalization to be safe it is important to correctly handle objects that as part of their finalization method re-establish reachability of themselves or other objects, or that create new objects. Also, a correctly implemented object must be able to trust that it will disappear after being finalized.

If garbage collection is based on reference counting [Col60], the earliest moment to deallocate an object is when the reference count of that object drops to zero. Therefore it is straightforward to call a finalization routine for the object just before collecting it, check whether its reference count is still zero, and if so deallocate the object [AN88][Atk89]. However, reference counting introduces a significant extra cost during normal operation and cannot collect cyclicly related objects.

More general garbage collection techniques are based on an *asynchronous* detection of unreachability: The unreachability of an object is not immediately detected but only after completion of a global analysis performed by the garbage collector. Several such garbage collection techniques exist [Knu68] [Coh81], including modern refinements to minimize the overhead for systems with high garbage creation frequency [Ung84][Wil92]. For all these algorithms the finalization problem becomes hard: Calling the finalization method of an object may in principle invalidate the results of the reachability analysis. A radical solution would be to finalize a single object at a time and to re-do the reachability analysis afterwards. While this would work, the implied costs would be unacceptable by far.

A solution described in [Atk89] are *weak pointers*. Here, the garbage collector checks for reachability by only tracing normal (strong) pointers. When detecting that an object became unreachable by means of strong pointers, remaining weak pointers to that object are invalidated, i.e. set to NIL. Hence, a weak pointer has the semantics of a strong pointer except for the possibility that it may automatically turn NIL. [Atk89] also introduces so-called *forwarding objects*, where a forwarding object has the same interface as the object it forwards to, but implements it by merely forwarding all requests. [Atk89] describes how weak pointers can be combined with forwarding objects to implement finalization and identity directory mechanisms in a safe way. Hence, weak pointers and forwarding objects can be considered building blocks useful to construct higher-level services. However, this assumes that weak pointers are used quite infrequently, since the implementations

proposed for finalization and identity directories are rather inefficient. To give an impression of the resulting solutions, Figure 2.13 shows a data structure used to implement finalization and identity directories by combining weak pointers and forwarding objects.

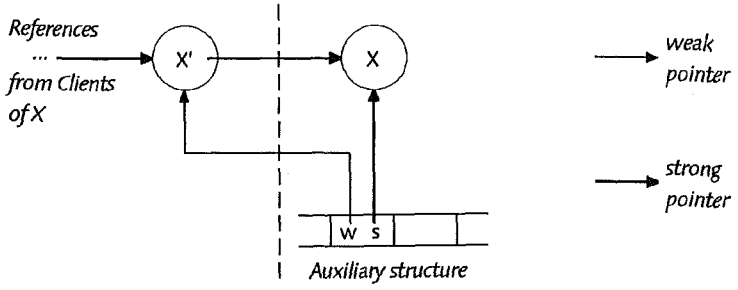


Figure 2.13 – Combining Weak Pointers and Forwarding Objects.

Finalization support for object *X*: The object gets registered in a special auxiliary structure using a strong pointer *s*. All other references to *X* are indirect by means of the forwarding object *X'*, where *X'* is registered in the same auxiliary structure using a weak pointer *w*. As soon as *X'* becomes unreachable and this is detected by the garbage collector, *w* becomes NIL. By periodically inspecting the auxiliary data structure, entries with $s \neq NIL \wedge w = NIL$ can be found, and the corresponding object can be finalized. By scanning the auxiliary structure to retrieve existing objects, an identity directory can be provided. Then each search request is answered by returning a forwarding object instead of the found object.

Weak pointers are not as light-weight as it seems. First of all, the garbage collector should notify implementations of auxiliary structures as the one explained above. Otherwise, each such structure needs to be scanned at arbitrary intervals to look for finalization candidates. Secondly, the complexity introduced to the garbage collector in order to look for weak pointers and invalidate them appropriately is noticeable. These two problems can be solved by requiring explicit registration of weak pointers [Rov84][WDH89]. Thirdly and perhaps most importantly, the use of forwarding objects introduces a run-time penalty for every access to a finalizable object. Also, it becomes hard to support *directly* accessible fields in objects, as the access to such fields is not controllable by forwarding method invocations.

Another issue is the safe support of weak pointers and forwarding objects, which easily becomes an issue of the used programming language. However, having weak pointers or forwarding objects as a language construct makes the concept even more heavy-weight.

A proper compromise may be reached by merging garbage collection, finalization, and identity directory services. The idea is to explicitly register finalizable objects, and to use the registration mechanism at the same time to implement identity directories. The registration structure is known to the garbage collector. Thus it can be utilized to almost avoid additional costs if no finalizable objects are registered: For each registered object a (usually small) cost is added to the garbage collection time. Accesses to registered objects are just as efficient as for other objects. Figure 2.14 illustrates the principle, an implementation is described in Subsection 3.4.2.

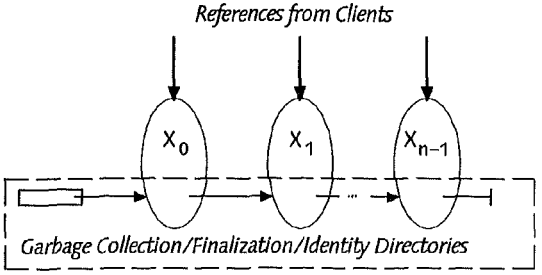


Figure 2.14 – Integration of Garbage Collection, Finalization, and Identity Directories.

The registered objects are linked using sort of weak pointers, where the list forms an identity directory. The location of these weak pointers is directly known to the integrated garbage collector; for other purposes weak pointers are not supported. After marking, the collector traverses the list and picks unmarked entries for finalization. All subtle interactions of garbage collection, finalization, and identity directory services are fully encapsulated in a single component of the system. Therefore, instead of introducing weak pointers as a language concept, the unsafe subset of the used programming language (e.g. module SYSTEM) can be used to implement weak pointers.

Of course, this approach reduces flexibility to a certain degree: The building blocks used to construct the integrated service (e.g. weak pointers) are not individually available. However, during the Ethos project these building blocks were never needed. Identity directory services on the other hand are heavily used in the Ethos system. Finalization is used less frequently, but proved useful in some subtle situations. (For an example cf. 3.4.5, covering the Ethos file system.) To increase the efficiency of searching identity directories it is possible to organize the structure holding finalizable objects in a more refined way than by using a single linked list. (In Ethos a list of lists is used and objects are registered in at most one of these lists.)

2.5.4 Treatment of Exceptions

In Subsection 2.1.4 it has been argued that exception handling of a certain kind is crucial for extensible systems. The point was that the correctness of an extension called from within a base service cannot be guaranteed. Hence, the calling service will need to handle exceptions occuring during execution of the extension, if it has to guarantee invariants.

The main problems with exception handling are: How to handle the complicated flow of control that results if an exception occuring at one place is caught at another, and how exceptions interact with a strong typing scheme. In principle, one may introduce a language feature to handle exceptions, e.g. similar to Modula-3 [Nel91]:

TRY S_i EXCEPT e_0 => T_0 | e_1 => T_1 | ... e_{n-1} => T_{n-1} END;
S_k

Here, S_i is a statement that may raise an exception. If it terminates normally, execution proceeds with statement S_k. However, if one of the exceptions, say e_j is raised, the respective handling statement T_j is executed before continuing with S_k. If an exception occurs which is not listed, or if a handler raises an exception, the TRY statement itself raises an exception.

A problem is: How can the programmer of the TRY statement know which kinds of exceptions can occur? If S_i invokes a late-bound procedure, the possible exceptions that can be raised need be declared together with the signature of the procedure, and occurrence of other exceptions would have to be prevented by the compiler. In conjunction with extensions, declaration of possible exceptions becomes problematic. For example, a procedure P bound to a type T may be declared to raise exceptions out of the set E_P. Then, for a subtype T' of T, an overriding procedure P' is forced to raise exceptions out of a set $E_{P'}$, where $E_{P'} \subseteq E_P$ must hold. This exhibits a conflict: On the one hand, the extended procedure performs refined operations and may meaningfully raise new exceptions. On the other hand, the base type may be the only one known to a client, and it guarantees that only exceptions out of E_P will be raised. Since the exception set may be considered the type of an out-parameter of an exception raising procedure, the described conflict is just a special case of the covariance problem (cf. 1.1.3).

In fact, the proper integration of exception handling mechanisms into object-oriented languages is an open research issue (e.g. [Don90][Lac91]). A certain trend can be observed, treating exceptions as objects passed to handlers. This way, an exception is an instance of (a subtype of) a type, and handlers can deal with general exceptions or with refined "sub-exceptions". For example, this approach has been taken in the latest C++ release [ES90].

Instead of introducing a language-level concept, support for exception handling may be offered by the system. In practice, for Ethos a rather simple approach proved sufficient (3.4.1). The idea is to provide a central stack of exception handlers. Code that needs to guarantee an invariant pushes an exception handler before invoking a critical operation. After successful completion, the exception handler is popped again. However, if the operation terminates exceptionally, the system calls all exception handlers currently on the stack in LIFO order. Each called handler checks and perhaps re-establishes invariants.

```
HandleException
[[ ... ]]
Critical
[[ Push(HandleException); ...; Pop(HandleException) ]]
```

This concept is quite coarse, as it eventually terminates execution completely. If the system supports multiple threads, only a single thread will be terminated. In a single-threaded system, things are more complicated. There, after executing the last exception handler, the system has to restart, probably following some configuration settings. It may be useful to allow exception handlers to install some tasks that will complete or restart work that was in progress when the exception occured. However, often it will suffice to inform the user, as in general it is not expected that the cause of the exception can be resolved automatically.

To conclude, rudimentary exception handling is required to allow an extensible system to guarantee critical invariants in the presense of erroneous extensions. Whether exception handling is raised to a language concept or not, and whether resumption of execution after exceptions is done in a fancy way, is an entirely different issue. Such refinements of the exception handling strategy may ease the life of the programmer (if uncontrollable exception sources are a problem), but have no influence on safety aspects of a system.

2.6 Other (Traditional) OS Functionality

Besides the specific OS features detailed in the previous section, there are many more traditional OS features, many of which need be covered in an effective OS design. This section contains a brief survey of the more important features, while Chapter 3 adds details and decisions regarding the Ethos system. Hence, the following is mainly meant to recall the relevant aspects before looking at the concrete Ethos system. No evaluations or justifications are given in this section – decisions made for a concrete system should take

the context into account, i.e. there is no decision that *in itself* is right or wrong. Instead, the whole system and its application domain must be known to arrive at proper decisions.

Device Abstractions. A computer consists of a variety of physical devices, including processing and memory units, peripheral devices adding secondary storage, and input/output channels. To simplify the context of program execution, an operating system hides the physical devices and their intrinsic subtleties by defining logical devices. A logical device fulfills a certain, clearly defined specification and abstracts from the used set of physical devices that implement the logical device.

Multiprogramming. An important device in every computer is the processing unit. Many operating systems provide an abstraction of the processor (or processors) that allows to execute programs under the assumption that a potentially unlimited set of processing units is available. Historically, this was done to support multiple users on a single machine: While queueing theory shows that processing entire jobs at a time leads to the shortest average service time, it leads to unacceptable delays in an interactive setting where one user has to wait for the completion of another user's task.

Memory Management. Another important resource is the memory system. Adequately managing the memory is usually a cornerstone of every operating system. Systems that fail to manage memory efficiently and effectively are rendered useless. Typical operating systems provide memory abstractions that allow programs to execute under the assumption that the system has a potentially unlimited set of almost arbitrarily sized memory blocks. Historically, this was done to mask small primary memories by automatically swapping from and to secondary stores (e.g. *demand paging*). Another important issue is the support of multi-programming in a multi-user setting: unintended side-effects via shared memory are to be avoided.

Protection. The tendency to use a single computing system for many different tasks issued by different users having different *authorization levels* leads to the problems of resource protection. Besides physically protecting devices by locking them away, robust logical protection becomes a necessity if multiple users are supported simultaneously. The range of possible protection strategies is wide: It spans from simple login passwords, to protection of specific resources, up to specific access protection for individual operations on individual objects.

User Interfaces. An aspect of growing importance is the support of a uniform yet powerful user interface. While early system designs used command line oriented interpreters (*shells*), more modern designs are centered around *graphical user interfaces* following *direct manipulation* principles.

Administrative Support: Finally, but quite important in commercial settings, an operating system may automatically evaluate cost functions. Thereby users can be *billed* for used resources. Also, cost functions enable *fair scheduling*: If two programs compete for a certain resource, the one that used up less resources in its execution history could be preferred.

3 Ethos: A case study

A system which is extensible to everything short of infinity.
(Whitney, 1875)

The Ethos System serves as a vehicle to validate the ideas detailed in the previous chapters. As such Ethos can be seen as a case study. To avoid the touch of Ethos being a toy system, great care has been taken to ensure that the system is actually running and usable.

Analyzing the implications of conceptual work is particularly hard when engineering a complex system. A multitude of interacting design decisions need be taken; often it is not clear if all interrelations have been judged the right way until after a prototype of the system has been built. Hence, a strong motivation for building prototype systems in an engineering field is the experimental judgement of the validity of design decisions. Where this kind of judgement is felt important, the corresponding system may be classified as a *real system.*

This stands in contrast to disciplines investigating isolated problem domains. The need for prototyping systems in the engineering fields should not be misunderstood as insufficient understanding of the problem domain. It is often so that complex systems are *their own most adequate model*; i.e., no adequate model of the system exists that is significantly simpler than the system itself. Hence, the scientific approach of analyzing models at a sufficiently high level of abstraction often fails for real systems. Either the level of abstraction is set so high that relevant details of the modelled system are missing, or the model itself is more complicated to understand or analyze than a prototype of the real system would be.

In a sense, the prototype is the simplest model of the system. However, if the prototype is meant to be refined into the final system, this distinction is blurred. Also, several types of prototyping may be distinguished, e.g. *architectural, explorative* and *evolutionary prototyping.* While architectural prototyping aims at validating and judging design decisions, explorative prototyping is used to explore the space of possible design decisions. Evolutionary prototyping refers to the method of starting with a small kernel system and incrementally improving it to meet requirements. As such it is also a product development technique. The Ethos project aims at supporting all three kinds of prototyping.

3.1 Motivation and Goals

3.1.1 Why Another OS?

Applied studies of extensibility on various levels of abstraction require systems that themselves span a multitude of abstractions. An operating system, in the vague sense of the term sketched in the first chapter, by its very nature spans a wide range of abstraction levels. This makes operating systems ideal objects of study.

Most existing operating systems support extensions only on top of the operating system. This prevents extending the core structures and mechanisms of the operating system itself. Therefore, some systems reduce the "hard-wired" part of the system to a so-called micro-kernel, allowing for extensions on all typical OS abstraction levels. However, the use of a micro-kernel alone does *not* solve the problem of how to structure an extensible operating system. (In fact, concentrating on the micro-kernel design means swapping out most problems to components "out there somewhere".)

Finally, designing a new system from scratch opens many opportunities [Wir89]. For example, it can be studied how to bootstrap a system on a bare machine. Also, the number of design decisions bound by external constraints can be kept small. Last but not least, designing from scratch gives a certain guarantee that problems will be solved completely. This stands in sharp contrast to designing a system by layering it on top of an existing system. In the latter case inadequate design decisions are often caught by adapting behavior of the host system.

> A short example may illustrate this point. When designing a file system, the buffer management policies are very important for the overall system performance. However, when the file system is based on a host file system, the best buffer management policy may be not to buffer at all and to rely on the host system's buffering. Even worse: If complicated buffer management strategies are devised, it gets hard to measure their effectiveness, as all measurements see an interference with the strategies of the host system.
>
> Considering modern caching hardware, this point seems to get weaker. This is not so, since the predictability of the overall system's behavior is reduced with each additional layer. Hence, sitting on top of a host operating system is even worse, when that system itself is already competing with heuristics burned into the hardware.

3.1.2 A Case Study: Concentrating on Important Points

Based on the argumentation above, it was decided to design a system on a bare machine. The effect of solving problems by delegation to an underlying system has thus been avoided. However, the amount of work induced by such a decision is not negligible. Many small wheels have to be reinvented, always keeping the big picture in mind. Therefore it is especially important to concentrate on the essential.

One of the most influential decisions taken was that *everything should be realized using a single (programming) language.* The obvious aims for all levels of the system are: A high degree of readability and understandability, the possibility of easy restructuring, and a reduction in development cycle times. Hence, the trivial choice of using assembly language on all levels fails. Instead, a single high-level language is needed that can adequately cover a broad range of abstraction levels.

The Ethos system contains not a single line of assembly code (but see the remarks below) and no new languages have been introduced for its application-level tools. For example, all device drivers are written in the same language that is also used to describe configuration settings. At a first sight this seems quite a chore. First of all, it contradicts the often made claim (e.g. [Ben88]) that using "little languages" geared towards a particular purpose is preferable over a single "does it all" language. Secondly, one might argue that a language handy enough to perform user level configuration tasks might not be efficient enough to code a device driver. Or, conversely, that a language supporting the effective coding of low-level functions is just not adequate for manipulations at the user level. This might be called the *semantic gap* argument. Thirdly, a language covering all the needs seems doomed to be over-stuffed with concepts and features, making it difficult to learn and master. While such arguments have some truth in them, misconceptions are their usual cause.

Using little languages. If a language is tailored to support a particular tool, it can be small and concise. However, if a user has to master many tools, the *union* of all the language concepts must be learned. The UNIX system with its special languages for shell scripts and dozens of tools has demonstrated this clearly. Also, if tools are used in combination, the individual tool languages interact in a way leading more to a *Cartesian product* than to a union of the concept sets. The resulting combinatorial explosion of facts that the user needs to know is felt unacceptable.

Semantic gap. Considering again the script languages of the UNIX system it is all to obvious that a "user-level" language that is powerful enough to grow

with the user's demands is by no means simple. All the essential concepts of programming need be there. However, instead of using a sane programming language, many such script languages are ad-hoc conglomerates of features that have been added as needed. Other user-level examples are database query languages like SQL. Here, the tendency is to add new structures to cover all computable functions. (Standard SQL is limited to primitive recursive functions.) Then again the result is a complete programming language.

The converse argumentation is a little bit more fruitful. A language capable of expressing device driver code almost certainly is of imperative nature. Thus, using a single language on all levels either asks for a so-called wide-spectrum language (e.g. [B*85]) or forces application-level programming to follow the imperative paradigm. The former approach so far led to languages being rather large indeed, while the latter case may seem overly restrictive. However, it is hoped that adding the object-oriented flavor to an imperative language, most needs can be covered.

Conceptual Over-Stuffing. The expressive power of a language does not simply correlate with the amount of concepts provided. A few concepts, carefully chosen by the language designer, can well add up to a language of sufficient generality and power. Of course, care must be taken not to fall into the other extreme: A minimal set of completely orthogonal concepts can often be improved by adding a few redundant concepts. A classical example are control constructs. In theory, a single concept (say while loops) suffices for a language to be Turing-complete. In practice however, a small set of redundant concepts (say recursion or conditionals) is more adequate to allow frequently occuring patterns to be expressed in a natural way. This is where the language designer's wits get in.

The language chosen to implement the Ethos system is Oberon-2 [MW91]. The selection criteria can be split into two categories. On the one hand, several years of experience with the Oberon system [WG89][WG92] proved the language Oberon [Wir88a] useful for almost all purposes. (The next section explains how the Ethos system relates to the Oberon system.)

On the other hand, several concepts that are hard-wired within the Oberon system are made extensible within Ethos. This is particularly critical at the level of interfaces that need to support a high throughput at a small granularity. An example is the file system, where individual bytes may cross the interface. In Ethos, byte or character streams and primitives accessing the bitmap display are the most critical interfaces. Section 2.3.3 explained the Carrier/Rider Separation concept. Putting this concept forward into a practical implementation implies adding indirections that affect just these bottleneck interfaces. The Oberon concept of message records (more on this below) dispatched by message handlers is too expensive to be applied on this level.

Early experimental versions of the Oberon text system were based on message handling. Since the resulting text system was prohibitively slow, the extensibility of the text system was sacrificed and the interface cast into a set of normal procedures.

The problem with Oberon when applied to object-oriented interfaces on a fine-grained level is that its message concept is *too general*. If a redundant concept would be added such that the programmer could specify the cases where the full generality is not required, more efficient solutions would be possible. An approach taken in Oberon-2 is the addition of *type-bound procedures* (also called methods). Such procedures almost perform like normal procedures as far as their efficiency is concerned [MTG89]. However, the decision which actual implementation to use is delayed until execution time. For type-bound procedures, the first parameter is treated in a special way. Its actual parameter's type (hence the name "type-bound") is inspected to select the appropriate procedure implementation. Since the Oberon type extension scheme allows a variable of a certain type to hold an object of a type that is an extension of the variable's type, the introduction of type-bound procedures leads to an efficient form of late binding.

Type-bound procedures are less flexible than message handlers. However, besides performance advantages, the reduced flexibility is often welcome: The degree of freedom available when using type-bound procedures is limited *by specification*. To put it in other words, the possibility of specifying the minimal set of procedure signatures available for a certain type increases the effectiveness of the type system and thus the scope of static checkability. Furthermore, since it is not possible to change the binding of a type-bound procedure at run-time, the set of such procedures attached to a certain type forms a strong basis for expressing *invariants*: It is impossible to invoke an inadequate procedure for a certain object.

Additionally, the use of type-bound procedures increases the readability of a program. The guiding idea is that one should not introduce more flexibility at any one point than is felt absolutely necessary and that these decisions should be reflected by the type system as accurately as possible. Naturally, such decisions can be mistaken. However, the *possibility* of taking wrong (i.e. overly restrictive) design decisions should not be traded against the *impossibility* of static program checking. (For languages and systems like Smalltalk this is just the case. Nearly *no* errors can be detected until after a program is run and actually has reached a state in which it fails.)

To conclude, Oberon-2 is a complete superset of Oberon. Besides type-bound procedures it adds a few other concepts. The most important ones are the possibility of declaring *read-only exported* variables and the support of variables of type *open array*. Both features increase the expressiveness of the type system in a natural way. Examples for their beneficial use follow in the subsequent sections covering selected Ethos

components.

3.2 Ethos as an Evolutionary Successor of Oberon

Standing on the Shoulders of One's Predecessors. (Isaac Newton)
Standing on the Toes of One's Predecessors. (Paul Feyerabend)

Ethos is an evolutionary successor of the Oberon system in the sense that many of the fruitful ideas realized in Oberon have been retained within Ethos. One way to look at the Oberon system is to view it as an environment supporting rapid development of programs. Such programs span the range from libraries and tools to applications and user interfaces. However, many parts of the Oberon system are fixed and cannot be changed without replacing some of its modules. Of course, such replacements immediately endanger the stability of the system and often cannot be done without rebooting the system, always risking its complete failure.

An important aim of the Ethos project was to enlarge the range of extensibility, i.e. to push down the limits of things that are burned into the system and to allow most services to be implemented equally well by extensions. With a few exceptions this has been achieved. Before looking at details of the Ethos system, the following covers an analysis of advantages and limits of the concepts found in Oberon.

3.2.1 Strong Concepts of the Oberon System

Oberon supports *commands*: Any exported parameterless procedure can be invoked by the system and thus may serve as a command issueable by the user. A command implements an atomic action of the user applied to global data structures. This leads to the second important concept: All data manipulated by the user are anchored in the system's *global state*. When a command terminates the global state is maintained. Results of one command are readily available to serve as arguments to the next command. Due to the static checking capabilities of the Oberon compiler (enabled by the strong type system), commands usually leave the system intact, even when terminating abnormally. This is important when developing programs.

Usually, the global state of the system is anchored within the (itself globally anchored) *viewer system*. Thus, instead of referring to abstract data (i.e. by means of file names or by "piping" data from one command to another) the typical Oberon command evaluates the contents of existing viewers and creates new viewers. By marking a viewer or (part of) a viewer's contents, the user has full (visual!) control over the evolution of data modified by a

sequence of commands. For example, simply by marking some visible text, the user may invoke the Oberon compiler and cause compilation of the marked text. (It is not even necessary to have the source text ever being stored in a file.)

Having the notion of commands, the concept of *applications* in the sense of isolated programs providing some packaged sets of operations is dropped. Instead, commands are named by a pair of module name and procedure name. When invoking a command, the system checks whether the module implementing the command is already loaded. If not, the module and missing imported modules are sought and loaded. This feature of *loading modules on demand* greatly reduces the initial size of the Oberon system while keeping it open for an unlimited set of extensions.

The flexible and extensible organization of Oberon makes earlier languages like Modula-2 [Wir82] inadequate. For example, the viewer list needs to hold variants defined by extensions, i.e. is a heterogeneous list. Since extensions are loaded on demand, the set of viewer variants is not closed, rendering solutions based on variant records impracticable. To allow for heterogeneous structures, in Oberon the notion of *type extension* has been introduced [Wir88a].

> Type extension allows the declaration of a record type, say *T1*, to be based on some existing record type, say *T*. Then *T1* is called an *extension* of *T* and *T* is called the *basetype* of *T1*. For every variable of type *pointer to T*, an assignment of a *pointer to T1* is allowed. Hence, the type of a variable and that of an object the variable refers to may differ: The variable becomes polymorphic (*inclusion polymorphism, cf.* 1.1.1). Similar rules exist for reference parameters of record type, where the actual parameter type may be an extension of the formal parameter type. Mechanisms are provided to test an object type (*type test*), and to guard a variable such that fields defined by the extended type may be accessed (*type guard*). Together, these type extension mechanisms fulfill the demands stated above.

An often posed criticism is the single-threaded nature of Oberon. In many conventional systems the need for multiple processes occurs just because the running system is seen as a collection of running applications. As Oberon considers the running system as a *single* extensible "application" and as all user-interactions are based on atomic commands, the Oberon system has no principle need for multiple processes. Indeed, the notion of quasi-parallel execution has been abolished in Oberon (with the exception of interrupt handlers). As a result, programming is greatly simplified (but cf. 3.4.4).

3.2.2 Limits of Oberon's Extensibility

Contrary to its flexibility, significant parts of the Oberon system are not extensible. This is intentionally so and the primary reason is efficiency. The Oberon system's extensibility is based on message records and message handlers. Each extensible object has a handler attached to it by means of a procedure variable. A handler has two parameters: The handled object, and a reference parameter usually declared to be an empty record. The latter can take any actual type that extends the (empty) formal type. A record passed this way is called a *message record* and the handling procedure a *message handler*. A handler uses a sequence of type tests to select messages to handle. A handler may *forward* a message to some other handling procedure.

Enabling extensibility on system levels that are known to be performance bottlenecks requires careful trade-offs between flexibility and efficiency. While the message scheme of Oberon is flexible, its inherent costs are rather high. The Ethos project aims at demonstrating that the decrease in overall performance can be kept tolerable, even when increasing the extensibility of critical system components. As indicated in 3.2.1, this is mainly achieved by using type-bound procedures instead of message handlers wherever appropriate.

By design the Oberon system has no processes. This greatly simplifies programming. However, situations exist that require the preemption of the current task. This is the case if time-critical external events must be handled asynchronously to user events. For device handling it suffices to have interrupt handlers briefly suspend the main task. However, if the external event requires immediate processing at a higher level, it is not adequate to do so in the interrupt handler. To solve such problems one might want to add a preemptive scheduler. Ethos supports this by providing the atoms required for adding processes of various semantics. In other words, while Ethos has no concept of processes built into it, it provides the means for safely adding processes as an extension.

Where are the differences between Oberon and Ethos? Ethos adds extensibility to the core services found in the Oberon system. This includes the display, file, file directory, and text systems. The Carrier/Rider Separation concept is consistently used to decouple data storage from data access. A generic service for object internalization and externalization is available. All devices are available through extensible services. Hence, devices that conform to an existing device type can be integrated into the system without disruption of service. Furthermore, applications usually do not directly create objects of a certain type but request an instance of a certain type from a so called *directory*

object (cf. 2.3.3). For example, an application opening a text frame to display a text uses the directory object for text frames. By switching this directory object, the application will use different text frames. To allow for safe extensions, Ethos provides a largely refined finalizing garbage collector. Finally, Ethos adds an especially lightweight concept for programming preemptive tasks.

Why are the Ethos enhancements important? One of the primary goals of Ethos is to support run-time *integration of extensions on all levels of the system.* By enhancing the globally available directory objects, the system adapts to newly installed features at run-time. For example, a file system providing access to remote files may be added any time. By installing a refined file directory object, requests for files with a certain path name can automatically be directed to the new file system, while all other requests pertain to the previously available file system. It is crucial that directory objects enable run-time extensibility of applications building on objects provided by lower layers. In the file system example, an application using files can readily use a newly installed file system without any change.

> At a first glance system configuration via directory objects seems dangerous. What if a directory object returns an object not fulfilling minimal requirements? This is precisely the duty of strong type checking: If the directory object by declaration returns objects of a certain type, it is impossible to get an object not conforming to that type. Of course, the actual implementation of the returned object may contain errors leading to misbehavior. Certain run-time exception handling possibilities have been added to Ethos to cope with the more fatal problems of this kind. Usually the system can recover from such situations simply by switching back to some default directory object and by eliminating objects depending on malfunctioning objects. As Ethos is meant to be a programming environment, such counter-measures are usually left to the user.

Another goal of Ethos is to support the writing of *portable programs*, i.e. programs that are largely independent of a particular machine architecture. This is achieved by carefully designing interfaces to *abstract devices*. As long as a program does not rely on features of a specific device, the resulting device independence is guaranteed. Additionally, a set of mapping operations is available that allows for portable creation and consumption of files. This includes the generic externalization and internalization of arbitrary objects, where generic object treatment is driven by the object's types. To enable such type-driven operations Ethos provides access to run-time type information. This eliminates the need for a second-level type system within the language, but cf. 3.4.3.

3.3 Overview of the Ethos System Structure

The design of the Ethos system follows relatively few design principles. The fundamental ideas behind these principles have been described in Section 2.3. This section looks at the realization of the principles in Ethos. Specific components will be treated in Section 3.4.

3.3.1 Modular Structure

The module hierarchy (2.4.1) of an extensible system is hard to capture as it evolves with each installed extension. Figure 3.1 shows the module hierarchy of a typical functional Ethos system. Importing modules are drawn above imported modules. Top-level modules, i.e. those that ought not be imported by other modules, have their names set in bold typeface. Modules belonging to a minimal working configuration are boldly framed, extension modules (existing and proposed ones) are lightly framed. The shaded area at the bottom encloses the set of public but unsafe modules (SYSTEM is a pseudo-module). The shaded areas in the upper middle group modules supporting a particular data model (shown are the texts and graphics subsystems).

Similar to the decomposition of the Oberon system [WG92], Ethos may be understood in terms of layers. The classical "onion" model of layered systems [Dij68] requires a layer to completely hide all lower layers. This is not so for systems like Ethos or Oberon, where the layering has a solely organizational nature helping to maintain an overview over the system structure. All layers remain visible and accessible to all higher layers. Restrictions, if any, are by convention, i.e. by recommending that higher modules ought not import certain low-level modules. (Within limits, such conventions may be enforced by not publishing selected module interfaces, i.e. symbol files.) The basic Ethos system contains only the single restricted module Devices, being the system counterpart to the language module SYSTEM.

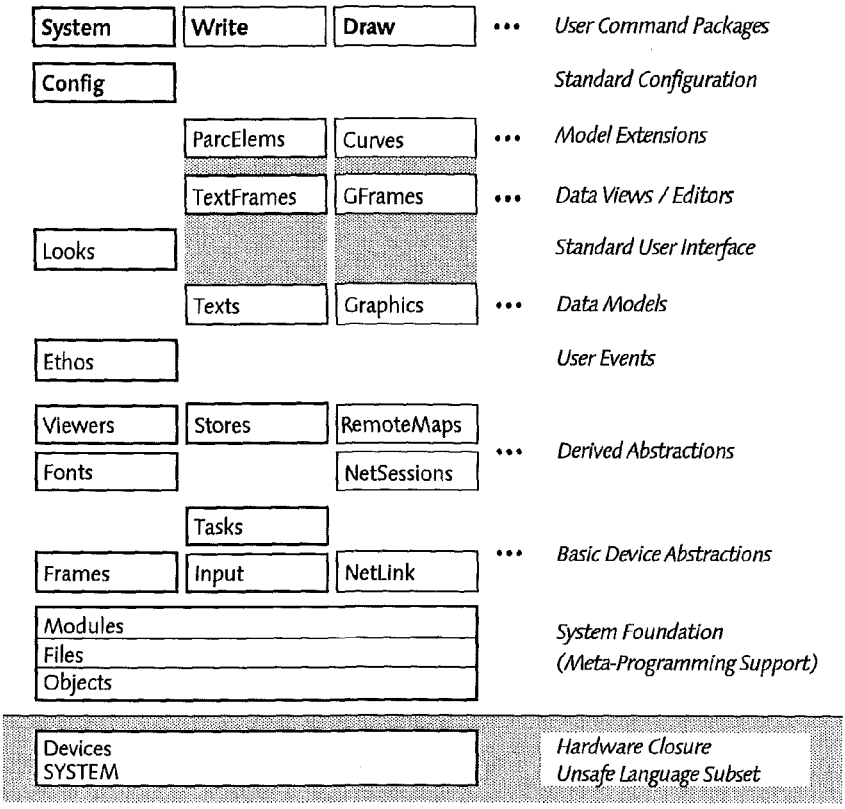


Figure 3.1 – Module Hierarchy.

The module layering reflects the conceptual separation of the system into various levels of abstraction. Additionally, it is influenced by the need to build a foundation structure on which to stack the whole system. Therefore, certain higher-level functions have been moved down in the hierarchy to form the *System Foundation* layer. This includes the basic object definitions, the initial file system, and the initial module loader. (Below the foundation layer, another module with public but restricted interface exists: Devices provides a closure for the hardware.) The module structure above the foundation is not fixed, i.e. can be modified to customize the system. This holds for one exception: Some module named "Ethos" must be present to enter the second bootstrap stage (3.4.8).

On top of the foundation layer, *basic device abstractions* follow. This includes real and virtual devices as well as the corresponding abstract device interfaces. Here, devices not needed in the foundation are added. An

especially important type of abstract device is introduced in Frames: PixelMap, the carrier type for two-dimensional pixel arrays, together with the corresponding rider type Frame (3.4.7). An example for virtual devices is defined in Tasks: Scheduler, a carrier type for processor abstractions, together with the corresponding rider type Task (3.4.4).

Building on basic device abstractions, the next layer adds *derived abstractions*. A derived abstraction is based on an abstract device defined in one of the lower layers. A typical derived abstraction is Mapper defined in module Stores (3.4.5.2). Mapper extends StreamRider defined in Objects. A Mapper is used to generically externalize and internalize arbitrary object structures (e.g. texts) to and from arbitrary positionable streams (e.g. files). Another interesting derived abstraction is the font machinery encapsulated in Fonts which delivers specific fonts (3.4.7.3).

Having all the fundamental but still rather general abstractions at ones disposal, the next layers add *dispatching* of user events (module Ethos, 3.4.8.2) and high-level *data models* (e.g. Texts or Graphics). On top of the dispatcher, the standard user interface, i.e. the common "look and feel" of the system, is encapsulated in module Looks (3.4.8.3).

The *data view / edit* layer adds frames to the data models. A frame combines the user interface and user event dispatcher functionality with that of a specific data model (3.4.8.4). It implements the viewer and part of the controller components as found in the MVC principle.

The last layer providing modules with a programming interface adds model extensions. The *element* extensions of the text system are a typical example (3.4.6).

Finally, top-level modules that are not meant to be imported by other modules package user commands.

3.3.2 Rationals Behind Particular Feature Assignments to Modules

The module decomposition was driven by the aim to create *safe module interfaces*. (cf. 2.4.1). Except for the pseudo module SYSTEM (provided by the compiler) and the module Devices all Ethos modules are safe. The point of a safe module interface is that if the implementation of a module with a safe interface is correct, *no safe client module* can possibly invalidate invariants established by this or some lower module.

As mentioned in 2.4.1, the only known loophole in the language that forces a module *interface* to be unsafe is the special compatibility rule for reference parameters of type ARRAY OF SYSTEM.BYTE. Obviously, a module using such a parameter type to return some value *cannot* prevent that a client (written in the safe subset of the language, i.e. without referring to SYSTEM) uses this parameter to over-write an arbitrary variable

with a meaningless value. Hence, for a module to have a safe interface, the interface *must not* use reference parameters of type ARRAY OF SYSTEM.BYTE! (In Ethos, ARRAY OF CHAR has been used instead. For a client to perform an unsafe mapping of some data structure to an array of characters, the *client* has to import SYSTEM and use SYSTEM.VAL to *explicitly* cause the unsafe type cast.)

Most Ethos modules follow the canonical module structure detailed in Section 2.4.3. It was felt important to avoid clustering things into a module when such a clustering would appear unnatural. For example, module Ethos avoids the over-stuffing found in the corresponding module Oberon of the Oberon system. On the other hand, sets of interrelated types have been packaged into a single module if strong invariants between instances of these types need be maintained. Examples are the strongly interrelated base types of carriers and riders. (The scheme used to do so has been explained in section 2.4.1.)

3.3.3 Abstractional Structure

A secondary structure is present in form of the type hierarchy (2.4.2), which is embedded into the primary structure defined by the module hierarchy. This secondary structure reflects the *abstractional structure* of the Ethos system, as opposed to the modular structure introduced above: Almost every abstraction in the Ethos system is manifest in form of one or more types in the type hierarchy. The very few exceptions are abstractions implemented as services hard-wired into a module and available only via exported procedures. Such exceptions will be explained in detail when covering the associated module.

It is noteworthy that neither the language Oberon-2 nor the conventions of the Ethos system require the type hierarchy to be a tree. Instead, it is a forest where several independent trees exist, and only the largest of which is rooted in type Objects.Object.

The type hierarchy of the standard Ethos system is shown in Figures 3.2 to 3.5, where each box contains the part of the type hierarchy introduced by the corresponding module. The Oberon-2 type system distinguishes between records and pointers referring to records. This distinction would clutter the figures and hence is ignored. Also, auxiliary types that represent no significant abstraction on their own are left out. Each type that is defined to be a subtype of another is listed with indented position beneath its base type. If necessary, types from imported modules are repeated, using an italic typeface and a qualified form *module.type*.

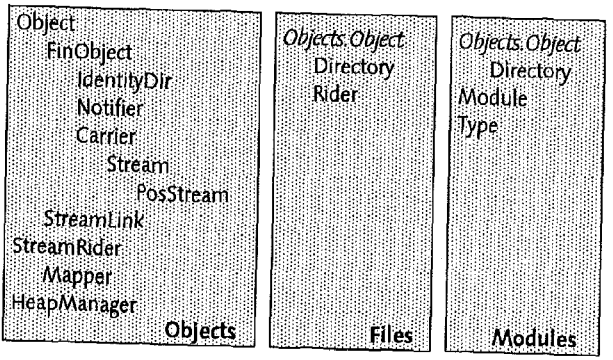


Figure 3.2 – Foundation Type Hierarchy.

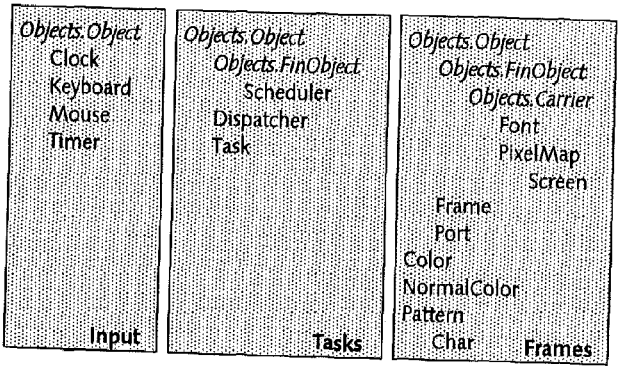


Figure 3.3 – Type Hierarchy of Basic Abstractions.

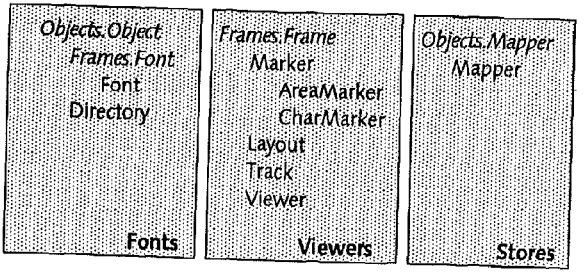


Figure 3.4 – Type Hierarchy of Derived Abstractions.

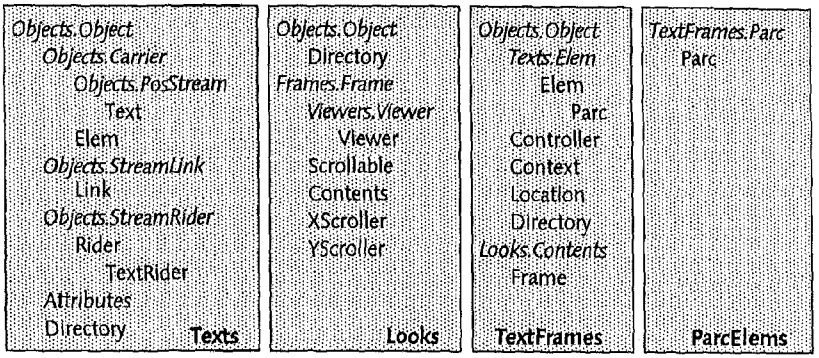


Figure 3.5 – Type Hierarchy of User Interface and Text System.

3.4 Interesting Components of the Ethos System

Besides being an existence proof for the more general design principles, Ethos is also a working system. As such it contains solutions for many problems of varying complexity. This section looks at some of the more interesting problems.

The following sections contain many code fragments. To distinguish pseudo-code from compilable Oberon-2 code, the former is expressed using an (extended) Dijkstra notation. Especially, the semantics of **do** and **if** constructs are as specified in [Dij76]. Simple forms to declare procedures and variables have been added. Interfaces of Oberon-2 modules are introduced using the pseudo-keyword DEFINITION. An ellipsis (...) following the DEFINITION header indicates that only part of the module interface is presented.

3.4.1 Interfacing to the Machine

At the bottom of Ethos module *Devices* provides the hardware closure required by the higher modules in the Foundation Layer (3.3.1). Devices supports the installation of *interrupt, exception,* and *supervisor call handlers,* the *trap dump handler,* and the *main procedure.* Devices also contains the two device drivers needed in the foundation modules, driving the startup and the real-time clock. Both are used by the startup file system which in turn is used by the module loader to boot the system.

Finally, the memory allocation interface is defined in Devices (the memory is seen as a special device). The standard memory allocator (3.4.2.) is implemented in module Objects from where it gets installed in Devices.

3.4.1.1 Installation Support

The following procedures support installation of various handlers:

```
DEFINITION Devices;  ...
   PROCEDURE InstallDump (DP: Handler);
   PROCEDURE InstallMain (MP: Handler);
   PROCEDURE InstallIP (n: INTEGER; IP: Handler);
   PROCEDURE InstallSVP (n: INTEGER; SVP: Handler);
```

InstallDump, InstallMain. Whenever an exception forces the system to interrupt the currently executing code, control passes to module Devices. Devices in turn copies the processor state into exported variables, calls (if installed) the dump procedure, resets the main system stacks and processor state, calls all pending exception handlers (see below) and finally calls the installed main

procedure. If no main procedure is installed the system halts, if the main procedure returns it gets called again.

InstallIP, InstallSVP. Assuming a vectorized interrupt system, for each interrupt channel number a handler may be installed. Likewise it is possible to install supervisor procedures.

Fatal errors in hardware handling functions (stack overflow, dereferencing an unbound pointer, etc.) should lead to a run-time exception, just as for other program parts. However, hardware checks of such conditions are often disabled when executing in supervisor mode. Consequentially one would want to run *the whole system* in user mode. Then again, interrupt handlers and operations accessing certain hardware features are often forced to execute in supervisor mode. On the other hand, to be extensible a hardware handler may need to perform up-calls to extensions. The dilemma then is: How can an operation necessarily executing in supervisor mode perform an up-call to an extension – such that the extension executes in user mode – *and* continue to execute when the extension returns?

The supervisor mode present in many machines prejudicates the way an extensible system can be built. The InstallSVP mechanism makes code executing in the supervisor mode extensible.

```
PROCEDURE ChangeContext (VAR from: SET; to: SET);
```

ChangeContext. This special supervisor procedure completes the breach of the machine's supervisor protection scheme. It allows changing the current execution context: user or supervisor stack, and user or supervisor mode. The first parameter is returned such that when applied as a second parameter to a second call to ChangeContext, the effect of the first call is reverted. Note that this, just like the installation procedures, is a *very* low-level feature to be used by the knowledgeable system programmer. (This is just in the flavor of module Devices which – as explained earlier – has an *unsafe* interface.)

Installing supervisor procedures or switching contexts from user to supervisor mode seem to be clear breaches in a system's safety concept. However, in Ethos the supervisor mode is *not* used to prevent "application" code from accessing protected information, but to *enable* access to certain hardware features not accessible in user mode.

A possibility is the concentration of all operations executing in supervisor-mode into a single module. This has been done in the case of Oberon's Kernel module. Another solution, which has been taken for Ethos, is to provide special means to effectively invalidate the strict user-mode / supervisor-mode separation. On the one hand, new supervisor procedures can be installed at any time from anywhere in the system. On the other, contexts can be switched using procedure ChangeContext. In what follows the worth of this rather uncommon feature of the Ethos system is discussed more thoroughly.

3.4.1.2 Exception Handling Strategy

```
DEFINITION Devices;  ...
   PROCEDURE PushEP (EP: Handler);
   PROCEDURE PopEP (EP: Handler);
```

PushEP, PopEP. As a lightweight compensation for missing exception handling in the used language, Devices allows to push exception handlers before executing a critical code section. If the section completes normally, the exception handler should be poped. (PopEP uses its argument to check for proper pairing of PushEP/PopEP calls.) If the critical section terminates exceptionally, the system falls back into the central exception handling mechanism. There, the exception first causes a trap dump to appear, followed by calls to the pending exception handlers, and completed by calling the main procedure. Exception handlers are called in LIFO fashion. The precise semantics of PushEP and PopEP are:

```
eno: INT; ex: ARRAY MaxEno OF PROC;

PushEP (ep: PROC)
   [[ if eno < MaxEno → ex[eno] := ep; eno := eno + 1 fi ]]
PopEP (ep: PROC)
   [[ e := eno – 1;
      do e ≥ 0 ∧ ex[e] ≠ ep → e := e – 1 od;
      if e ≥ 0 → eno := e fi
   ]]
OnException
   [[ do eno ≥ 0 → eno := eno – 1; ex[eno] od ]]
```

Typical code sequences using exception handlers involve up-calls to installable extensions. The following code fragment illustrates the protection of a global lock variable using an exception handler:

```
VAR lock: BOOLEAN;   (*lock some global resource*)

PROCEDURE* UpCallEP;
BEGIN lock := FALSE   (*release resource if exception occured*)
END UpCallEP;

PROCEDURE UpCall (proc: PROCEDURE);
BEGIN
   Devices.PushEP(UpCallEP);
   lock := TRUE; proc; lock := FALSE;
   Devices.PopEP(UpCallEP)
END UpCall;
```

The combination of exception handlers and context switching is especially powerful. It enables a low-level part of the system to perform up-calls to arbitrary user code without losing safety, i.e. with protection against exceptional terminations. For example, the garbage collector up-calls finalization methods in user mode, but itself executes in supervisor mode. The idea is sketched below:

```
PROCEDURE* UpCallEP;  ...  (*clean-up on exceptional termination of up-call*)
END UpCallEP;

PROCEDURE DoUpCall (proc: PROCEDURE);  (*called in sv mode on sv stack*)
   VAR old, new: SET;

   PROCEDURE Dispatch (proc: PROCEDURE);  (*called on user stack in sv mode*)
      VAR old, new: SET;
   BEGIN
      Devices.ChangeContext(old, {Devices.umode});
      Devices.PushEP(UpCallEP);
      proc;  (*up-call on user stack in user mode*)
      Devices.PopEP(UpCallEP);
      Devices.ChangeContext(new, old)
   END Dispatch;

BEGIN
   Devices.ChangeContext(old, {Devices.ustack});
   Dispatch(proc);
   Devices.ChangeContext(new, old)
END DoUpCall;
```

Technical detail: The two-step context switch (first, from supervisor to user stack; second, from supervisor to user mode) is done assuming that the supervisor stack is not accessible in user mode. Hence, a single step switch would prevent access to objects on the supervisor stack, e.g. the parameter *proc* resides on the supervisor stack but is used after doing the switches.

3.4.2 Memory Management and Garbage Collection

3.4.2.1 Objects

The discussion of the Ethos memory management facilities has been divided up into four subtopics: Memory allocation, garbage collection, object finalization, and treatment of subobjects. All of these topics are covered by the standard heap manager defined in module Objects. The following definitions from Objects are relevant for the subsequent discussions.

```
DEFINITON Objects;  ...
   TYPE
      Object = POINTER TO ObjectDesc;
      ObjectDesc = RECORD
         PROCEDURE (VAR O: ObjectDesc) CopyFrom (VAR o: ObjectDesc);
      END;
```

```
FinObject = POINTER TO FinObjectDesc;
FinObjectDesc = RECORD (ObjectDesc)
   PROCEDURE (F: FinObject) Finalize (VAR canceled: BOOLEAN);
END;

IdentityDir = POINTER TO IdentityDirDesc;
IdentityDirDesc = RECORD END;

HeapManager = POINTER TO HeapManagerDesc;
HeapManagerDesc = RECORD (ObjectDesc)
   PROCEDURE (H: HeapManager) Install;
   PROCEDURE (H: HeapManager) GetInfo (VAR alloc, free, cycles: LONGINT);

   PROCEDURE (H: HeapManager) NewDir (): IdentityDir;
   PROCEDURE (H: HeapManager) SafeToFinalize (f: FinObject): BOOLEAN;
   PROCEDURE (H: HeapManager) Register (d: IdentityDir; f: FinObject);
   PROCEDURE (H: HeapManager) First (d: IdentityDir): FinObject;
   PROCEDURE (H: HeapManager) Next (f: FinObject): FinObject;
END;

VAR
   heap, stdHeap-: HeapManager;
```

The main objects of concern of Objects are, of course, objects. Hence, the primary type defined is Object. All that is required for objects is that a CopyFrom method is available, i.e. that generic copying is supported. Derived from Object, a subtype FinObject adds the finalization property by requiring implementation of a Finalize method (3.4.2.4). A HeapManager is used to encapsulate the memory allocation and garbage collection mechanism. The standard heap manager is available via read-only variable *stdHeap*, while the currently active one is referred to via variable *heap*. Objects are registered for finalization by means of *identity directories.* Identity directories are provided by the heap manager (NewDir), can be traversed (First, Next), and new objects can be registered (Register). A finalizable object may double-check against fake calls of its Finalize method: SafeToFinalize(f) returns true *iff* f.Finalize was indeed called by the heap manager.

The special form of the CopyFrom method is important. Firstly, it is defined on the level of ObjectDesc, i.e. the underlying record type, and not on the level of the pointer type Object. Hence, even objects with types derived from ObjectDesc (and thus statically allocatable), have the copying feature available. Secondly, CopyFrom has been chosen in favor of a perhaps more intuitive CopyTo method. To maintain invariants, it is important that the *modified* and not the modifying object gets control. Also, using CopyTo one might over-write part of an object by only accessing a base-type projection. Reading from a base-type projection using CopyFrom is equivalent to the *projection* defined for Oberon record assignments [Wir88a].

3.4.2.2 Memory Allocation

In Ethos, as in Oberon, all data are stored in global variables, in local variables on a stack, or in anonymous variables in the heap. Global variables are allocated upon module load-time and are not deallocated until after the module itself is unloaded. Allocation and deallocation on the stack is done implicitly by procedure entry and exit sequences generated by the compiler. Dynamic allocations are explicitly requested by the programmer using the standard procedure NEW.

> There are no provisions for explicitly deallocating data from the heap. This is important, as deallocating a heap block with remaining pointers referring to it introduces so-called *dangling pointers*. Dereferencing a dangling pointer is one of the most subtle kind of errors in a program. (In fact, it is amazing to see how much time is spent in typical programming environments to locate such errors, often by means of so-called "debuggers". [Gri91b]) In a closed program such errors can be avoided. However, in an extensible system creation and life-time of pointer aliases is uncontrollable: An error in some client module can therefore break the memory invariants otherwise guaranteed by a host module. Writing to a dereferenced dangling pointer can destroy *global* system invariants and can therefore have arbitrary effects on the running system.

The heap management fully relies on a *garbage collector* to reclaim storage used by data that is no longer accessible. The technique used for allocation and collection has been described in [PHT91], and goes back to the original Oberon implementation [WG89][WG92]. The idea is to allocate blocks in multiples of some minimal chunk size B (typically 16 or 32 bytes). For the first N–1 multiples of B, special free-lists $A[i]$, $0 \le i < N-1$ are used. All other blocks (having sizes iB, $i \ge N$) are linked into a single list $A[N-1]$. Allocation of a block of size s consists of computing the class $K = \lceil s/B \rceil$ and removing a block from free-list $A[k]$ where

$$k \ge K \wedge (\forall i: K \le i < k: A[i] = NIL) \wedge A[k] \neq NIL$$

If $k = K \wedge k < N-1$, the block is consumed in its entirety and thus removed form free-list $A[k]$. If $k > K \vee k = N-1$, the block may be larger than the requested one. If it is, the block is split into two parts, the first one being of the requested size and returned, the second one being again a block of size iB, $i > 0$. The latter block is inserted into the appropriate free-list.

3.4.2.3 Garbage Collection

If allocation proceeded for some time, garbage collection becomes necessary to reclaim storage. In principle, it suffices to call the collector whenever the allocator fails to find a sufficiently large free block. The obvious problem is that the unexpected execution of the collector may cause unexpected

discontinuities in the behavior of an executing program. Hence, a different strategy for calling the collector may be sought. A simple one, also used in the Oberon system, periodically calls the collector when the system is idle. As the system's typical operating mode is the execution of a series of atomic commands issued by the user, a simple heuristic is to count the number of executed commands and call the collector after every *n*-th execution.

> The Oberon system relies on this strategy, i.e. whenever memory is requested it is either allocated or the current command is aborted, but the collector is never run except for the periodic calls during idle times. By the nature of most Oberon commands, this strategy works quite well in most cases. It has the advantage that between commands the stack contains no heap references. Thus the Oberon collector has no need to inspect the stack for active references. However, if a command produces lots of temporary heap structures, the Oberon heuristics is overly restrictive. For example, a Lisp interpreter [Gri91] for the Oberon system uses a recursive evaluation model [WH81], where many heap structures quickly turn into garbage. Due to the collection strategy, the Lisp interpreter runs out of memory far earlier than necessary.

A refined strategy sticks to the periodic collection principle, but additionally calls the collector whenever the allocator fails. This is done in Ethos and has been done for several ported versions of Oberon; first for SPARC-Oberon [Tem91]. A discussion of possible garbage collection strategies may be found in [Coh81]; for Oberon in particular cf. [PHT91] and [WG92].

> A completely asynchronous garbage collector [DLM*78] was rejected for the known inefficiency of the original approach. An investigation of the current literature (e.g. [SS91]) suggests that sufficiently efficient asynchronous collectors depend on memory management hardware to detect asynchronous mutator activities. However, it was felt that Ethos should not rely on having memory management hardware.

Like Oberon, Ethos uses a mark and sweep collector [McC60] based on a recursive descent algorithm using an in-place threaded stack [SW67]. All objects allocated on the heap have a *type tag*, i.e. a pointer at a fixed place referring to a *type descriptor*, cf. Figure 3.6. The information in the type descriptor is used by various parts of the system. For garbage collection purposes it is only relevant to know that a descriptor contains the size of an object and the offsets of pointers contained in an object. Again following [PHT91], this pointer offset table is laid out as shown below. (For simplicity, word sizes and address widths of 4 bytes are assumed.)

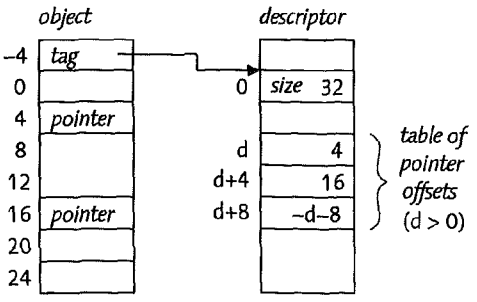
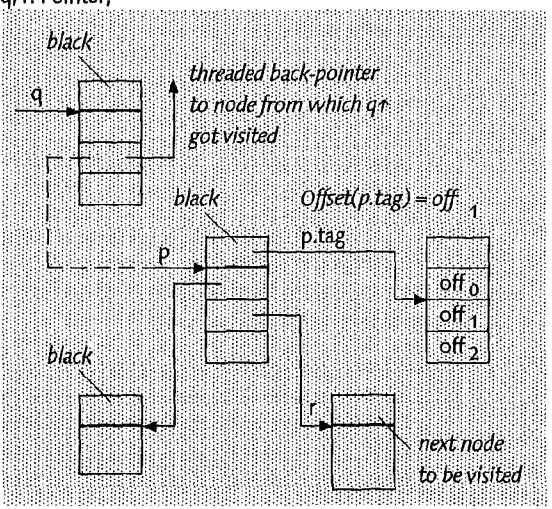


Figure 3.6 – Object and Decriptor Layout.

The Mark procedure uses the tag of an object to remember which descendant of that object has been visited last. To do so, the tag is set to offset *d* (the beginning of the pointer offset table) when first reaching an unmarked object. Then, after visiting a descendant, the tag is incremented to point to the next entry in the pointer offset table. When a negative value is found in the offset table, it is used to reset the tag and leave the object. The Mark procedure is sketched below.

Objects carry a mark field with values out of {*white*, *black*}, where a *black* object is marked. (Further "colors" will be introduced below.) The auxiliary function Offset(p.tag) = Mem[p.tag] returns the pointer offset of the descendant of p currently referred to by p.tag.

```
MarkFrom (p: Pointer)
  q, r: Pointer;
```



```
[[  if p = NIL ∨ p.mark = black → skip
    [] p ≠ NIL ∧ p.mark = white → p.mark := black; p.tag := p.tag + d; q := NIL;
       do Offset(p.tag) ≥ 0 → r := p[Offset(p.tag)];
          if r = NIL ∨ r.mark = black → p.tag := p.tag + 4
          [] r ≠ NIL ∧ r.mark = white → r.mark := black;
             p[Offset(p.tag)] := q; q := p; p := r;
             p.tag := p.tag + d
          fi
       [] Offset(p.tag) < 0 ∧ q ≠ NIL → p.tag := p.tag + Offset(p.tag);
          r := q[Offset(q.tag)];
          q[Offset(q.tag)] := p; p := q; q := r;
          p.tag := p.tag + 4
       od;
       {Offset(p.tag) < 0}
       p.tag := p.tag + Offset(p.tag)
    fi
]]
```

The main invariant of the MarkFrom loop is

$p.tag = black \wedge p.tag = p.tag_0 + d + 4(k-1), k > 0$
$\Rightarrow$ *all objects reachable from* p *via descendants* 0..k–1 *are marked*
$p.tag = black \wedge p.tag = p.tag_0$
$\Rightarrow$ *all objects reachable from* p *are marked*

For a more formal invariant it would be necessary to formalize the notion of "reachability via a descendant pointer", i.e. to model the heap structure as a directed graph.

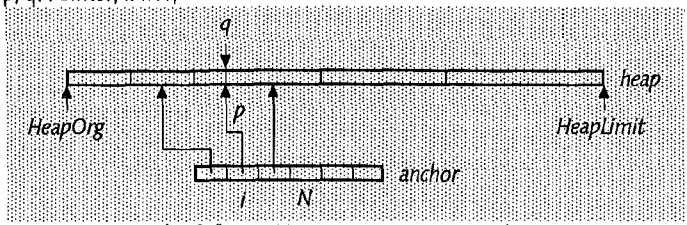
To avoid special cases, the Ethos system has been organized such that a single root pointer suffices to mark all objects in the heap. This includes module blocks allocated by the module loader. Hence, the block containing the global variables and code of a loaded module is tagged. Its type descriptor lists the offsets of pointers contained in global variables of the module.

As mentioned above, Ethos supports collecting garbage when relevant pointers on multiple stacks may exist. A possible solution are type descriptors for activation records. Instead, a conservative marking approach [Bar88] [BW88] is used: Every four-byte integer on the stack is taken as a potential pointer candidate and checked for validity.

The validity checks are close to the ones described in [Tem91]. A 4-byte aligned 4-byte integer is taken as a candidate if it points to an 8-byte aligned block in the heap, the potential block is unmarked and has a 16-byte aligned tag value pointing into the heap. Candidates passing these tests are inserted into a list (procedure AnchorCandidate). The list is sorted and compared against the heap to drop remaining invalid pointer candidates. All candidates that pass this final test are used as roots for block marking (MarkAnchorClosure).

Taking stack traversal into account, the outer garbage collection routine is:

```
N: INT; anchor: ARRAY MaxAnchors OF Pointer;

MarkAnchorClosure
   p, q: Pointer; i: INT;
```




```
   [[  i := 0; p := anchor[0]; q := HeapOrg;
       do i < N ∧ q < HeapLimit →
          if p ≤ q → i := i + 1; p := anchor[i];
             if FreeBlock(q) → skip [] ¬FreeBlock(q) → MarkFrom(q) fi
          [] p > q → q := q + size(q)
          fi
       od
   ]]

AnchorCandidate (p: Pointer)
   [[  if p MOD 8 ≠ 0 ∨ HeapOrg > p ∨ p ≥ HeapLimit → skip
       [] p MOD 8 = 0 ∧ HeapOrg ≤ p ∧ p < HeapLimit →
          if p.tag MOD 16 ≠ 0 ∨ HeapOrg > p.tag ∨ p.tag ≥ HeapLimit → skip
          [] p.tag MOD 16 = 0 ∧ HeapOrg ≤ p.tag ∧ p.tag < HeapLimit →
             anchor[N] := p; N := N + 1;
             if N < MaxAnchors → skip
             [] N = MaxAnchors → Sort(anchor, N); MarkAnchorClosure
             fi
          fi
       fi
   ]]

TraceStack
   sp: Address;
   [[  N := 0; sp := StackPointer;
       do sp < StackOrg → AnchorCandidate(Mem[sp]); sp := sp + 4 od;
       if N = 0 → skip
       [] N > 0 → Sort(anchor, N); MarkAnchorClosure
       fi
       {(∀x : x ∈ Stack ∧ validPointer(x) : x.mark = black)}
   ]]

Collect
   [[ MarkFrom(root); TraceStack; {reachable(x) ⇒ x.mark = black} Sweep ]]
```

MarkFrom is the procedure described above, and Sweep is a simple linear scan over the whole heap, where adjacent free blocks are merged and entered into the appropriate free-list. It has been described in [PHT91] and is not repeated here. (The idea is to uniformly treat all blocks, including free blocks and type descriptors, as blocks tagged with a descriptor containing the block length. Initially all free-lists are cleared. Then, sequences of unmarked blocks are merged and resulting free blocks are inserted into the appropriate free-list.)

At a first glance it seems that the order in which free-lists are built is not significant. If considering a steady state, i.e. one where the heap has been fragmented in a way that allocation of small blocks happens across the heap, this is certainly correct. However, it was found that as long as the periodic collection prevents heap saturation, a proper strategy for building the free-lists can be employed to cause smaller blocks to cluster at the beginning of the heap. Hence, a few, large blocks will exist at the end of the heap area. This is important to decrease chances that a request for a large block will fail due to heap fragmentation.

The used strategy is simple: Whenever rebuilding free-lists during the sweep phase, each list is sorted in order of ascending addresses. A request for a small block that cannot be satisfied by the corresponding free-list then leads to a splitting of the first suitable block counted from the beginning of the heap. In fact, it was found that this strategy is sufficiently effective to avoid heap compaction mechanisms. Oberon for IBM RS/6000 uses this strategy to shrink the heap whenever the block at its end becomes larger than a certain threshold, and on the other hand grows the heap whenever a request for a block, after garbage collection, can still not be satisfied. This allows a *potentially* large Oberon heap without permanently claiming too much memory.

3.4.2.4 Object Finalization

A problem not attacked in general by the Oberon garbage collector is that of *object finalization.* As explained in 2.5.3, finalization is supported by Ethos in tight conjunction with identity directory services.

The Oberon file system provides a procedure Old: Name→File to retrieve existing files by name. The semantics are such that repetitive queries for the same file name retrieve the *same* file (identical pointer value) as long as there is an existing reference to that file in the system. As a result, the programmer may freely compare pointer values to check for file identity. The file system itself also utilizes this property: A file is the instance of caching, i.e. sector buffers are directly assigned to a file. Consistency among multiple accesses is guaranteed just by means of the pointer identity property.

Internally, the Oberon file system uses a linked list of open files to check whether a call to Old corresponds to a file that has already been opened. The only problem is: When to release a file from that list? Since the system cannot rely on Close calls (other pointers to the file may still be active), the only safe way is by means of garbage collection. However, the collector will never remove a file, as it is always reachable via the linked list of opened files! In Oberon this dilemma is solved by special treatment of files, i.e. the collector is aware of the linked list in the file system, and treats it specially.

For Ethos, this decision was felt unacceptable. While Oberon intentionally made low-level services like the file system unextensible, this is not so in Ethos. Instead, a solution was sought that would allow to add extensions in a safe way, such that the

extensions can provide a service with semantics similar to that of the standard file system. (Anecdote: The original Oberon solution was too specialized even for Oberon itself. It had to be changed to also treat a list of fonts specially, as it was found that the Fonts module should have a semantics similar to the Files module...)

The requirements of an identity directory/object finalization service are:

> While a registered object is reachable, it should not be removed from the identity directory used for registration.

- An object registered for finalization is to be considered unreachable if no reference *besides the registration service* exist (i.e. if no "external" references exist).
- When an object registered for finalization becomes unreachable it should eventually become collected.
- An object registered for finalization should be notified before being collected (i.e. it should be "finalized").
- During finalization an object should be able to perform any normal action, including allocation of new objects and re-generation of external references to itself. (This is important, as restrictions to what a finalization code is allowed to do could not be checked efficiently.)
- Whatever the object does during finalization (allocation of new objects, re-establishment of reachability, or exceptional termination), it should not be able to affect the collectors correctness.
- The total cost imposed by supporting finalization should be a small value proportional to the number of objects actually registered for finalization. (The overhead should be negligible if finalization is not used.)

The need for having finalization in a system has been stressed in the literature, e.g. [Rov84][WDH89][Atk89]. Few truly safe implementations of object finalization exist, though.

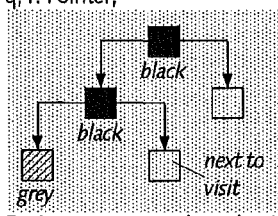
Finalizable objects in Ethos must be of a subtype of type Objects.FinObject. For finalizable objects, the following protocol exists. The registration of objects happens in so-called *identity directories* which in turn can be traversed to enumerate the currently registered carriers.

```
DEFINITION Objects;  ...
   PROCEDURE (F: FinObject) Finalize (VAR canceled: BOOLEAN);

   PROCEDURE (H: HeapManager) Register (d: IdentityDir; c: FinObject);
   PROCEDURE (H: HeapManager) First (d: IdentityDir): FinObject;
   PROCEDURE (H: HeapManager) Next (f: FinObject): FinObject;
```

The key idea leading to safe finalization is to split notification and collection of a finalizable object and to perform each at the end of a separate collection period. Hence, if the notification causes the object to become reachable again, the next time the collector checks the object, this will be discovered and the object will not be collected. By extending the Collect procedure described above, the algorithm looks as shown below.

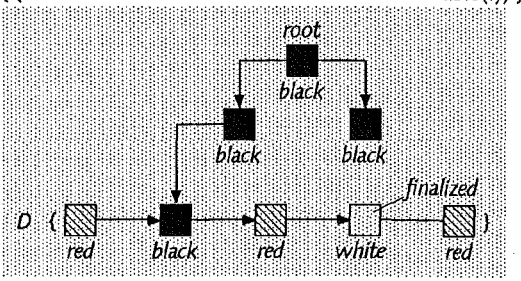
While Objects supports the use of many identity directories, for the sake of simplicity the following presentation assumes that there is only one, namely *D*. Also, the shorthand notation **doall** $f \in D$ is used instead of an explicit linear scan through the finalizable objects *f* registered in *D*. Each finalizable object contains a state field with values out of {*stable, cand, finalized*}. A detailed explanation of the algorithm follows the formal definition. The key problem is to detect objects reachable from finalization candidates (done in TentativelyMarkFrom). This is required to detect the situation where one finalizable object *x* has a reference leading to another finalizable object *y*. Then, it is unsafe to finalize *y* before *x*, as *x* could re-establish external reachability of *y*.

```
D: IdentityDir;

TentativelyMarkFrom (p: Pointer)
   q, r: Pointer;
```

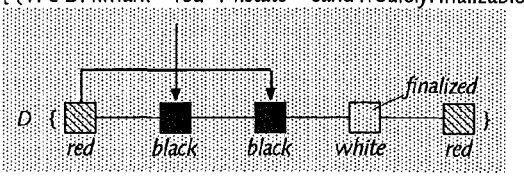



```
[[  {p ≠ NIL ∧ p.mark = white}
    p.mark := black; p.tag := p.tag + d; q := NIL;
    do Offset(p.tag) ≥ 0 → r := p[Offset(p.tag)];
       if r = NIL ∨ r.mark = black → p.tag := p.tag + 4
       [] r ≠ NIL ∧ r.mark = red → r.mark := black; p.tag := p.tag + 4
       [] r ≠ NIL ∧ r.mark ∈ {white, grey} → r.mark := black;
          p[Offset(p.tag)] := q; q := p; p := r;
          p.tag := p.tag + d
       fi
    [] Offset(p.tag) < 0 ∧ q ≠ NIL → p.tag := p.tag + Offset(p.tag); p.mark := grey;
       r := q[Offset(q.tag)];
       q[Offset(q.tag)] := p; p := q; q := r;
       p.tag := p.tag + 4
    od;
    {Offset(p.tag) < 0}
    p.tag := p.tag + Offset(p.tag); p.mark := grey
]]
```

```
MarkCandidates
   f: FinObject;
   [[  doall f ∈ D →
         if f.mark ≠ white ∨ f.state = finalized → skip
         [] f.mark = white ∧ f.state ≠ finalized → f.mark := red; f.state := cand
         fi
       od
       { (∀f ∈ D: f.mark = red ⇒ f.state = cand ∧ ¬Reachable(f)) }
```




```
   ]]
MarkDirClosure
   f: FinObject; m: Mark;
   [[  doall f ∈ D →
         if f.state ≠ cand → skip
         [] f.state = cand → m := f.mark;
            f.mark := white; TentativelyMarkFrom(f);
            f.mark := m
         fi
       od
       { (∀f ∈ D: f.mark = red ⇒ f.state = cand ∧ SafelyFinalizable(f)) }
```




```
   ]]
HandleCandidates
   f: FinObject;
   [[  doall f ∈ D →
         if f.mark ≠ red → f.state := stable
         [] f.mark = red →
            if f.state ≠ finalized → skip [] f.state = finalized → D := D \ {f} fi
         fi
       od
       { (∀f ∈ D: f.state = cand ⇒ SafelyFinalizable(f) ) }
   ]]
```

```
Finalize
    f: FinObject; cancel: BOOL;
    [[  doall f ∈ D →
            if f.state ≠ cand → skip
            [] f.state = cand → f.state := finalized; cancel := FALSE;
               f.Finalize(cancel);
               if ¬cancel → skip [] cancel → f.state := stable fi
            fi
        od
    ]]

Collect
    [[  MarkFrom(root); TraceStack; { (∀x ∈ heap: Reachable(x) ⇒ x.mark = black) }
        MarkCandidates; MarkDirClosure; HandleCandidates;
        Sweep; { (∀x ∈ heap: x.mark = white) }
        Finalize
    ]]
```

Whenever the garbage collector detects that a registered object has no external references, it calls the Finalize method of that object. If the object re-establishes an external reference, or, more explicitly, cancels the finalization by setting the *canceled* flag, the object is not collected. This must take care of any object that itself became reachable just by the survival of the object, i.e. objects that are reachable from the object. However, what does "no external references" mean? The problem is that an object may have references to other objects. Say the finalizable object *x* contains a pointer to the finalizable object *y* and both are externally unreachable, i.e. finalization candidates. Let *y* get finalized first. Then *x* gets finalized and as a side-effect let *x* re-establish external reachability of *y*. Hence y has been finalized and yet is reachable again, violating the semantics of finalization. Thus, only *safely* finalizable objects are considerable for finalization. Such objects have no external references (externally unreachable) and are not reachable from any other finalization candidate (internally unreachable).

To cope with these problems, finalizable objects have a state value associated with them. For an object *f*, the state is kept in *f.state* and takes values from {*stable, cand, finalize*}. The live-time of a finalizable object may then be characterized by its various state values:

f.state = stable ⇒ f *is not considered for finalization*
f.state = cand ⇒ f *is internally and externally unreachable, i.e. to be finalized*
f.state = finalized ⇒ f *has been finalized, i.e. is to be collected*

The collector has been extended to use additional marking "colors". Marking values out of the set {*white, grey, red, black*} are used, where *white* and *black* have the same meaning as before. The sweep phase treats *grey* and *red* values like *black* ones. A finalizable object is marked *red*, if it is found to be externally unreachable and therefore needs to be checked for reachability from another finalization candidate. To check for such references, the special marking procedure TentativelyMarkFrom is used. It works almost as MarkFrom presented before, but upon recursive *ascent* it leaves objects marked *grey* instead of *black*. Also, on recursive descent it marks objects *black* but treats *grey* objects like *white* ones. Whenever a *red* object is reached it is marked *black* but not further traced. Therefore, TentativelyMarkFrom can be used to check whether from any one object a path exists to a *red* object (which in turn becomes *black*).

> Termination of TentativelyMarkFrom is guaranteed, as the current *descent* path is marked *black*. Since the new marks added by TentativelyMarkFrom are *grey*, a second execution of TentativelyMarkFrom starting at a different object can traverse paths that the former call has traversed before.

Thus, by marking all finalization candidates *red*, calling TentativelyMarkFrom for each of them, and then dropping all candidates that have turned *black*, the safe candidate set is determined. The principles behind this algorithm have – independently – been discovered before [AN88].

The possible transitions of state values are:

stable → cand :	*after marking took place, the object is still unmarked* (white)
cand → stable :	*checking internal reachability caused marking of the object* (black)
cand → finalized :	*just before calling the finalization method*
finalized → stable :	*finalization got canceled*
finalized → stable :	*the next mark-phase after finalization detected reachability* *)

*) *occurs* only *if the finalized object itself misbehaved during finalization*

Figure 3.7 illustrates the state transitions as they correlate with the various procedures of the garbage collector. Actions happening in the *n*-th collection period are marked with "(n)", while those happening in the subsequent collection period are marked "(n+1)".

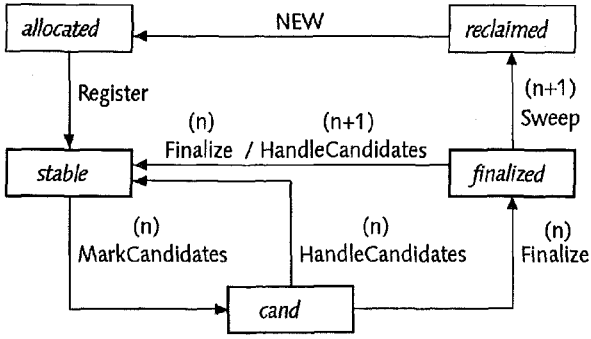


Figure 3.7 – State Transitions during Object Finalization.

It is noteworthy that the Finalize procedure is called *after* the sweep-phase took place. Therefore, an exceptionally terminating finalization method will leave the system intact. The corresponding object will survive or be removed depending on its reachability established before the finalization method terminated. All other finalization candidates will be taken care of after the next collection period.

Remark on references between finalizable objects: Generally, it is expected that finalizable objects are not interrelated in a cyclic fashion. If cycles exist, such rings of objects will never be finalized because there is no possible *safe* way to do it. Also, even acyclic references between finalizable objects are considered the exception. For a chain of such objects, say *A* referring to *B*, *B* referring to *C*, etc., the collector has to finalize one object at a time – starting with *A* – even if the whole chain is externally unreachable. In the current Ethos implementation references between finalizable objects did not occur at all.

3.4.2.5 Treatment of Subobjects

While object finalization is considered an important concept of the Ethos system, in what follows a mechanism is described which allows to tune system performance, but which is not considered a concept in its own right. As a motivation consider a font system that answers requests for certain fonts by returning font objects. Each font object in turn answers requests for certain characters rastered for a specific device resolution. Typically, a font object rasters all its characters when being created, or loads the rastered characters from a file. It then maintains an array of character objects. Each character object describes the metrics (bounding box and the like) of the corresponding character and contains a pointer to a raster object. The raster object contains the character's raster for a specific device. The following picture illustrates this.

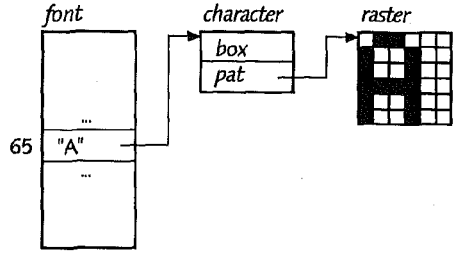


Figure 3.8 – Font System using Separate Objects for Fonts, Characters, and Raster Data.

A font is indexed in the range 0..255, but typically many characters are undefined, i.e. mapped to a single "empty" character object. Still, a typical font contains about 100 characters. As a result, for each rastered version of a font, about 200 objects (character plus raster objects) are allocated. If one considers some 10 fonts (different sizes of the same font family count separately) used in a system and allows support for, say, two different resolutions, then about 4000 objects are independently kept in the heap. This has consequences for the garbage collector which has to mark and scan all these objects each time it is invoked. However, all objects belonging to a font will not be reclaimed until after the font itself becomes unreachable. An object related that tightly to another object shall be called a *subobject* of the latter (which itself is called the "parent-object" of the former). Hence, the knowledge that character and raster objects are subobjects of a font object could be used to enhance the collector's efficiency. Another point is that it is often much more likely to have pointers to a parent-object than to any one of its subobjects. In the case of font objects, references to character objects (from outside of their font) tend to be short-lived.

> With the upcoming Unicode fonts the index range grows to 0..65'535 to accomodate all glyphs found in all the languages used worldwide. For font objects supporting Unicode, the argumentation above gets even stronger, as the number of individual subobjects per font may increase to something like 20'000 objects. For the numbers given above, one would have to expect some 400'000 objects!

A possibility would be automatic "aging" of such objects, as is done in generational garbage collectors [Ung84]. In this case, all objects that "survived" a collection period will be aged and henceforth collected at a slower rate, e.g. only with every n-th collection cycle. This can be modified to cover many generations collected at different speeds [UJ88]. However, generational collectors do not make use of a particular subobject relationship but rather "guess" the stability of an object by looking at its relative age. In the font example, it is likely that a font persists for quite some time and migrates into an old generation. Then it will take considerable time before the system

notices that the font became unreachable and could be collected.

For the Ethos collector a more straight-forward way has been chosen: Subobjects are allocated *within* their parent-object. Hence, they are normally not seen at all by the collector's mark and sweep phases. However, if a pointer to a subobject is found during marking, it must be treated in a special way. Instead of marking the subobject, which is not visible to the sweep phase, the parent-object must be marked. Hence, the marking algorithm must be aware of subobject pointers and treat them different from normal pointers. (Would the pointer to the subobject simply be ignored, the collector might collect the parent-object despite of its still reachable subobject. Hence, the subobject reference would become a dangling pointer!) Figure 3.9 illustrates the allocation principle.

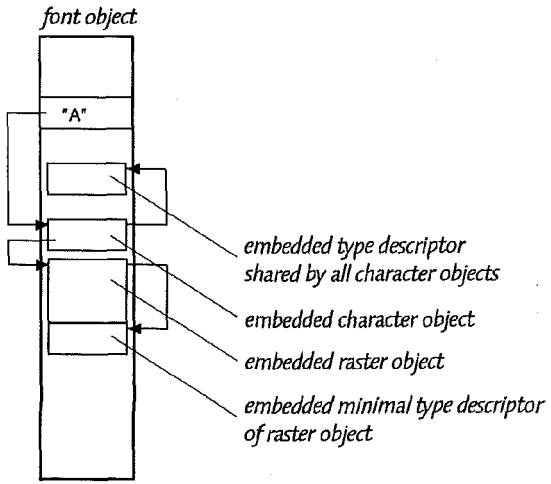


Figure 3.9 – Using Subobjects to Embed Character Objects into Font Objects.

The allocation of subobjects inside their parent objects is not directly supported by Ethos services. Hence, a module implementing subobject relations needs to resort to the unsafe language part, i.e. module SYSTEM. The correct allocation of subobjects is thus a matter of correctness of a module implementing such a scheme. Correctly allocated subobjects are aligned in a certain way and have type descriptors that themselves are subobjects of the same parent object. If subobjects are allocated correctly, client modules need not know whether a pointer to an object is actually a pointer to a subobject, as the collector treats pointers to subobjects in a special and safe way.

How can normal pointers and subobject pointers be possibly distinguished? The solution is to use an *implicit flag*. Normal objects in the heap are always aligned: A normal pointer p has the property p MOD $B = 0$ (where B is the

allocation chunk size defined in 3.4.2.2). Hence, if subobjects are allocated in a way that this property is guaranteed not to hold, subobject pointers are easily detectable. For example, by aligning subobjects such that a subobject pointer q has the property q MOD $B = 2^k$, bit k of a pointer value directly distinguishes subobject from normal pointers.

To show how the mark phase can cope with subobjects, the modified MarkFrom is listed below with the changes set in italic typeface. TentativelyMarkFrom is modified analogously. It should be noted that subobjects have proper tags and usually need a valid type descriptor. (For example, a character object might be inspected using a type test.) This type descriptor is simply another subobject allocated in the same parent-object. To avoid overly complex effects on the marking algorithm, the restriction has been introduced that a parent-object must be a leaf, i.e. contain no pointers other than to its subobjects. This is a transitive property, thus subobjects may not contain pointers other than to subobjects in the same parent-object. With these restrictions holding, the marking algorithm merely has to check for the subobject pointer property, and if it holds, has to mark the parent-object instead of marking the referenced object. However, it need not traverse potential descendants of that parent-object.

The leaf restriction could be removed at the price of a more complex marking algorithm. Also, in the current implementation this restriction is by convention, i.e. should be respected by allocators of subobjects. Like other subobject allocation criteria it is not checked.)

```
MarkFrom (p: Pointer)
  q, r: Pointer;
  [[ if p = NIL ∨ p.mark = black → skip
     [] p ≠ NIL ∧ subobj(p) → p.tag.tag.mark := black
     [] p ≠ NIL ∧ ¬subobj(p) ∧ p.mark = white → p.mark := black;
        p.tag := p.tag + d; q := NIL;
        do Offset(p.tag) ≥ 0 → r := p[Offset(p.tag)];
           if r = NIL ∨ r.mark = black → p.tag := p.tag + 4
           [] r ≠ NIL ∧ subobj(r) → r.tag.tag.mark := black; p.tag := p.tag + 4
           [] r ≠ NIL ∧ ¬subobj(r) ∧ r.mark = white → r.mark := black;
              p[Offset(p.tag)] := q; q := p; p := r;
              p.tag := p.tag + d
           fi
        [] Offset(p.tag) < 0 ∧ q ≠ NIL → p.tag := tag + Offset(tag);
           r := q[Offset(q.tag)];
           q[Offset(q.tag)] := p; p := q; q := r;
           p.tag := p.tag + 4
        od;
        {Offset(p.tag) < 0}
        p.tag := p.tag + Offset(p.tag)
     fi
  ]]
```

3.4.2.6 Coding of Finalization and Subobject Attributes

The sections above introduced several attributes attached to every object (in form of a tag), and in some cases even to every pointer (in the form of certain bits with special meaning). In this section it is shown how these attributes are coded in Ethos.

A tag needs to contain the following information:

outside mark phase:

- address of type descriptor *(used for type tests and method lookup!)*

during mark phase:

- address of type descriptor, or of next descendant offset in type descriptor
- marking value ∈ {*white, red, grey, black*}

Every pointer (including tags outside the mark phase, cf. remark below) needs to contain the following information:

- pointer refers to a normal object or to a subobject

Tags are encoded into a 4-byte word (Figure 3.10). Other pointers just have the *subobj* bit (at the same position as tags).

> At a first glance it seems unnecessary to have the *subobj* bit in tags. However, as will be explained in Section 3.4.3, normal pointers to type descriptors may exist. Since type descriptors are allocated in the heap, such pointers are followed by the mark phase. Hence, a tag is just a specially placed pointer pointing to an otherwise normal object in the heap. Indeed, type descriptors themselves have tags pointing to a minimal descriptor appended to the end of the descriptor. This appended descriptor has a tag pointing to itself.

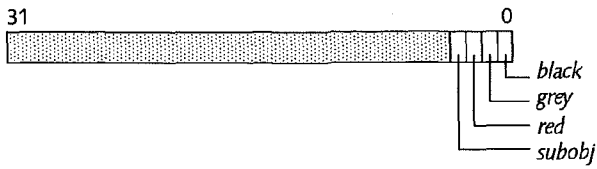


Figure 3.10 – Flag Bit Encoding in Tags.

The *subobj* bit is part of the actual address (due to proper alignment). Bits *black*, *grey*, and *red* are cleared during the sweep phase. Hence, outside of the mark phase the tag contains a proper type descriptor address. A slight problem arises during the mark phase. Since the tag is used to step through the pointer offset table within a type descriptor its value changes in

increments of the size a single offset value takes. In the extreme case, as for the Ceres implementation, the offsets are stored as 2-byte integers. This implies that the *grey*, *red*, and *subobj* bits are meaningless while the descendants of an object are traversed. Luckily, this is acceptable: The MarkFrom procedure does not inspect these bits if the *black* bit is set, which is done before traversing an objects descendants.

Josef Templ [PHT91] has proposed to add another bit to tags, called the *array* bit. The idea is to support iterative traversal of arrays in the heap. This proposal may be combined with the coding explained above. (In fact, this combination has been done in the ported version of Ethos, cf. 3.6.) Since this *array* bit is required even while traversing descendants, it is important to assign it to bit one and to increase the size of offset entries in the type descriptors to 4-byte integers. The *subobj* bit should not be moved to a higher bit as this has direct consequences for the alignment of subobjects and the resulting internal fragmentation of parent-objects. Hence, the following coding is proposed, where the sweep phase clears the *black*, *grey*, and *red* bits:

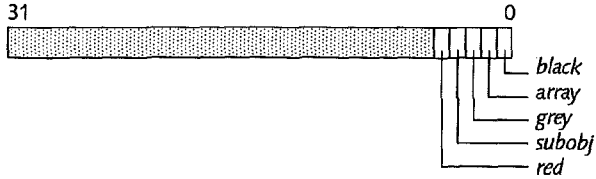


Figure 3.11 – Flag Bit Encoding in Tags, Supporting Arrays of Arbitrary Size.

3.4.3 Module Loader and Meta-Programming

The module loader is a central component of the system: Following Section 2.5.2 it is crucial for run-time extensibility. The Ethos standard loader is provided by module Modules. For several reasons the Ethos loader has been constructed differently than the one found in the Oberon system [WG92]. This section briefly introduces the loader function and otherwise concentrates on the essential technical differences.

The main function of the loader is to take a module name, check whether the module is already loaded, and if not, load it into the running system. As a module may in turn import other modules that have not yet been loaded, it seems quite natural to formulate the loader recursively. This is indeed the approach taken for the Oberon loader.

After loading, the modules have to be initialized in a proper sequence, i.e.

no module should be initialized before all imported modules have been initialized, and every module should be initialized eventually. For cyclic imports this is obviously impossible; cyclic module imports are therefore forbidden. This is checked and the loader refuses to load module groups containing cycles.

Surprisingly, just the final step of module initialization is the most interesting one. In order to initialize a module, the loader has to pass control to the body of the newly loaded module. If the body code performs some plain variable initializations and otherwise returns properly, no problem occurs. However, a module body cannot be trusted as it may perform arbitrary computations. Following the general Ethos principle of maintaining system invariants even in the presence of certain errors, it was felt unacceptable that abnormal body behavior may lead to global inconsistencies of the system.

This is just what happens in the case of the recursive Oberon loader. If the body of a module *M* terminates exceptionally, the loaded module will remain in the system, although only partially initialized. Even worse, modules loaded before and importing *M* will also remain in the system, completely uninitialized. To complete the disaster, it is possible that one of these modules not only imports *M* but also, say, *N*. Then *N* may not even be loaded at the time the exception happens. The principle problem is that exceptional termination violates the stack principle and leaves the recursive loader in a dangerous situation. To recover from such a situation would require adding a complicated exception handling mechanism. (Note that the Ethos exception handling mechanism would not suffice: The state information required to recover is contained in the stacked activation records and thus lost after the trap!)

3.4.3.1 A Non-Recursive Module Loader

To avoid all the problems mentioned above, the Ethos loader has been designed in a *non-recursive* fashion, following an idea of Beat Heeb [Hee88]. It has been refined to gracefully recover from exceptions.

The basic idea is *not* to use the stack to maintain relevant loader state information. Instead, the loader maintains three global sets of modules: the *load-*, the *init-*, and the *ready-set*. A newly encountered module is put into the *load-set*. Modules in the *load-set* are loaded one after another and moved to the *init-set*. Finally, modules from the *init-set* are initialized and moved to the *ready-set*. Figure 3.12 explains how the sets are kept in two rings of module descriptors, where the first ring represents the *ready-set*, and the second ring represents the remaining sets. (This particular structure has been chosen to ease the implementation of the required set operations.) A module descriptor contains the name of the module plus a few pointers into the corresponding

module block. This is illustrated in Figure 3.13. A module block cannot be allocated before knowing the precise space requirements, i.e. before reading the corresponding object file. The separate module descriptors allow to maintain information about modules not yet loaded.

The following explanations are based on the Ceres version of Ethos. Following a restriction of the used NS32xxx processor, the Ceres architecture does not allow module descriptors to be allocated freely. Instead, all module descriptors must reside within the first 64K of memory. Hence, they are managed separately from the normal heap using a fourth set – the *free-set* – of module descriptors. Whenever a fresh module descriptor is needed, it is taken from the free-set. (If the free-set would become empty, loading fails.)

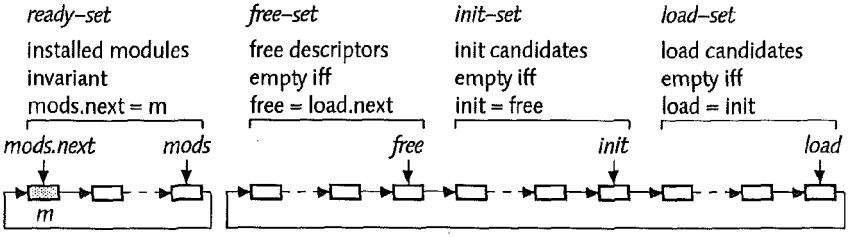


Figure 3.12 – Structure of the Module Sets.

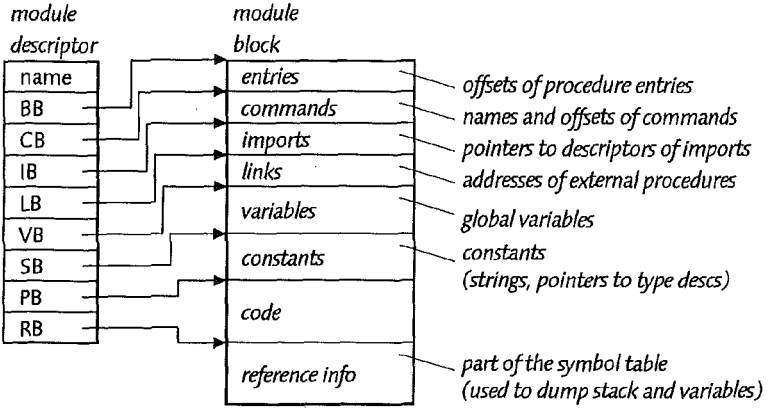


Figure 3.13 – Module Descriptors and Module Blocks.

The loader is implemented by procedure ThisModule which is *goal-driven*, where the goals are defined by the module sets. Figure 3.14 illustrates the transitions of a module from one set to another. The loader does not try to initialize any module until after the transitive closure of *all* imported modules

is in the *init-set*. This property is used when resetting the loader after a failure (cf. below). Figure 3.15 gives an example how a particular set of modules is loaded.

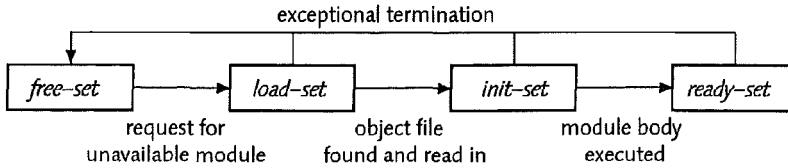


Figure 3.14 – Transitions between the Module Sets.

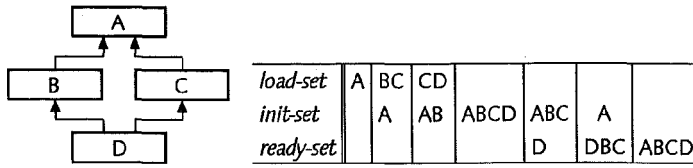


Figure 3.15 – Example of Loading a Set of Modules – Starting with A, Importing B, C, and D.

The loader algorithm is sketched below.

```
mods, free, init, load: Module;
res: INTEGER;

SearchInList (beg, end: Module; name: Name) returns (m: Module);
   [[  m := beg;
       do m ≠ end ∧ m.name ≠ name → m := m.next od
   ]]

Clear (m: Module);
   [[ m.name := "~" (*reset other module fields*) ]]
ResetLoadGoals (from: Module)
   m: Module;
   [[  m := mods.next;
       do m ≠ mods ∧ m ≠ from → m := m.next od;
       if m ≠ from → skip [] m = from → from := free fi
       m := from;
       do m ≠ load → m := m.next; Clear(m) od;
       init := from; load := from
   ]]
ResetGoals
   [[ ResetLoadGoals(free) ]]
```

```
LocateModule (name: Name) returns (m: Module)
   [[  m := SearchInList(mods.next, mods, name);
       {m.name = name ∨ m ∉ ready-set}
       if m.name = name ∨ free = init → skip
       [] m.name ≠ name ∧ free ≠ init → m := SearchInList(free.next, init, name)
       fi;
       {m.name = name ∨ m ∉ ready-set ∪ init-set}
       if m.name = name ∨ init = load → skip
       [] m.name ≠ name ∧ init ≠ load → m := SearchInList(init.next, load, name)
       fi;
       {m.name = name ∨ m ∉ ready-set ∪ init-set ∪ load-set}
       if m.name = name → skip
       [] m.name ≠ name ∧ load.next = free → m := NIL; res := "out of memory"
       [] m.name ≠ name ∧ load.next ≠ free → load := load.next;
          m := load; m.name := name    {m.name = name ∧ m ∈ load-set}
       fi
       {m = NIL ∨ (m.name = name ∧ m ∈ ready-set ∪ init-set ∪ load-set}
   ]]

LoadModule (m: Module; f: File)
   (* create new module block, load module from file f, use LocateModule to spot imports
      (this may cause imports to be entered into the load-set), if error detected set res *)
LoadNextModule
   m: Module; f: File;
   [[  m := init.next; f := ObjFileOf(m.name);
       if ValidObjFile(f) ∧ KeyMatch(m, f) → LoadModule(m, f)
       [] ¬ValidObjFile(f) → res := "invalid object file"
       [] ¬KeyMatch(m, f) → res := "key mismatch"
       fi;
       if res ≠ 0 → skip [] res = 0 → init := init.next fi
   ]]

LinkModule (m: Module)
   (* use entry lists of imported modules to link m *)
LinkLowestModule
   p, m: Module;
   [[  {load = init ∧ res = 0}
       p := free; m := p.next;
       do p ≠ init ∧ ¬ImportsInitialized(m) → p := m; m := m.next od;
       if p = init → res := "cyclic import"
       [] p ≠ init → p.next := m.next;
          m.next := mods.next; mods.next := m; mods := m;
          LinkModule(m);
          Devices.PushEP(ResetGoals);
          m.body; res := 0;
          Devices.PopEP(ResetGoals)
       fi
   ]]
```

```
ThisModule (name: Name) returns (m: Module)
   top: Module;
   [[  res := 0; top := init;
      m := LocateModule(name);
      do (load ≠ init ∨ init ≠ free) ∧ res = 0 →
         if load ≠ init → LoadNextModule
         [] load = init → {init ≠ free} LinkLowestModule
         fi;
         if res = 0 → skip [] res ≠ 0 → ResetLoadGoals(top) fi
      od;
      If res = 0 → skip [] res ≠ 0 → m := NIL fi
   ]]
```

To understand the special behavior of the non-recursive loader as opposed to its recursive counterpart, the special routines ResetLoadGoals and ResetGoals need be studied in more detail. To begin with, one might consider the cases where no exceptional terminations of module bodies are involved. In this case there are two calls to ResetLoadGoals, both in ThisModule. ResetLoadGoals is invoked if the loading of a module *m* failed. In this case all pending modules *of the current ThisModule invocation* need be removed. Here, the stack fashion of the recursive loader is simulated by stacking the value *top*, indicating which module was prepared for processing before calling ThisModule. However, why is *top* required at all, i.e. why isn't it sufficient to call ResetGoals when detecting a load error? (Before continuing, the reader may pause and try to find out!)

One has to consider the case where the body of a module just being initialized recursively invokes the loader. In this case, the loader may fail to load the requested modules. However, the loader should not disturb the state it was in before being invoked recursively. This is just done by resetting only up to *top*. An important example is a module that uses the loader to install an optional feature. If the feature is not present, the loader clearly should proceed without throwing away the module that checked for the option.

> The procedure ResetLoadGoals has to check whether *top* still points to a module in the *init-set*. If not, all modules in the *init-* and *load-sets* have to be removed, as the original module pointed to by *top* has moved to the *ready-set* before the error occured.

A similarly tricky condition is present in the calling sequence invoking a module's body (procedure LinkLowestModule). Here, after calling the module body, the global result state *res* of the loader is reset. Why? Again, if the body performed a recursive invocation – causing the result state to become non-zero – this result is *not* relevant for the ongoing initialization operations and should not lead to an abortion of it.

Next, one might look at the call to ResetLoadGoals after calling LinkLowestModule. There is only one condition that causes the invocation of ResetLoadGoals at this point: The presence of cyclic imports among the

modules to be initialized. Since the loader does not allow cyclic imports, *all* involved modules are thrown away.

Finally, when the body of a module traps, i.e. terminates exceptionally, ResetGoals is called as an exception handler. In turn all modules currently under preparation are dropped and the loader is reset to a stable state, i.e. one involving only correctly loaded and initialized modules.

The module that trapped remains in the *ready-set*, as it may already be referenced by another module that has been loaded and initialized due to a recursive loader invocation. An exceptional termination of a module body is considered to be a programming error in that module leading to a potentially inconsistent module initialization. Since this is also possible for other programming errors leading to normal termination, leaving the module in the *ready-set* is a consistent policy.

Why it is at all a good idea to move a module to the *ready-set* before its initialization has been completed? Again, this is a rather subtle point, and again, it involves recursive loader invocation from within a module's body. Consider the case where the body of a module, say *M*, is invoked while *M* is left in the *init-set*. Then, as soon as the loader is invoked from within the body, the loader will follow its goal-driven strategy until the *load-set* is empty. Then it will look for a module in the *init-set* that is ready for initialization, and will find module *M*! Thus the loader will again call the body of *M*.

When moving the module to the *ready-set* first, the loader instead proceeds to initialize the next higher modules and returns when it is done with that. In turn, the body of module *M* continues. This is the best a loader can guarantee: If a module body invokes the loader recursively, it is not possible at the same time to continue the execution of the body immediately *and* to return the recursively requested module in a properly initialized state.

By now, one might argue that the price paid for the loader's robustness is too high. Indeed, the recursive Oberon loader is simpler. However, the havoc situations that it may establish in the system are found clearly unacceptable. Monitoring the work of less experienced programmers (students) revealed that such situations do occur and that they are rather hard to diagnose. (Usually, the ubiquitous reaction is to reboot the system – just to be sure ...)

It can be claimed that a recursive loader could cope with these problems by establishing global state and using marking techniques similar to the ones presented above. Indeed, this is deemed possible. (The MacOberon loader, for example, uses this approach to deal with *some* of the mentioned problems.) However, combining recursion and global state negates most of the elegance of the recursive solution: If all problems are to be solved, the recursion state cannot be used to determine what to do next, and the global state needs to be inspected anyway. (It is interesting to see that the recursive Oberon loaders implemented for the various system ports behave very differently when being confronted with such problematic scenarios.)

Another objection might be that most of the tricky problems could be circumvented by not allowing a module body to recursively invoke the module loader. (The problem

of exceptional termination of module bodies remains.) This, however, is far too restrictive. For example, even the standard Oberon system uses two such loader invocations during its bootstrap process! (The first one to load module Oberon after initializing module Modules; the second one to load module System after initializing module Oberon.)

3.4.3.2 Module Unloading

The inverse operation of module loading is module unloading. During normal operation of the Ethos system, modules are never unloaded. However, when programming new modules it is necessary to replace old module versions by new ones.

The rational for never automatically unloading a module once it has been loaded is twofold. On the one hand a module represents state anchored in its global variables. It is never known whether the user later wants to use the stored data, or not. On the other hand, the main argument for automatic unloading would be storage preservation. This argument is of no practical significance since typical Oberon(-2) modules are rather small, and the system loads each module at most once.

At first glance, module unloading is trivial: The list of loaded modules is searched for a named module and, if found, it is removed from that list. The matter gets more complicated when looking at the many interrelations in the running system a module may be involved in. Firstly, a module may have *clients*, i.e. modules importing it. Secondly, a module provides entities referenceable from anywhere in the system. These are type descriptors of types defined by the module, and externally referenceable procedures.

The client problem can be solved easily by maintaining a reference count in every module. The count indicates how many client modules currently exist that import the module at hand. If a module's reference count does not equal zero, the unloading of this module is prevented.

Far harder is the second problem. The type extension scheme permits a variable of a base type to refer to an object of an extended type. Hence, it is possible that a variable defined in a lower-level module refers to an object of a type defined by the module to be removed. Likewise, method descriptors and procedure variables reachable from a lower module may refer to procedures or methods defined in a higher module. For mere type descriptor references, the problem is solved easily by allocating type descriptors in the heap. Thus, when unloading a module, the type descriptors it defined remain valid and are eventually collected when the last object of these types disappears.

In principle, there are several alternative solutions to cope with the problem of procedural references. Firstly, it is possible to prevent unloading of modules with pending procedural references. Secondly, one may release a module from the module list upon a request for unloading, but keep the whole module in the heap until after the last procedural reference is gone. (This is done in a

newer version of Sparc-Oberon.) Thirdly, all procedural references to a module about to be released may be invalidated. (This is done in the Ceres-1/2 versions of Oberon and, to some extent, in the Ceres-3 version of Ethos, cf. inset below.) Fourthly, the problem may be ignored, i.e. the system may be left in an unsafe state where invocation of an unloaded module is possible. (This is done in the Ceres-3 version of Oberon.)

The first approach tends to make it hard to unload a module, as the user has to know (and eliminate) all existing procedural references. The second approach may be understood as a kind of "renaming" of a loaded module. Its effect is very similar to the (theoretical) method of renaming the new version of a module and just leaving the old version as it is. Removing the unloaded module from the module list allows its removal by the garbage collector as soon as no references exist any more. The third approach also has its problems: It may lead to system failure if some installed extension gets partially malfunctioning by invalidating some of its procedural references. Finally, the fourth approach is undesirable, as it leaves the system in an unpredictable state.

The Ethos version for Ceres-3 uses heuristics close to the third approach. The heuristics are build into procedure Clear (3.4.3.1) called within FreeThis (below). Clear resets a module descriptor before returning it to the *free-set*. In case of the Ceres-3 implementation, all external procedural references take an indirection via the program base *PB* contained in the module descriptor. Clear sets the program base to point to a special area filled with *Break* instructions. As long as the module descriptor is not reused for a new module, this heuristics prevents dangling procedural references to be invoked unnoticedly. To avoid early re-use of the procedure descriptor, released modules are added to the end of the list representing the *free-set*, while newly assigned module descriptors are taken from the beginning of that list.

Often it is necessary to unload more than one module, e.g. when replacing a complete subsystem. Since all modules are members of a directed acyclic graph, automation of unloading typical subgraphs supports common replacement tasks. Figure 3.16 illustrates two typical cases of unloading subgraphs. If module *A* is imported by module *B*, then *A* is called a *host* module of *B*, and *B* is called a *client* module of *A*. Directed edges in the import graph go from host to client modules, e.g. from *A* to *B*.

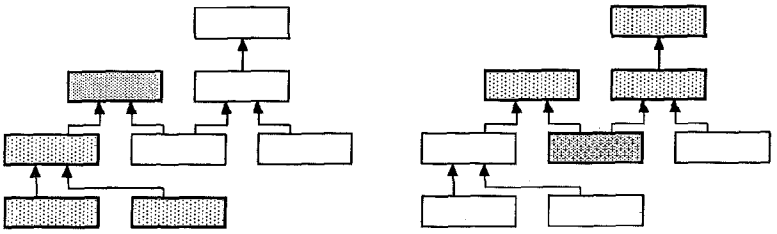


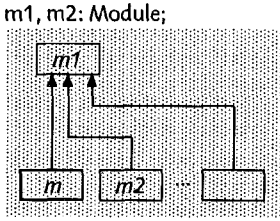
Figure 3.16 – Module Unloading. Left: Incl. Host Modules – Right: Incl. Client Modules.

In both graphs the modules to be unloaded are shaded. In the example on the left-hand side, a top-level module (darkly shaded) and all of its independent host modules are unloaded. Here, a host module is called independent, if it has no other client modules. In the example on the right-hand side a bottom-level module (darkly shaded) and all of its clients are unloaded.

The first method (*host unloading*) is most useful for conventional module stacks, where a single top module exists. For example, the whole compiler may be unloaded by unloading its top module and all of its independent hosts. The second method tends to unload rather large parts of the system if applied to a commonly used module. The idea is to apply it to a new base module defining some new abstract type. Then all the clients and all derived extensions will be unloaded. For example, one might work on an extensible graphics editor in which case unloading of the foundation module, say Graphics, causes unloading of the entire graphics subsystem. While the first method is available also in the Oberon system, the second has been added in consideration of the typically inverted structure of extensible systems, where no single top module exists.

Both unloading strategies are implemented rather easily in a recursive fashion. As can be seen from procedure FreeModule, any combination of single module unloading and the two closure methods is supported.

```
FreeThis (m: Module)
   [[  p := mods;
       do p.next ≠ m → p := p.next od;
       Clear(m); p.next := m.next;
       m.next := free.next; free.next := m;
       if mods ≠ m → skip [] mods = m → mods := p fi;
       if init ≠ free → skip [] init = free → init := m fi;
       if load ≠ free → skip [] load = free → load := m fi;
       free := m
       {m ∈ free-set}
   ]]

FreeClients (m: Module)
   m1, m2: Module;
```



```
[[  if m.refcnt = 0 → skip
    [] m.refcnt ≠ 0 → { (∀m1 : m1 ∈ ready-set : m1 imports m ⇒ m1 ∈ {m.next .. mods}) }
       m1 := m.next;
       do m1 ≠ mods.next → m2 := m1.imports;
          do m2 ≠ NIL ∧ m2 ≠ m → m2 := m2.nextimp od;
          if m2 ≠ m → m1 := m1.next
          [] m2 = m → {m1 imports m} m2 := m1.imports;
             do m2 ≠ NIL → m2.refcnt := m2.refcnt – 1; m2 := m2.nextimp od;
             FreeClients(m1);
             m2 := m1; m1 := m1.next; FreeThis(m2)
          fi
       od
    fi
]]

Free (m: Module; hosts: BOOL)
   m1: Module;
   [[  if m.refcnt ≠ 0 → res := "unloading failed"
       [] m.refcnt = 0 → m1 := m.imports;
          do m1 ≠ NIL → m1.refcnt := m1.refcnt – 1;
             if ¬hosts ∨ m1.refcnt ≠ 0 → skip
             [] hosts ∧ m1.refcnt = 0 → Free(m1, hosts)
             fi;
             m1 := m1.nextimp
          od;
          FreeThis(m)
       fi
   ]]

FreeModule (name: Name; this, clients, hosts: BOOL)
   (* free this (named) module, free its clients, free its hosts *)
   m: Module;
   [[  m := LocateModule(name);
       if m = NIL → res := "not loaded"
       [] m ≠ NIL → res := 0;
          if ¬clients → skip [] clients → FreeClients(m) fi;
          if ¬this ∧ ¬hosts → skip [] this ∨ hosts → Free(m, hosts) fi
       fi
   ]]
```

3.4.3.3 *Meta-Programming and Reflection Facilities for Modules, Commands, Types, and Objects.*

An important aspect of an extensible system is the capability to *reflect on its own state*. Examples are services that support listing all modules currently loaded, that invoke commands by name, or that support generic type-driven externalization and internalization of arbitrary objects. In all these cases it is necessary to somehow access information *about* the currently established

system. Such information is called *meta-information*, and the use of it is called *meta-programming*. If a program operates on the basis of its *own* meta-information, this is called *reflection*.

Meta-programming in the Oberon system is provided for accessing module and commands by name. It is not available for types and objects. Also, the attributes of a loaded module are only accessible via low-level use of untyped information. (Oberon module descriptors export a bundle of addresses pointing to various sections of a module block. The information they point at is only accessible using module SYSTEM and is only interpretable by following machine-dependent conventions.) The Ethos module Modules generalizes these meta-programming capabilities and in turn allows to extend services using these features without the need for unsafe programming.

```
DEFINITION Modules;  ...
   CONST
      insert = 0; delete = 1;   (*directory notification codes*)
      temp = 0; unloaded = 1;   (*module states*)

   TYPE
      Command = PROCEDURE;

      Type = POINTER TO TypeDesc;   (*TypeDesc intentionally not exported*)

      Module = POINTER TO ModuleDesc;
      ModuleDesc = RECORD
         PROCEDURE (M: Module) Info (): ModInfo;

         PROCEDURE (M: Module) GetName (VAR name: ARRAY OF CHAR);
         PROCEDURE (M: Module) GetCommands (): CmdInfo;
         PROCEDURE (M: Module) GetImports (): ImpInfo;
         PROCEDURE (M: Module) GetTypes (): TypInfo;

         PROCEDURE (M: Module) ThisCommand (name: ARRAY OF CHAR): Command;
         PROCEDURE (M: Module) ThisType (name: ARRAY OF CHAR): Type;
      END;

      Name = ARRAY 32 OF CHAR;
```

```
ModInfo = POINTER TO ModInfoDesc;
ModInfoDesc = RECORD (Objects.ObjectDesc)
   next: ModInfo; mod: Module;
   state: SET; (*temp, unloaded*)
   BB, CB, IB, LB, VB, SB, PB, RB: LONGINT;   (*machine dependent*)
   size, key, refcnt: LONGINT;
   name: Name
END;

CmdInfo = POINTER TO CmdInfoDesc;
CmdInfoDesc = RECORD (Objects.ObjectDesc)
   next: CmdInfo; cmd: Command; name: Name
END;

ImpInfo = POINTER TO ImpInfoDesc;
ImpInfoDesc = RECORD (Objects.ObjectDesc)
   next: ImpInfo; mod: Module; name: Name
END;

TypInfo = POINTER TO TypInfoDesc;
TypInfoDesc = RECORD (Objects.ObjectDesc)
   next: TypInfo; type: Type; name: Name
END;

DirNotifyMsg = RECORD (Objects.NotifyMsg)
   dir: Directory; op: INTEGER; mod: Module; name: Name
END;

Directory = POINTER TO DirectoryDesc;
DirectoryDesc = RECORD (Objects.ObjectDesc)
   ref: Files.Directory;
   notify: BOOLEAN;   (*if true, directory change notifications are broadcasted*)
   res: INTEGER; cand: Name;   (*cand: name of the offending module causing res*)

   PROCEDURE (D: Directory) FreeModule
      (name: ARRAY OF CHAR; this, clients, hosts: BOOLEAN);
   PROCEDURE (D: Directory) GetDir
      (prefix: ARRAY OF CHAR; this: BOOLEAN): ModInfo;
   PROCEDURE (D: Directory) ThisModule (name: ARRAY OF CHAR): Module;
END;

VAR
   dir, stdDir-: Directory;
```

The module loader is encapsulated into a so-called module directory. As can be seen from the interface excerpt above, it is possible to install a different loader by changing the value of variable *dir*. (As always, the standard loader is still accessible via the read-only variable *stdDir*.) As the module loader contains the definition of object file formats, different formats may be

supported by installing loader extensions. Module directories provide error state information corresponding to the last lookup, a reference to the file directory to be used when seeking currently unloaded modules, and a flag indicating whether the directory should notify about changes (cf. 3.4.8.4).

Modules are implemented as abstract objects, i.e. no internals are directly exported. Instead, a module has a set of methods that return various information about a module. For meta-programming purposes, the methods GetCommands, GetImports, and GetTypes are of interest. These allow to operate on the module hierarchy and the set of commands and types provided by each of the modules. (The function procedures ThisCommand and ThisType are provided for efficiency and convenience in the cases where a single command or type with known name is looked for.)

As a matter of style, Ethos modules do not provide enumeration features for abstract data structures. Instead, selective copy-out mechanisms are used that take a snapshot of the structure and return a description in a separate linked list. This avoids the hazards with iterators in imperative programming languages. In the example of Modules, the entire module list, or any of the module-specific lists of imports, commands, or types are copied out.

Further meta-programming capabilities are provided on the level of types and objects. It is possible to retrieve the type of an object and it is possible to create instances of a given type. Also, a type may be picked by name and the name of a type may be extracted. Finally, it is possible to reflect on the type hierarchy by examining the level of a type, and by retrieving the base types of a type.

In principle, it would be possible to extend the type-level meta-information to cover all information contained in the original type declaration. This would require changing the compiler and the object file format such that the required information is effectively passed from the compiler to the run-time data structures (type and module descriptors, module blocks). Likewise, the meta-information on the module-level could be extended. Such refined capabilities are not required in the current Ethos system, but could be added as extensions. However, their precise treatment is beyond the scope of the project described in this thesis.

Since all record types defined anywhere in the system are in principle retrievable by name, it is possible to create instances of non-exported types. To allow some control over this conceptual loop-hole, retrieval of types – by name or by object – that are not derived from Objects.ObjectDesc is prevented.

The procedures providing meta-information on the type system are:

```
DEFINITION Modules;  ...
   PROCEDURE GetTypeName (t: Type; VAR name: ARRAY OF CHAR);
   PROCEDURE BaseOf (t: Type; lev: INTEGER): Type;
   PROCEDURE LevelOf (t: Type): INTEGER;
   PROCEDURE ModuleOf (t: Type): Module;

   PROCEDURE TypeOf (obj: Objects.Object): Type;
   PROCEDURE NewObj (VAR obj: SYSTEM.PTR; t: Type);
   PROCEDURE GenObj (src: Objects.Object; VAR dest: SYSTEM.PTR);
   PROCEDURE CopyObj (src: Objects.Object; VAR dest: SYSTEM.PTR);
```

For convenience, the procedures GenObj and CopyObj have been added. Both implement commonly used combinations of other procedures presented above and may be used to implement a more efficient shortcut. GenObj takes the type of a source object and creates a new instance of that type. CopyObj makes use of the fact that the thereby created object is derived from Objects.ObjectDesc and calls the method CopyFrom to actually copy the data from the source to the newly created destination object.

The power of the provided operators may be judged by looking at some typical uses (where x and y are variables pointing to some objects, s, t, and u are variables of type Type, and p is a Boolean):

```
p := SameType(x, y):          p := TypeOf(x) = TypeOf(y)
p := x IS t:                  s := TypeOf(x); p := BaseOf(t, LevelOf(s)) = t
p := SubType(x, y):           t := TypeOf(y); p := x IS t
u := CommonBaseType(s, t):    i := Min(LevelOf(s), LevelOf(t)); u := NIL;
                              do i ≥ 0 ∧ u ≠ NIL →
                                 if BaseOf(s, i) ≠ BaseOf(t, i) → i := i + 1
                                 [] BaseOf(s, i) = BaseOf(t, i) → u := BaseOf(s, i)
                                 fi
                              od
```

The rationale for placing all these features into module Modules is the relation between a type and its defining modules. Hence, a module below Modules – e.g. Objects – could not possibly provide these services. The introduction of a separate higher, almost empty module for this purpose was felt pompous.

3.4.3.4 Further Design Decisions and Technical Remarks

When designing the Ethos module loader several smaller problems had to be attacked. This section covers some of the more interesting ones. However, the material presented below is not essential for the understanding of the underlying concepts.

Everything in the Heap. For Ethos the design decision was taken to allocate everything either on the main stack or in the heap. (If additional stacks are used to support, say, coroutines, these are allocated in the heap, too.) This decision stands in contrast to those taken for many Oberon implementations where module blocks and, in some cases, type descriptors are not allocated within the heap but maintained in separate areas. On Ceres, this results in a conflict between a growing heap and a growing module block area. The heap may grow to its limits and then get fragmented by subsequent garbage collections. In turn, it is no longer possible to load new modules, although, in principle, sufficient space would be available. To cure this problem without allocating modules in the heap requires a second-level scheme managing the allocation and release of memory pages *within* the heap. Also, the separate allocation strategy complicates the garbage collector: Pointers contained in global variables of a module reside in the module area and are explicitly traversed. In Ethos the unification of all allocation mechanisms made it easy to use a single marking algorithm starting at a single, fixed root address.

Loading Packaged Module Sub-Trees: Libraries. For experimental purposes the Ethos loader has been made up to support a second object file format. Files of this format are called *libraries* and essentially consist of a directory plus a series of standard object files. The idea is to allow packaging of complete, closed sub-systems into a single file. For example, the Ethos compiler [Cre91] consists of nine modules. However, only the top-level module has a command and programming interface meant to be used by other modules not part of the compiler. Also, it is somewhat error-prone to ensure that all nine modules – that is, that all nine object files – are present and consistent. Bundling all compiler modules into a single library file named after the top-level module does the job quite nicely. Then, is it necessary to *restrict* the mechanism to *closed* sub-systems? In fact, the restriction is a matter of conventions. Since the loader takes a library apart and installs each module individually, it is possible to package any set of modules into a library. The problem occurs when trying to locate one of these modules before it got loaded: For the top-level module its name can be used to find the library file, but for other modules no simple mapping exists to determine which library needs be searched. (A possible remedy is to nevertheless package such sets of

modules, and to preload the whole library as part of the startup configuration.)

Detecting Calls to Unloaded Modules: A Heuristics. As discussed above in the section on unloading modules, problems occur when procedural references (via procedure variables or method tables in type descriptors) to an unloaded module remain. Several solutions to this problem have been discussed where all the cleaner ones essentially prevented this situation from happening at all. The Ceres version of Ethos is based on the standard object file format as defined by the Ceres Oberon implementation. Thus, the Ceres version does not have sufficient information at its hands to implement any of these strategies. Instead, a heuristics has been implemented which catches problems in most cases. Firstly, module descriptors are not immediately reused after unloading a module. (They are eventually, as the pool of available descriptors is limitted to a rather small amount by the Ceres hardware.) Secondly, all external calls to procedures are implemented using an indirection via the callee's module descriptor. Thirdly, as long as not being reused, a module descriptor of an unloaded module points to a special code block statically provided by the loader. This code block is filled with *Break* instructions. Changing the code base in the descriptor immediately effects all existing procedural references to the corresponding module: If a procedural reference is used to invoke a procedure (or method) implemented in an unloaded module, an exception is risen.

> Detail: The *Break* instruction is a single byte instruction with code 242. The byte following this instruction is by convention understood as the break number. Hence exception handling code can detect that the cause of the exception was a call to an unloaded module.

Extensibility of Heap Manager and Module Loader. One might note that the internal representations of objects, types, and modules are not exported. How can a new heap manager or module loader be added then? The current implementation relies on implicit module coupling (2.4.1) to achieve this: The internal conventions (memory layout, invariants) used by the heap manager or module loader need be maintained by the extensions. This is not as unsafe as it seems since – without importing module SYSTEM – it is *simply not possible* to implement a heap manager or module loader that does more than adding filtering functions to the standard implementations. If, on the other hand, the need comes up to experiment with, say, different object file formats, it is *possible* to add a module loader (at run-time!) that first checks for a new format, and falls back to the standard loader otherwise. If the new loader detects an object file of the new format, it instead does the loading itself and follows the conventions of the standard loader to install the module into the module list.

Using implicit module coupling works, but tends to require re-programming large portions of the extended service. Also, it has all the problems of implicit module coupling (2.4.1). One way to improve the situation is to refine the modules Objects and Modules into frameworks of modules and types. Then, the new modules constituting the implementation details of Objects and Modules would be considered private (i.e. are not forced to provide system-wide safe interfaces). Thereby, the internal details could be provided in a structured and typeful manner, supporting extensions on that level. This approach has been chosen in the Choices operating system [CJM*89], which mimics fairly conventional operating systems but has its structures decomposed into object-oriented frameworks.

3.4.4 Preemption – Tasks, Coroutines, Threads, Processes

A rather old issue of operating system design is the support of pseudo-parallel execution. Originally, the goal was to design single-processor time-sharing systems simultaneously usable by many users (e.g. [Org72]). Each user should have the illusion of possessing an individual, slower machine. The construct providing such a "virtual processor" is called *process*. Processes are completely *isolated* from each other: A standard process – as in UNIX [Bac86] – carries its own address space, its own stack, its own processor state, its own logical device assignments, and so on. Two processes executing on the same machine thus can impose nearly no *side-effects* on each other. (The exception being shared files and perhaps other shared system resources.) This imposes significant costs on process creation and process switching, also called context switching.

Later, the use of quasi-parallel execution of a single program was found an appropriate structuring tool (e.g. [Hoa84]). By having multiple processes collaborating for a single user or application the cost of individual processes and that of process scheduling and context switching became much more critical. Hence, the introduction of more "light-weight" quasi-parallel constructs was sought. While all of them differ significantly in their actual concurrency semantics, the common idea is to reduce the amount of context associated with each process. By sharing a single address space, i.e. by sharing global data, a more light-weight construct called *thread* (also called *light-weight process*) results. Many modern operating systems support threads. Often a set of threads executes in a shared address space, and the set of threads together with the address space form a process (e.g. the Mach Kernel [ABB*86]).

By removing the property of automatic context switching (preemption), and thereby most of the thread-specific processor state, threads reduce to *coroutines* [Wir84]. A different alternative is to remove the stack and to stay with preemption. This new construct is called *engine* to emphasize that it is the driving part of a parallel state *machine* (see below). Finally, by removing all state information (except for the existence of an almost empty task descriptor), one arrives at *(non-preemptive) tasks*. Figure 3.17 illustrates the relation between these different constructs.

The various possible combinations of constructs like Coroutines or Engines with separate address spaces, or separate bindings to global resources have not been considered meaningful, as the resulting weight does not justify removal of any of the other, far more light-weight attributes.

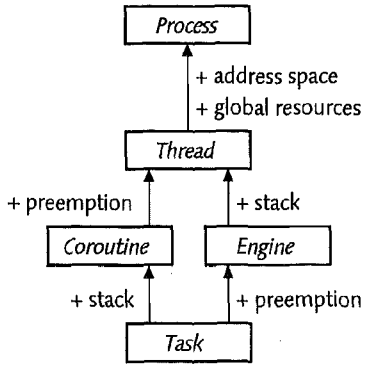


Figure 3.17 – Relation Between Various Tasking Concepts.

The benefits of all constructs above are various levels of relaxation of the sequential programming model. As long as preemption is avoided, i.e. as long as context switches happen at points clearly controlled by the programmer, few problems result. This is true for non-preemptive tasks and for coroutines. The resulting style of programming is often called *cooperative multi-tasking*. It is the dominating programming model in the Oberon system, but also in the Macintosh OS [App85].

Further relaxation of the sequential model has its costs. Either one has to pay for the high overhead of true processes providing the illusion of multiple, mostly independent and sequential executions. Or one has to pay the price of introducing problems of non-sequential execution, i.e. the danger of uncontrolled side-effects resulting from arbitrarily interleaved execution. To cure the effects of the latter, many secondary constructs have been proposed. The two most important ones being *Semaphores* [Dij65] and *Monitors* [Hoa74]. Recent developments focussed on encapsulating the problems of thread communication and synchronization into so-called parallel object-oriented languages (e.g. [Ame87]). The basic concept is to allow some kind of non-blocking message send. In its purest form this leads to actor languages [AH87], a survey of various approaches may be found in [KL89]. Often, the tendency is to uniformly treat the various multi-*processor* and distribution architectures, where true parallelity exists and should be exploited.

Decisions taken for Ethos. Ethos follows Oberon in being primarily designed to execute on a single-user workstation. For Oberon, exploitation of parallelism was not considered important. The Oberon system avoids preemption, i.e. except for interrupt-handlers all code is executed strictly sequentially, and only non-preemptive tasks are supported. Also, it is quite easy to add coroutines

usable while a command is executing. This has been used successfully to write simple simulation software. The lack of allowing garbage collection while stacks may contain valid pointers, and the impossibility to provide coroutine stacks that are protected against overflows render the coroutine packages less useful.

Experience with networking software [Szy90a] exhibited a weakness in the non-preemptive design of Oberon. In particular, the requirement that a local workstation be responsive to the network while at the same time executing some time-consuming task caused difficulties. Currently, a project is underway investigating the possibility of introducing threads to the Oberon system.

For Ethos, a far simpler approach was taken: Besides supporting non-preemptive tasks as found in the Oberon system, Ethos adds *preemptive tasks* (*engines*). An engine is particularly light-weight in that it does not require its own stack: Engines preempt the main process and execute on its stack, but engines never preempt other engines of same priority. (Currently, Ethos supports only a single priority for all engines.) The idea is to isolate time-critical activities and implement them using engines while staying with the purely sequential programming model for the majority of the code. In a sense, an engine can be considered to be an interrupt handler handling some *virtual interrupt*. The engine scheduler guarantees invocation of all engines staying in a ready state at a fixed frequency (if engines cooperate). Therefore, engines can be used to implement activities bound by real-time constraints in a simple way. The typical approach is to implement some kind of state machine that uses an engine to perform transitions. For example, the Ethos profiler uses an engine to monitor the main process and creates statistics of relative frequencies of executed procedures.

The tasking abstractions are defined in module Tasks. The following excerpt shows the important details.

```
DEFINITION Tasks;  ...
   TYPE
      Task = POINTER TO TaskDesc;
      TaskDesc = RECORD (Objects.ObjectDesc)
         base–: Scheduler; link–: Dispatcher; safe–: BOOLEAN; timer–: Input.Timer;
         PROCEDURE (T: Task) ConnectTo
            (base: Scheduler; link: Dispatcher; safe: BOOLEAN; timer: Input.Timer);
         PROCEDURE (T: Task) Do;
         PROCEDURE (T: Task) Resume;
         PROCEDURE (T: Task) Suspend;
         PROCEDURE (T: Task) Sleep (dt: LONGINT);
         PROCEDURE (T: Task) State (): LONGINT;
      END;
```

```
Dispatcher = POINTER TO DispatcherDesc;
DispatcherDesc = RECORD (Objects.ObjectDesc)
   PROCEDURE (D: Dispatcher) Resume;
   PROCEDURE (D: Dispatcher) Sleep (dt: LONGINT);
   PROCEDURE (D: Dispatcher) State (): LONGINT;
   PROCEDURE (D: Dispatcher) Suspend;
END;

Scheduler = POINTER TO SchedulerDesc;
SchedulerDesc = RECORD (Objects.FinObjectDesc)
   enabled–, preemptive–: BOOLEAN;
   PROCEDURE (S: Scheduler) Init (preemptive: BOOLEAN);
   PROCEDURE (S: Scheduler) SetControl (enable: BOOLEAN);
   PROCEDURE (S: Scheduler) SetTask (t: Task; safe: BOOLEAN);
   PROCEDURE (S: Scheduler) Step;
END;

VAR
   pre, stdPre–, idle, stdIdle–: Scheduler;
```

The organization follows the generalized Carrier/Link/Rider Scheme. At the bottom, the actual scheduling discipline is provided by a scheduler (*Carrier level*). At the top, the application-specific activity to be scheduled is implemented as an instance of (a subtype of) type Task (*Rider level*). The N:1 relation between tasks and schedulers and the bottleneck interface is realized by instances of type Dispatcher (*Link level*). The two standard scheduling disciplines (preemptive and idle) are available via variables *pre* and *idle*, respectively. As usually, the variables *stdPre* and *stdIdle* allow access to the standard implementations of these.

A scheduler is driven by calls to its Step method. For the idle scheduler this is done by the main loop, while for the preemptive scheduler Step is invoked from a timer interrupt handler (see below). A scheduler may be disabled (SetControl).

A new task is set to a particular scheduling discipline by means of a scheduler method SetTask which in turn informs the task by calling ConnectTo. The Do method of a task is periodically invoked as long as the task is ready. The task can be suspended at any time and resumed later. (A suspended task is subject to garbage collection as soon as it becomes unreachable.) Also, it is possible to set a task asleep for a certain amount of time. The latter possibility is mainly used to reduce the load of the system. The task methods Resume, Suspend, Sleep, and State all forward to the corresponding Dispatcher bottleneck. (As generally done in the Carrier/Link/Rider scheme, the link object is created by the carrier on invocation of a Set* method. The dispatcher object carriers information the

scheduler requires to schedule a set task.)

There are no special signalling mechanisms, hence tasks are required to poll their enabling conditions, e.g. by inspecting a global variable. (A signalling scheme could be easily added as an extension.)

Preemptive tasking is based on the possibility of installing a scheduler in a timer. Timers are defined in module Input and have two characteristic values: *unit* and *grid*. *Unit* defines the resolution of the Time method, i.e. the precision of the timer in ticks per second, while *grid* defines the number of ticks (measured in units) between two scheduler invocations. Since the used timer is known to a task, the task can adjust its own behavior to real-time conditions.

```
DEFINITION Input;  ...
   TYPE
      Scheduler = PROCEDURE (pc: LONGINT);
      Timer = POINTER TO TimerDesc;
      TimerDesc = RECORD (Objects.ObjectDesc)
         unit–, grid–: LONGINT;
         schedule–: Scheduler;
         PROCEDURE (T: Timer) Init (S: Scheduler; unit, grid: LONGINT);
         PROCEDURE (T: Timer) Time (): LONGINT;
      END;

   VAR
      timer, stdTimer–: Timer;
```

Monitors. For engines to be used safely, it is important that preemption can be controlled. A simple but efficient approach has been taken by allowing individual modules to be registered as *monitors*. In fact, code executing within a monitor module will never be preempted. (Careful use of monitors is required though.) The scheme could be extended easily by introducing priorities for engines and only preventing preemption by engines that have a lower priority than the monitor. This has not been done and the simple monitor construct provided sufficed in all cases considered so far.

```
DEFINITION Tasks;  ...
   PROCEDURE Protect (m: Modules.Module; hard: BOOLEAN);
   PROCEDURE Unprotect (m: Modules.Module);
   PROCEDURE GetStateOf (m: Modules.Module; VAR monitor, hard: BOOLEAN);
```

By declaring a module to be a *hard* monitor, unprotecting the module is disallowed. Otherwise, it is possible to turn a module temporarily into a monitor by means of paired Protect and Unprotect invocations. The foundation modules Devices, Objects, Files, and Modules, as well as module

Tasks itself are predeclared (hard) monitors.

Multiple Stacks. The *possibility* of having multiple stacks should not be blocked by the system design. Hence, the garbage collection mechanism tracing references from the stack (module Objects, cf. 3.4.2) has been slightly generalized to support special objects, called reference blocks, that are scanned for potential references. In principle, the standard stacks are predefined reference blocks. (RefBlocks are finalizable objects, hence there is a registering but no unregistering facility.)

```
DEFINITION Objects;  ...
   TYPE
      RefBlock = POINTER TO RefBlockDesc;
      RefBlockDesc = RECORD (FinObjectDesc)
         PROCEDURE (R: RefBlock) GetBounds (VAR lo, hi: LONGINT);
         (*the range [lo, hi) is scanned for pointer candidates*)
      END;

      HeapManagerDesc = RECORD (ObjectDesc)
         ...
         PROCEDURE (H: HeapManager) RegisterRefBlock (rb: RefBlock);
      END;
```

In Section 3.5.3 the usefulness of the engine abstraction together with reference blocks will be illustrated by showing how to add threads to the system without changing any of its existing parts.

3.4.5 Files, Streams, and Object Externalization

One of the most crucial parts of an operating system is its file system. The performance of the file system, the elegance and simplicity of its interface, and the coherence of the provided semantics influence almost all other parts of the system. One of the clear advantages of the Oberon file system is its simple interface combined with its sufficiently powerful semantics. The performance of the Oberon file system, as implemented on the Ceres machine, has proven to be highly acceptable for most applications. (The performance mostly depends on the speed of the disk, and the Ceres disk subsystem is rather slow.) Implementations of the Oberon system on other machines provide file systems that delegate operations to the underlying operating system. For ports to UNIX machines with fast disk subsystems the provided performance was found more than adequate. Hence, it is felt that the simple interface of the Oberon file system is an ideal bottleneck interface in the sense of the Carrier/Link/Rider separation scheme.

The Oberon system has its file system "burned in", i.e. it is not possible to extend it by any means. This has been done intentionally so to allow for acceptable performance on slow machines. In Ethos the principle design decision to be open on all levels applies to the file system level as well. Hence, Ethos provides a replaceable file directory object that returns file objects and maintains the file directory.

3.4.5.1 Sequential and Positionable Streams

To explore all benefits of the Carrier/Link/Rider scheme, the mechanism is actually rooted on a level below files: General byte streams are defined in module Objects. Derived from that are arbitrarily positionable streams. Files then are just a special implementation of such positionable streams. (Texts are another example of positionable streams, cf. 3.4.6.) The standard stream rider interface has been chosen to be able to read and write all basic types of Oberon-2. Thus, iteration over components of constructed types suffices to map any structure to a stream and vice versa. (How to deal with extensible structures in a convenient way is explained below.)

```
DEFINITION Objects;  ...
   TYPE
      StreamRider = RECORD
         base–: Stream; link–: StreamLink; eos: BOOLEAN; res: LONGINT;
         PROCEDURE (VAR R: StreamRider) ConnectTo (base: Stream; link: Object);

         PROCEDURE (VAR R: StreamRider) Available (): LONGINT;
         PROCEDURE (VAR R: StreamRider) Pos(): LONGINT;
         PROCEDURE (VAR R: StreamRider) Read (VAR x: CHAR);
         PROCEDURE (VAR R: StreamRider) ReadBlock
            (VAR x: ARRAY OF CHAR; beg, len: LONGINT);
         PROCEDURE (VAR R: StreamRider) SkipBlock (len: LONGINT);

         PROCEDURE (VAR R: StreamRider) ReadShort (VAR n: SHORTINT);
         PROCEDURE (VAR R: StreamRider) ReadInt (VAR n: INTEGER);
         PROCEDURE (VAR R: StreamRider) ReadLong (VAR n: LONGINT);
         PROCEDURE (VAR R: StreamRider) ReadReal (VAR x: REAL);
         PROCEDURE (VAR R: StreamRider) ReadLongReal (VAR x: LONGREAL);
         PROCEDURE (VAR R: StreamRider) ReadString (VAR s: ARRAY OF CHAR);
         PROCEDURE (VAR R: StreamRider) ReadBool (VAR b: BOOLEAN);
         PROCEDURE (VAR R: StreamRider) ReadSet (VAR s: SET);

         PROCEDURE (VAR R: StreamRider) Write (x: CHAR);
         PROCEDURE (VAR R: StreamRider) WriteBlock
            (VAR x: ARRAY OF CHAR; beg, len: LONGINT);

         PROCEDURE (VAR R: StreamRider) WriteShort (n: SHORTINT);
         PROCEDURE (VAR R: StreamRider) WriteInt (n: INTEGER);
```

```
    PROCEDURE (VAR R: StreamRider) WriteLong (n: LONGINT);
    PROCEDURE (VAR R: StreamRider) WriteReal (x: REAL);
    PROCEDURE (VAR R: StreamRider) WriteLongReal (x: LONGREAL);
    PROCEDURE (VAR R: StreamRider) WriteString (s: ARRAY OF CHAR);
    PROCEDURE (VAR R: StreamRider) WriteBool (b: BOOLEAN);
    PROCEDURE (VAR R: StreamRider) WriteSet (s: SET);
    PROCEDURE (VAR R: StreamRider) WriteLn;
END;


StreamLink = POINTER TO StreamLinkDesc;
StreamLinkDesc = RECORD (ObjectDesc)
    PROCEDURE (S: StreamLink) Available (VAR r: StreamRider): LONGINT;
    PROCEDURE (S: StreamLink) Pos (VAR r: StreamRider): LONGINT;
    PROCEDURE (S: StreamLink) Read (VAR r: StreamRider; VAR x: CHAR);
    PROCEDURE (S: StreamLink) ReadBlock
        (VAR r: StreamRider; VAR x: ARRAY OF CHAR; beg, len: LONGINT);
    PROCEDURE (S: StreamLink) SkipBlock (VAR r: StreamRider; n: LONGINT);
    PROCEDURE (S: StreamLink) Write (VAR r: StreamRider; x: CHAR);
    PROCEDURE (S: StreamLink) WriteBlock
        (VAR r: StreamRider; VAR x: ARRAY OF CHAR; beg, len: LONGINT);
END;


Carrier = POINTER TO CarrierDesc;
CarrierDesc = RECORD (FinObjectDesc) END;

Stream = POINTER TO StreamDesc;
StreamDesc = RECORD (CarrierDesc)
    PROCEDURE (S: Stream) SetRider (VAR r: StreamRider);
    PROCEDURE (S: Stream) Flush;
END;

PosStream = POINTER TO PosStreamDesc;
PosStreamDesc = RECORD (StreamDesc)
    PROCEDURE (S: PosStream) SetRiderAt (VAR r: StreamRider; pos: LONGINT);
    PROCEDURE (S: PosStream) Length (): LONGINT;
END;
```

The Stream Bottleneck Interface. The canonical Carrier/Link/Rider separation is manifest in the triples Stream / StreamLink / StreamRider and PosStream / StreamLink / StreamRider, respectively. The abstract StreamRider has to be subclassed in order to implement concrete mappings from basic types to sequences of bytes and vice versa. The N:1 relation between riders and streams and the bottleneck interface is realized by instances of type StreamLink. Obviously, the bottleneck interface for streams is simple: In principle, it supports reading and writing of individual bytes. Indeed, it would be most elegant to support nothing else. Alas, for performance reasons it is

necessary to also support *block operations*. Using block operations it is possible to transfer almost arbitrary numbers of bytes by means of a single invocation. Since the byte stream interface is most critical to the overall system performance this traditional tuning measure was felt necessary.

The knowledgable reader may compare the Oberon and Ethos byte block operations. While the Oberon file system provides signatures like (... VAR x: ARRAY OF SYSTEM.BYTE; len: LONGINT), Ethos stream operations have signatures of the form (... VAR x: ARRAY OF CHAR; beg, len: LONGINT). There are two noteworthy differences.

Firstly and trivially, the additional *beg* parameter adds more flexibility when implementing scatter-gather schemes in order to avoid double-buffering. As an example, consider the need to write bytes starting at index 5 of an array. Without the *beg* parameter one either has to use single-byte writes, or one has to first copy the required portion into an auxiliary array. (Admittedly, in most cases the actual parameter is the constant zero.) Secondly and more subtle, the use of SYSTEM.BYTE has been avoided. For a discussion of the problems with SYSTEM.BYTE, cf. 2.4.1.

3.4.5.2 Object Externalization and Internalization

When mapping primitive data types to a stream of bytes it is sufficient to use mapping methods "hard-wired" to the used rider. The task is more difficult when the mapped data structure contains arbitrary objects. The reason is that such objects may actually be of any subtype of the type known to the mapping code. The mapping of an object to a stream of bytes is called *externalization*. The inverse mapping of a stream of bytes into an object is called *internalization*.

To externalize an object it is essentially sufficient to have an associated Externalize method. Internalization is by nature a bit more complicated, as it requires creating the object, which of course cannot be performed by a method of that very object. A typical solution is to write out some type information that upon internalization is used to allocate the required object. One can either use the name of a command that allocates objects of the appropriate type, or, more elegantly, one can use the run-time type information attached to every object and use it to write out the actual type name. Then the type name can be used during internalization to create an instance of that type. The major advantage of the latter approach is its potential for generic automation, i.e. it can be programmed once and for all, supporting all possible objects.

The command approach is typically used in the Oberon system, e.g. in the Write [Szy91] and Draw [WG92] editors. (In newer Oberon versions a module Types exists that provides about the same type meta-information as available in Ethos, cf. 3.4.3.3.) Instead of commands, Ethos uses the type-driven approach. The roots for this mechanism are defined in module Objects.

```
DEFINITION Objects;  ...
   TYPE
      Mapper = RECORD (StreamRider)
         org, len: LONGINT;
         PROCEDURE (VAR R: Mapper) OpenMap;
         PROCEDURE (VAR R: Mapper) ReadObject (VAR obj: Objects.Object);
         PROCEDURE (VAR R: Mapper) ReadCarrier (VAR c: Objects.Carrier);
         PROCEDURE (VAR R: Mapper) WriteObject (obj: Objects.Object);
         PROCEDURE (VAR R: Mapper) WriteCarrier (c: Objects.Carrier);
         PROCEDURE (VAR R: Mapper) CloseMap;
      END;

      Carrier = POINTER TO CarrierDesc;
      CarrierDesc = RECORD (FinObjectDesc)
         PROCEDURE (C: Carrier) Externalize (VAR r: Mapper);
         PROCEDURE (C: Carrier) Internalize (VAR r: Mapper);
      END;
```

Riders of type Mapper are used to read and write objects; refined support is available for objects that are actually carriers. For plain objects only the information is handled that is required to allocate new objects upon internalization. Additionally, if a carrier is used, the methods Internalize and Externalize are invoked, passing the mapper to the carrier in order to support recursive handling of data structures. For a concrete carrier class the Internalize and Externalize methods have to be overridden to actually read or write the individual fields. (Note: The cyclic interrelation of abstract carriers, abstract streams, and abstract mappers is the reason for putting all these abstract definitions into the single module Objects.)

Systems exist that support automatic externalization and internalization of complete data structures. An example is the "Pickle" system that is part of the Modula-3 library [Nel91]. There, run-time type information is used to automatically deal with fields of arbitrary objects. For many trivial applications the use of such a system allows certain simplifications. In practice, however, it is often required to use a more selective approach. Often, an object contains redundant information used to tune its performance, but not expected to be mapped to external representations. Also, it is quite usual that fields should be coded in some non-generic way to achieve compact representations. To cope with such demands, Pickles allows to register special externalization and internalization procedures that in turn override the default behavior.

In Ethos the opposite position was taken. Instead of providing a default which is often replaced, it is always required to provide the right externalization and internalization methods from the beginning. (As an extension, a class may be implemented that implements fully generic externalization and internalization for all its subclasses.)

3.4.5.3 Dealing with Aliens

The purpose of externalization and internalization is to decouple the creators and clients of data structures over time, space, or architectures. Decoupling over time is the basic concept of a file system: An arbitrary time may elapse between writing a file and reading it in again. Decoupling over space is what happens when using external representations to send or distribute data to other places. Finally, decoupling over architectures allows a sufficiently architecture-independent external form to be used to transfer data between different machine or software architectures.

In all these cases the fact that one deals with extensible systems adds a new potential for problems. When externalizing using a fixed format it can be expected that internalizing will either work or fail completely. However, if the externalization is based on a generic treatment of extensible data-structures, i.e. one where extension modules control particular externalizations, internalization may fail *partially*. The reason is that it is well possible that some of the extension modules required for internalization are not available. Either, because the modules have never been made available to the internalizing system, or because the available modules are subject to some version conflict preventing their installation.

If an external form contains the representation of an object that is not internalizable, such an object is called an *alien*. Three ways of dealing with aliens exist. Firstly, one might cancel the whole internalization process when a single alien is detected. This makes external forms vulnerable to configuration problems. Secondly, alien objects may be detected and reduced upon internalization. Here, object reduction means either discarding an object or projecting it to some base type that happens to be available. The resulting internal form is thus reduced compared to the full external form: Re-externalizing the structure leads to loss of information. Thirdly, alien objects may be kept in an abstract form. If this is done, re-externalization re-establishes the information that was there in the original external form.

The first alternative is considered rather problematic when designing extensible systems: The probability of having all existing installations in mutually fully consistent configurations decreases rapidly with the number of installations and even faster with the number of orthogonal extension options. It seems a valid choice only in situations where a missing object prevents meaningful continuation. The latter two alternatives are more likely to be considered when using generic externalization and internalization. However, which of the possibilities is best depends on the particular application. An example for the third choice is the text system covered by the next section. (There, extensible objects may float within a text, and aliens are kept as black-box objects. Storing a text containing such black-box objects and restoring it when the corresponding extension modules are available, the

original text can be recovered.)

For Ethos it was decided to support only the basic building blocks for supporting various application-level alien handling schemes. Each time a mapper finds an object that cannot be internalized, it returns (in its fields *org* and *len*) the origin and length of the object's representation in the source stream; otherwise it returns the object (and *len* = 0). Then, the mapper skips the alien block and can be used to proceed internalizing from the rest of the stream. In the client code using a mapper it is decided whether the alien will be discarded, partially reduced, abstractly encapsulated, or whether internalization is to be aborted.

Properly dealing with alien objects is considered an absolute necessity of an extensible system. Hence, it was felt acceptable to pay a certain price for being able to deal with aliens. The price paid for the Ethos externalization scheme is the restriction to *positionable* streams. In order to skip alien blocks in a stream, the internalization mechanism needs to know the length of blocks taken by particular objects. Since during externalization there is no easy way to compute these lengths in advance, a back-patching approach is taken that reserves a length field in front of each object block. After completing a block this field is adjusted to the number of bytes that the block actually took.

> Mappers check whether the stream they are set on is indeed positionable and cause a run-time exception otherwise. This is one of the few covariance conflicts in the Ethos interfaces: To be detected by the type system, it would be necessary to override method ConnectTo(base: Stream; ...) of type StreamRider by a method ConnectTo(base: PosStream; ...) for type Mapper. This is not allowed and would be unsafe as long as there is no run-time type-check introduced by the compiler. Type-systems that allow this kind of signature modification for overriding methods are said to support *covariant parameter types*.; cf. 1.1.1 and 4.4.3.

A more subtle consequence of handling aliens properly is the recursive externalization of objects. Obviously, if an object has some part objects, these cannot be internalized in a meaningful way, if the owning object cannot, cf. Figure 3.18. For example, one might consider a dictionary object that contains entries of word pairs. If the dictionary is an alien, then none of the entry objects can be internalized.

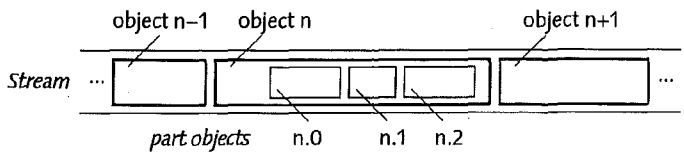


Figure 3.18 – Sequentialized Part Objects.

If object *n* cannot be internalized, so part objects *n.0*, *n.1*, and *n.2* cannot. For each object, there is a prolog generated by the generic externalization mechanism. It contains the type information and the length of the block taken by the object. In the case of object *n*, the length in the prolog includes the lengths of the part objects *n.0*, *n.1*, and *n.3*.

A standard implementation of an object mapper supporting portable external representations of all basic types is implemented in module Stores. Stores provides a subtype of Objects.Mapper that implements portable mappings for all standard types and type-driven, alien-aware mapping of objects. The type information is maintained in a dictionary such that multiple occurrences of objects of the same type can be coded efficiently. Of course, dictionary information for part objects is not available when the enclosing object is an alien. Hence, dictionaries are nested in a stack fashion by objects externalizing and internalizing their part objects. The following code fragment gives a typical example, assuming that objects of type MainObj will want to drop alien part objects of a subtype of PartObj.

```
TYPE
   PartObj = POINTER TO PartObjDesc;
   PartObjDesc = RECORD (Objects.CarrierDesc)
      ...
   END;
   MainObj = POINTER TO SampleObjDesc;
   MainObjDesc = RECORD (Objects.CarrierDesc)
      part: PartObj   (*may hold instances of arbitrary subtypes of PartObj*)
   END;

PROCEDURE (M: MainObj) Externalize (VAR r: Objects.Mapper);
BEGIN r.OpenMap; r.WriteCarrier(M.part); r.CloseMap
END Externalize;

PROCEDURE (M: MainObj) Internalize (VAR r: Objects.Mapper);
BEGIN r.OpenMap; r.ReadCarrier(M.part); r.CloseMap
   (*M.part = NIL if part-object does not exist or is alien*)
END Internalize;
```

By invoking methods OpenMap and CloseMap the mapper is informed to open a new dictionary and to close it again, thereby proceeding to use the previously active type dictionary. The only requirement is that the internalization code is written completely symmetrical to the externalization code.

Things change a little when an application should deal more specifically with aliens. In the code example above, aliens will just be discarded, i.e. the corresponding ReadCarrier invocation will return NIL with a non-zero value in the *len* field of the mapper. The following code creates a special closure object

that keeps the alien as a black-box and stores it again when re-externalizing the main object. For the sake of simplicity it is assumed that the stream that contained the original external form is retained as long as some reference to it exists. (For files this is the case; for other externalization streams it might be necessary to copy the data from the stream into some buffer.) Under these simplifying assumptions it is easy to use the state of the closure object to re-create the external form of the alien upon externalization.

```
TYPE
   Closure = POINTER TO ClosureDesc;
   ClosureDesc = RECORD (PartObjDesc)
      s: Objects.PosStream; org, len: LONGINT
   END;

PROCEDURE (M: MainObj) Externalize (VAR r: Objects.Mapper);
   VAR r1: Files.Rider; part: PartObj; len: LONGINT; ch: CHAR;
BEGIN
   r.OpenMap; part := M.part;
   WITH part: Closure DO len := M.len; part.s.SetRiderAt(r1, part.org);
      WHILE len > 0 DO r1.Read(ch); r.Write(ch); DEC(len) END
   ELSE r.WriteCarrier(part)
   END;
   r.CloseMap
END Externalize;

PROCEDURE (M: MainObj) Internalize (VAR r: Objects.Mapper);
   VAR c: Closure;
BEGIN
   r.OpenMap; r.ReadCarrier(M.part);
   IF (M.part = NIL) & (r.len > 0) THEN (*enclose alien object*)
      NEW(c); c.s := r.base(Objects.PosStream); c.org := r.org; c.len := r.len;
      M.part := c
   END;
   r.CloseMap
END Internalize;
```

A common pitfall is to believe that one could simply attach an appropriate externalization method to the closure type. Then it seems possible to avoid the explicit treatment of alien closures within the externalization method of the main object. This is not so, however, as the generic externalization mechanism would write out the type information of the alien closure and not that of the enclosed alien!

3.4.5.4 The File System

A concrete implementation of positionable streams is contained in module Files. Files also contains a standard concrete StreamRider implementation that maps all basic types into byte sequences using the most efficient coding of

the used machine. (The resulting mapping is not portable. Hence, it is usually used for files such as object-files, which are inherently non-portable.) Finally, Files defines an abstract file directory, the current directory object, and a default directory object. The latter always refers to the standard file directory implemented in Files.

```
DEFINITION Files;  ...
   CONST
      Insert = 0; delete = 1; replace = 2;   (*directory notification codes*)
   TYPE
      FileName = ARRAY 32 OF CHAR;
      PathName = ARRAY 64 OF CHAR;

      FileInfo = POINTER TO FileInfoDesc;
      FileInfoDesc = RECORD (Objects.ObjectDesc)
         next: FileInfo;
         length, date, time: LONGINT;
         name: FileName
      END;

      DirNotifyMsg = RECORD (Objects.NotifyMsg)
         dir: Directory; op: INTEGER; name: PathName
      END ;

      Directory = POINTER TO DirectoryDesc;
      DirectoryDesc = RECORD (Objects.ObjectDesc)
         notify: BOOLEAN;
         res: INTEGER;

         PROCEDURE (D: Directory) New (): File;
         PROCEDURE (D: Directory) This (name: ARRAY OF CHAR): File;
         PROCEDURE (D: Directory) Register (F: File; name: ARRAY OF CHAR);

         PROCEDURE (D: Directory) GetDir
            (prefix: ARRAY OF CHAR; full: BOOLEAN): FileInfo;
         PROCEDURE (D: Directory) Rename (old, new: ARRAY OF CHAR);
         PROCEDURE (D: Directory) Delete (name: ARRAY OF CHAR);

         PROCEDURE (D: Directory) GetInfo (VAR alloc, free: LONGINT);
      END;

   VAR
      dir, stdDir–: Directory;
```

The file directory can create new files (method New), retrieve existing files by name (This), and register a previously created file using a particular name (Register). The directory can be retrieved as a whole (GetDir), which causes the directory to be copied out into a list. GetDir may be constrained by adding

a non-empty prefix. Normally, GetDir returns names only. By specifying *full* information, GetDir also returns file lengths and dates/times of last modification. Of course, entries can be renamed or deleted. Finally, it is possible to acquire some information about the allocation state of the underlying device (GetInfo). The result code of the latest operation is reflected by the directory field *res*. By specifying a file directory to *notify*, notification messages are sent upon insertion, deletion, or replacement of directory entries (cf. 3.4.8.4).

As has been noted Section 2.3.3.1 introducing the Carrier/Link/Rider concept, the name "rider" was originally borrowed from the Oberon file system. Hence, it is interesting to compare the Ethos and Oberon file systems to clarify the different notions of "rider". In Oberon, a rider is a record structure mostly private to the file system implementation. It encapsulates the current state of access, i.e. the position in a file, a hint to a buffer, etc. Thus, an Oberon rider corresponds to an Ethos link object. However, an Oberon rider is declared and allocated by the client of the file system, while an Ethos link is allocated by the corresponding stream object.

On the other hand, an Oberon rider is used as an access medium to a file. Therefore, it also has similarities with an Ethos rider. The principal difference is that an Oberon rider works *only* in conjunction with files provided by the Oberon file system, while an Ethos rider works on every possible stream. For that reason the Oberon rider's role is played by the combination of an Ethos rider and an Ethos link.

File Buffering.

Since there is an increasingly significant access-time gap between primary and secondary stores, an effective file caching scheme has a major impact on the overall system performance. As in the Oberon system, Ethos maintains file buffers on the level of file objects. Each file object has a list of buffers attached to it. A physical file is organized into a sequence of pages, usually corresponding to sectors of the secondary storage device. A buffer holds the contents of one of these pages, plus the page number of the cached page.

Link objects contain a hint (pointer), which buffer should be used. Whenever a file access is performed, it is checked whether the hint is still valid, i.e. whether the buffer referred to by the link contains the expected page number. If so (*cache and hint hit*), the file access is performed by reading or writing the buffer; when a buffer is written to it is marked modified. If the buffer hint fails, it is checked whether any of the other buffers belonging to the file contains the required page. If so (*cache hit*), the hint is adjusted and the found buffer used. Finally, if the required page is not currently in any of the buffers (*cache miss*), the corresponding sector is sought and loaded into a new buffer. To prevent consumption of an unlimited number of buffers, a replacement strategy is used whenever the number of buffers used by a file reaches a certain limit. In this case, one of the existing buffers is taken, flushed to disk, if marked modified, and over-written by the requested page.

Other than in Oberon, the buffering strategy is chosen to be uniform, i.e. it

is not tried to detect certain access patterns – like sequential scan – in order to adjust the buffer replacement strategy. Instead, the system allocates new buffers until reaching the limit of buffers per file. Then all further buffer requests are handled by cyclic replacement. Thus, if a file is accessed sequentially, the last kN bytes are kept in the file buffer, where k is the maximum number of buffers per file, and N is the size of a buffer. Thus, small files are kept in memory after being read the first time. Also, many applications are effectively based on "file windows". For example, a file based text editor will exhibit a certain locality by clustering accesses within the currently edited range.

File Releasing.
Whenever a file object is unreachable it is subject to finalization, i.e. the garbage collector will discard it eventually. As long as a file is registered in the file directory, the external resources bound to a file, i.e. the disk sectors used, remain reachable and must be retained. However, when finalizing a file that is not (or no longer) registered in the file directory, the external resources of the file could be released. Likewise, when deleting a file from the file directory and when there are no remaining references to the file object, the sectors used by the file could be released. If these actions are taken automatically, it can be said that garbage collection extends to the disk.

The Oberon file system provides a dangerous alternative. Here, a file may be explicitly *purged*, whereby the programmer declares that the file is no longer used and its sectors should be released. If the programmer errs and purges a file too early, a *dangling reference* results; if the programmer follows a too conservative approach, a *memory leak* like situation follows. To avoid such hazardous situations, the Ethos file system *does not provide* a purge operation. Instead, the general finalization strategy is used to release unused disk sectors. The only significant increase in costs involves the file delete operation, which in practice is used rather infrequently.

> The question may arise why it is necessary at all to collect disk sectors. At least, whenever the system is booted, a disk reservation bitmap is created by scanning the directory, effectively releasing unused sectors. Experiences with Oberon show that the typical once-a-day booting of the system is more than sufficient to prevent exhaustion of disk space. However, the situation is different when operating a continuous service, as typical for remote servers. Then, consuming disk sectors without ever releasing them *guarantees* that the server will eventually run out of disk space.

File Directory.
The standard file directory is implemented using a B-Tree [BM72]. A global array of page buffers is used as a *level-cache*. Hence, accessing the same B-Tree page several times in a series of operations is optimized. Also, accessing file names close to each other in an alphabetic ordering is optimized.

As a neat side-effect the implementation of the B-Tree got simplified compared to an implementation caching pages directly within procedure activation records. Since the recursive B-Tree operations use pointers to pages in the level-cache, it is easy to deal with symmetric cases by first swapping pointers. If pages are directly kept on the activation stack, swapping pages is too expensive.

3.4.6 The Text System

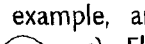
The Ethos text system has undergone several revisions. After some early experimental versions, it reached a state where its usefulness became clear. As a result, the Ethos text system got ported to the Oberon system, where it lost some of its flexibility inherent in the Ethos system, but also got matured to become a proven usable sub-system of Oberon. A result of this effort was the Write document editor [Szy92a] for Oberon, and a large set of extensions provided by Write users. Later, the new text system got completely integrated into Oberon, replacing the original text system by an upward compatible one. As soon as the Write editor and the underlying text system had stabilized within Oberon, they have been ported back to Ethos, where certain generalizations were again applied. – This thesis has been prepared using Write. The Oberon version was used to avoid the need for porting the many existing extensions to Ethos.

An Ethos text is an *attributed positionable stream.* Module Texts defines the abstract interface to texts, and provides – by means of a directory object – access to a standard text implementation. All riders connectable to general streams may be used with texts. The common case of transforming basic types into readable representations using the ASCII character set is covered by a rider implementation exported by Texts. Also, Texts exports an extension of that rider class, supporting access to text specific attributes. The latter rider type is constrained to streams of (a subtype of) type Texts.Text.

A detailed coverage of the text system and its extension model may be found in [Szy92a], also presenting a wide variety of extensions. [Szy91] adds discussions on certain design decisions. The following gives a concise outline and summarizes the important ideas.

Conceptually, a text is considered a *sequence of attributed text objects.* The most important text objects are normal characters. A text object may also be an instance of a class. Associated attributes are a font (combining family, size, and style), a color, and a vertical displacement. An EBNF syntax describes the abstract text model:

```
Text = {AttrObject}.
AttrObject = Attributes TextObject.
Attributes = font color offset.
TextObject = characterObject | extendedObject.
```

There are intentionally no structures superimposed on a text. It will be shown below how to incorporate range attributes (like paragraph formatting styles) without changing the text model. For the sake of simplicity, from now on, *character* will be used for "characterObject", and *element* for "extendedObject". For example, an element may be a graphics floating in a text (e.g. ). Elements have been used all over this thesis, in form of pictures, graphics, tables, and the like.

The simple text model leads to a small set of elementary editing operations defined on texts, like deleting an arbitrary subrange of the text. The unconstrained application of these editing operations is only possible due to the flat (unstructured) text model. Furthermore, the simple text model allows tools consuming or operating on texts to be implemented in a simple and straightforward way. For example, the compiler can readily operate on any text by scanning it in a standard linear fashion. To further ease the implementation of such tools, standard projections are defined from characters to elements, and vice versa: ($x{\downarrow}S$ denotes the projection of x into the set S)

$x \in \text{Elements} \Rightarrow x{\downarrow}\text{Character} := 1C_{16}$
$x \in \text{Characters} \Rightarrow x{\downarrow}\text{Element} := \text{NIL}$

In other words, a text can be interpreted as a plain character sequence, in which all elements are projected to a fixed ASCII code ($1C_{16}$, selected to be < $SPACE_{ascii}$). Thus, tools can treat elements as white space characters. A tool interested in elements only, can also treat a text as a sequence of elements, where characters are projected to the special element value NIL.

As mentioned above, the Oberon and Ethos text systems have converged by means of their extension concepts. However, the text editing model of Oberon texts is significantly more complex than that of Ethos. The following graph compares the Oberon and Ethos text editing models. It is surprising that the Oberon Texts module introduces that many abstractions (and therefore operations). On the other hand, the generalized stream interface of the Ethos text system almost enforces a more unified and simpler model.

Also, it is surprising that the Oberon Texts module contains a major design flaw in that it declares Reader and Writer types to be subtypes of the Files.Rider type. The reason for doing so is the mere fact that Reader and Writer are *implemented* in a file-based manner, and therefore (*internally!*) need to *use* riders. However, the subtype property cannot and indeed should not be used, as the Rider attributes reflect the conceptionally meaningless state of the *internal* use of file pieces. For example, the field *eof* inherited from Rider has no meaningful value for Reader and Writer instances. Likewise, the procedures of Files return meaningless values when a Reader or Writer is passed instead of a Rider. This points out a confusion of *is-a* and *uses-a* relations.

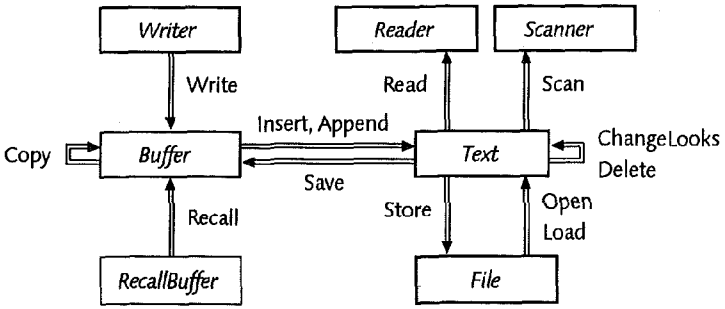


Figure 3.19 – The Oberon Text System.

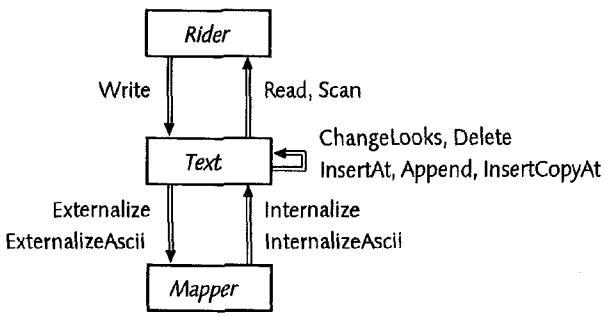


Figure 3.20 – The Ethos Text System.

The only text-specific operations are those supporting text editing. As in Oberon, subranges of the text are specified as intervals closed to the left and open to the right, e.g. [beg, end). Changing the attribute settings of a subrange (ChangeLooks) or deleting a subrange (Delete) involve a single text. Inserting a subrange of a text (InsertAt, Append), or inserting a copy of a subrange of a text (InsertCopyAt) involve two texts. Operation Append is a mere shortcut for the common case of inserting a text at the end of another one. Also, operation InsertAt may be seen as a combination of InsertCopyAt and Delete. Having special append and non-copying insertion methods enables text implementations to optimize common cases.

```
DEFINITION Texts;
   IMPORT Objects, Files, Fonts;

   CONST
      ElemChar = 1CX;
      (*NotifyMsg.op*)
         replace = 0; insert = 1; delete = 2;
```

```
   (*Rider.res*)
      Inval = 100; BadClass = 110;
   (*Rider.class*)
      Char = 0; String = 1; Short = 2; Int = 3; Long = 4; Real = 5; LongReal = 6; Set = 7;
      IntSet = {Short, Int}; LongSet = {Short, Int, Long};
      LongRealSet = {Real, LongReal};
   (*TextRider.class*)
      Element = 8;
   (*ChangeLooks(sel)*)
      font = 0; color = 1; offset = 2;

TYPE
   Attributes = POINTER TO AttributesDesc;
   AttributesDesc = RECORD (Objects.ObjectDesc)
      fnt: Fonts.Font; col, voff: LONGINT
   END;

   Elem = POINTER TO ElemDesc;
   ElemDesc = RECORD (Objects.CarrierDesc)
      base–: Text; w–, h–: LONGINT;
      PROCEDURE (E: Elem) Resize (w, h: LONGINT);
   END;

   Text = POINTER TO TextDesc;
   TextDesc = RECORD (Objects.PosStreamDesc)
      notify: BOOLEAN;
      PROCEDURE (T: Text) InternalizeAscii (VAR r: Objects.Mapper);
      PROCEDURE (T: Text) ExternalizeAscii (VAR r: Objects.Mapper);
      PROCEDURE (T: Text) InsertCopyAt (pos: LONGINT; t: Text; beg, end: LONGINT);
      PROCEDURE (T: Text) InsertAt (pos: LONGINT; t: Text; beg, end: LONGINT);
      PROCEDURE (T: Text) Append (t: Text);
      PROCEDURE (T: Text) Delete (beg, end: LONGINT);
      PROCEDURE (T: Text) ChangeAttr
         (beg, end: LONGINT; sel: SET; VAR attr: AttributesDesc);
   END;

   Link = POINTER TO LinkDesc;
   LinkDesc = RECORD (Objects.StreamLinkDesc)
      set, get: Attributes; elem: Elem;
      PROCEDURE (L: Link) SetAttr (sel: SET; VAR attr: AttributesDesc);
      PROCEDURE (L: Link) ReadElem (VAR r: TextRider; VAR elem: Elem);
      PROCEDURE (L: Link) ReadPrevElem (VAR r: TextRider; VAR elem: Elem);
      PROCEDURE (L: Link) WriteElem (VAR r: TextRider; elem: Elem);
   END ;

   Rider = RECORD (Objects.StreamRider)
      line: INTEGER; nextCh: CHAR;
      class: INTEGER;
      ch: CHAR; set: SET; bool: BOOLEAN;
      int, intbase: LONGINT; real: LONGREAL; elem: Elem;
```

```
        len: INTEGER; str: ARRAY 64 OF CHAR;
        PROCEDURE (VAR R: Rider) Scan;
        PROCEDURE (VAR R: Rider) WriteIntForm
          (n: LONGINT; base, w: INTEGER; fillCh, baseCh: CHAR);
        PROCEDURE (VAR R: Rider) WriteRealFixForm (x: LONGREAL; w, k: INTEGER);
        PROCEDURE (VAR R: Rider) WriteRealForm (x: LONGREAL; w: INTEGER);
        PROCEDURE (VAR R: Rider) WriteDate (date: LONGINT);
        PROCEDURE (VAR R: Rider) WriteTime (time: LONGINT);
      END;

      TextRider = RECORD (Rider)
        PROCEDURE (VAR R: TextRider) GetAttr (VAR a: AttributesDesc);
        PROCEDURE (VAR R: TextRider) SetAttr (sel: SET; VAR a: AttributesDesc);
        PROCEDURE (VAR R: TextRider) SetColor (col: LONGINT);
        PROCEDURE (VAR R: TextRider) SetFont (fnt: Fonts.Font);
        PROCEDURE (VAR R: TextRider) SetOffset (voff: LONGINT);
          (*SetColor, SetFont, SetOffset are shortcuts for SetAttr*)
        PROCEDURE (VAR R: TextRider) ReadElem (VAR e: Elem);
        PROCEDURE (VAR R: TextRider) ReadPrevElem (VAR e: Elem);
        PROCEDURE (VAR R: TextRider) WriteElem (e: Elem);
      END;

      Directory = POINTER TO DirectoryDesc;
      DirectoryDesc = RECORD (Objects.ObjectDesc)
        ref, spill: Files.Directory;
        fnt: Fonts.Directory;
        attr: Attributes;
        PROCEDURE (D: Directory) New (): Text;
        PROCEDURE (D: Directory) This (name: ARRAY OF CHAR): Text;
      END;

    VAR dir, stdDir-: Directory;
      log, logBuf, scarp: Text;

  END Texts.
```

Again, the Carrier/Link/Rider scheme has been applied. The interesting point is to note that texts and files are both positionable streams and that it is therefore possible to use both interchangingly wherever a positionable stream suffices. The only significant difference between texts and files are the option to *edit* texts – i.e. to insert and delete subsequences – and the attributes added to text-objects. Likewise, the rider defined in Texts can be used to create a standard ASCII file by simply setting it to a file. Or, the rider defined in Files can be used to write arbitrary bytes to a text which then can be used to edit the byte stream in an efficient manner.

By encapsulating the set of attributes attachable to a text-object in type Attributes, the text system can be extended to support new attributes. Doing so requires implementing extensions of texts, links, and text riders. Also, it is

possible to extend texts in other ways, i.e. to overcome the restrictions of standard texts by maintaining some kind of structure superimposed on a text.

When extending texts, the problem arises that there may be different text implementations existing at the same time in a system. This is of no problem unless inter-text operations are used. The definition of Texts contains three such operations: InsertCopyAt, InsertAt, and Append. It is important to note that in each case the *modified* text has control. In the cases of non-copying insertion (InsertAt, Append), the source text is modified by means of the Delete operation. Thus, it is again the modified text that has control.

To avoid covariance conflicts, it is necessary to support *any* text as parameter to these operations. Of course, the full support of attributes and an efficient implementation of copying and insertion is usually only possible, if both involved texts are of the *same* type. Hence, the method implementations check for type *identity* (e.g. using the type meta-information provided by module Modules). If both texts are of the same type, the internals of both texts are under control of the same module and operations are both efficient and complete. If the types of the texts differ, a different technique called *rider conversion* is used. Since texts have a common stream interface, it is always possible to copy from a text of a different implementation by reading it character by character, copying only those attributes defined for source and destination text.

Scanning a Text.

Often a text is used to pass parameter values. Hence, scanning of texts to extract values of basic types (like integers or strings) is a common task. In Oberon, a subtype of Texts.Reader, called Texts.Scanner, is used to scan texts. Since the Ethos rider interface requests a rider to *read* values of basic types directly, and since for texts it is not certain whether an appropriate sequence of characters will be encountered in a text, the standard text rider *scans* the text to return a value of the requested type. (If the scanned type does not match the requested type, an error status is returned.) Often the type of the next token to scan is not known in advance. Therefore the scan method of text riders is available to directly scan the next token. Then the scanner returns a *class* code, indicating which kind of token has been scanned in. As for all Ethos riders, all basic types of the language are supported. For example, it is possible to write a readable representation of a *set* to a text, and scan it back in.

Elements.

Having the text model defined, it is time to look at the precise definition of elements. At the model level, there is only one basic method defined for elements (Resize) allowing to redimension an element. Since Elem is a subtype of Objects.Carrier, elements also have methods for copying,

internalization, and externalization. That is, as far as module Texts is concerned, elements are text objects which have a bounding box, but no further semantics. In order to display an element its implementation has to be derived from some subtype of type Elem. Typically, view implementations define appropriate element subtypes, adding methods for proper interaction with a view. (Details follow in 3.4.8.)

Displaying Texts.
Obviously, a text also needs to be formatted, displayed, and printed. However, these operations are considered view specific and their description is thus postponed to Section 3.4.8, covering TextFrames. A question left open so far is how attributes spanning ranges of characters can be added to texts. In batch oriented typesetting systems (like TEX [Knu84]) interspersed *markups* are used to define attributes that remain valid till changed by a following markup. (A markup is a textually distinguished keyword interpreted by a formatter.) This idea is taken into the Ethos text model by using special elements as markups. For example, a *paragraph control* element (*parc*, for short) is used to define paragraph formatting attributes (left, centered, right, or block adjusted text, tabulator settings, line spacing, etc.). Since a markup is implemented as a single element and elements may be interactive, direct manipulation of attributes is possible. Also, the markup can be interpreted when displaying a text (cf. 3.4.8); changes to a markup can be made visible directly. Hence, a parc can behave similar to a LisaWrite [Wil83] or a MacWrite ruler.

Implementation of Standard Texts.
From the definition of texts it is clear that the two kinds of text objects, namely characters and elements, can be treated identically. Indeed this is done in the *Glyph* approach found in Extended InterViews [CL90]. However, the resulting performance and storage penalties seem prohibitive. Instead, *runs* are defined (Figure 3.21) and optimized. A run is a sequence of text objects having the same set of attributes. (In the current version, elements are always kept in a run of their own). This leads to the following syntax of the text implementation:

```
Text = {Run}.
Run = Attributes {TextObject}.
Attributes = font color offset.
TextObject = characterObject | extendedObject.
```

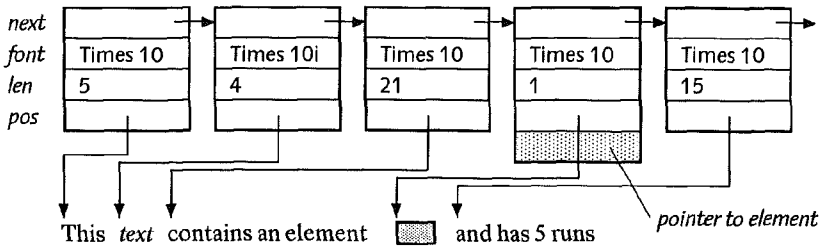


Figure 3.21 – Implementation of Texts Using Attributed File Pieces.

Note that the existence of runs is exclusively an optimization issue. Clients of texts need not be aware of it. Under the assumption that characters dominate elements, the implementation of character runs has been optimized. The resulting efficiency reaches that of text editors that have no concept of elements.

The standard text implementation has been designed such that texts of arbitrary size can be viewed and edited. Hence, a file-based approach using a so-called piece-list [Gut85] has been chosen. As the actual characters of a text are not maintained in memory, opening a text that contains no elements is very fast. However, if elements are contained in a text, they are all loaded when opening the text. An element might again be file-based, thus deferring costly read operations.

The directory object defined in Texts contains two references to file directories (*ref* and *spill*). The former is used whenever a text is retrieved using This, while the latter is used by file-based text implementations to create temporary files, e.g. when writing to a text, or when creating a new one using New. By replacing the *spill* reference to return memory-based "files", the text system can be tuned to retain newly written text in memory, while still being able to handle large file-based texts.

Whenever a text is opened that contains an element that has not yet been used in the current session, the defining modules are loaded. If this fails (e.g. the extension is unavailable), the facilities of module Stores are used to create an *alien element*. An alien element is displayed in a standard way (e.g. as a box) and is stored in a way to retain the information contained in the original element. Hence, it is possible to open documents even if certain extensions are currently not available. Furthermore, it is possible to *edit and store* such a document again without losing information contained in the alien elements.

3.4.7 The Display System

For modern interactive systems the flexible treatment of high-resolution raster devices became a standard requirement. Especially bitmapped displays and high-quality raster printers need be supported. Common to all these devices is the modelling of the output medium as a rectangular grid of picture elements – *pixels* – individually addressable using two-dimensional coordinates. Each of the pixels has a value out of a device-specific set. Typical value sets are {background, foreground} (e.g. {black, white}) for monochrome devices, {0..N–1} for grey-scale devices or for devices using a color-lookup table – *clut* –, and {0..N–1}×{0..N–1}×{0..N–1} for devices based on a three-dimensional color model (e.g. red-green-blue). Besides different coloring models, the devices differ significantly in their resolution – pixels per inch, often dots per inch (dpi) – and relative image size (pixels^2) they support. Table 3.1 lists some typical combinations of these parameters.

typical device	relative size [pixels^2]	resolution [dpi]	absolute size [mm^2]	color model [bit/pixel]
PC VGA 15" screen	640*480	64	253*190	1/4/8 clut
Macintosh 12" screen	640*480	72	225*169	1/4/8 clut; 24 RGB
Ceres 17" screen	1024*800	91	285*223	1
Chameleon 17" screen	1152*910	102	285*223	8 clut
Laser printer 1	2336*3425	300	197*289	1
Laser printer 2	3114*4566	400	197*289	1
Photo-Typesetter	37228*54614	2400	394*578	1

Table 3.1 – Characteristics of Typical Raster Devices.

The common subset of this stunning variety of devices is the *pixelmap* model. Therefore, the appropriate low-level abstraction that a display system should support is a pixelmap. Next, one has to decide what operations to perform on pixelmaps. In principle, a single operation SetPixel(x, y) would suffice, but just as block operations are provided for stream interfaces, it is important to select the proper higher pixelmap operations to achieve acceptable performance.

Since pixelmaps are device abstractions, it is necessary to define a bottleneck-interface, i.e. a set of primitive operations that has to be supported by every device presenting itself as a pixelmap. As for all bottleneck-interfaces, the set of operations is *not extensible* as it forms the connection between arbitrary clients using it and arbitrary device drivers implementing it. Chosing the right set of primitives is difficult, and other than for file-systems, the existing pixelmap interfaces differ significantly. On the one hand, it is not clear

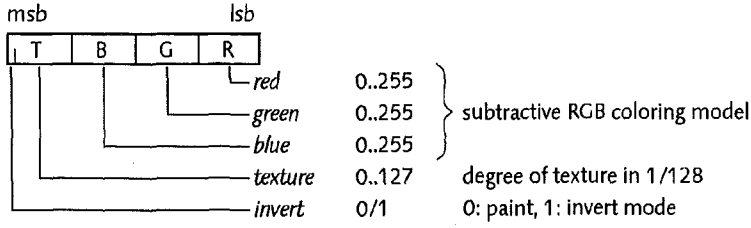
what coloring or imaging models to support. On the other hand the set of graphical primitives to be supported is highly application-dependent. For a discussion of many typical operations cf. [Sta89].

Another issue easily overlooked when designing a pixelmap interface is the available bandwidth between the issueing application and the performing graphics system. If both are located on a single machine, the bandwidth is usually quite high, e.g. limited by the bus bandwidth. Often the imaging device and the controlling application are not that close to each other. Examples are remote printing or display services. There, the bandwidth is typically limitted by some interconnecting network and the associated network protocols. A typical solution is the textual representation of the operations to perform and interpretation at the place of the device, e.g. PostScript [Ado85]. Another solution is the definition of a bottleneck that preserves sufficient information about the operations to be performed, such that the packaged parameters can be transmitted at modest costs. The latter approach is taken for example by the X Window System [SG86].

For many applications a single abstract pixelmap device covering all the needs of displaying and printing information avoids having separate code for printer and display output. For example, in the Oberon system the screen and printer abstractions are completely separate, hence inducing a doubling of code in many Oberon applications. A common result is that certain applications "cannot print", i.e. the additional printing code is just not implemented. This is not a new problem, and it has been attacked by systems like Display-PostScript [Web89]. However, it is felt that these systems are overly complex and tend to consume significant amounts of computing resources.

3.4.7.1 The Pixelmap Bottleneck Interface and the Coloring Model

The Ethos pixelmap abstraction is based on a rather general coloring model, an abstract resolution model, and a set of only four basic operations. The coloring model defines two color spaces, both effectively coded into a 32-bit integer. The first color space is device-independent ("abstract") and meant to be used by data models that carry color attributes. The second space is device-dependent and in its coding completely determined by individual devices. The device-independent color space is defined in Figure 3.22.



The meaning of texture is device dependent:
on a b/w device it means half-toning
on a color device it means intensity (i.e. grey value of color code FF'FFFF)

Special values with device-independent meaning:

`xx00'0000`	Black	*(texture value ignored)*
`1FFF'FFFF`	White, 25% intensity	
`3FFF'FFFF`	White, 50% intensity	
`5FFF'FFFF`	White, 75% intensity	
`7FFF'FFFF`	White, 100% intensity	
`FFFF'FFFF`	Fully invert	*(other values with invert=1 reserved)*

Figure 3.22 – Ethos Device-Independent Color Space.

Mapping from abstract colors to device colors is supported by a pair of mapping functions.

MapColor: Abstract-Color → Device-Color
NormColor: Device-Color → Abstract-Color

The abstract resolution model is based on a device-independent unit of 1/36'000 *mm*. Without rounding errors, the unit can be expressed as 1/12'700 *point* or 1/914'400 *inch*. (Rationals for chosing a device-independent unit are given in [Vet91].) All data model coordinates should be based on these units. Pixelmaps export a field *unit*, declaring how many abstract units relate to a single pixel of the device. Hence the mapping of global to device measures is a division by the device unit. Likewise, device coordinates can be mapped to global ones by multiplying with the device unit. (An application should cumulate coordinate values in the abstract space and avoid iterated mappings between both spaces to achieve maximum precision.) Table 3.2 lists some typical unit values.

dpi	units/dot	example
64	14'218	PC VGA 15" screen
72	12'700	Macintosh 12" screen
91	10'000	Ceres 17" screen
102	8'921	Chameleon 17" screen
300	3'048	Laser printer 1
400	2'286	Laser printer 2
2400	381	Photo-Typesetter

Table 3.2 – Typical Unit Values.

The unit of 1/36'000 mm is rather fine-grained: It supports a resolution of about 30 nm! Its values has been chosen in the course of the Opus project [Vet91] and was meant to support a variety of device resolutions and absolute measures (*mm*, *inches*, *points*) while avoiding rounding errors. In principle, one might argue that 36'000 should be replaced by a smaller value, e.g. some power of two. However, since the pixelmap units are *variable* values, no performance gain would result. The advantage of dealing with slightly smaller numbers is negligible: The photo-typesetter shows that 16-bit integers are still too restrictive.

Typical applications map colors and measures from abstract model to device space and then issue a *series* of primitive operations on the pixelmap. Hence, all primitives are defined to accept parameters in device colors and device coordinates. To decide which primitives should be supported, typical applications have been analyzed for their use of pixelmaps. The following set of primitives has been isolated for the bottleneck interface.

Dot (x, y, color)
Block (x, y, w, h, color, px, py)
Char (x, y, color, font, char)
String (x, y, color, font, array of char)

Block and String are mainly used to optimize common iterated applications of Dot and Char, respectively. For String this optimization is especially important when considering a transfer to a remote machine, as characters are expected to take a huge portion of the issued operations. The significance of not using dot to fill a rectangular area is evident. A subtle detail about Block is hidden by the two parameters *px* and *py* (read "pin-x" and "pin-y", respectively). To understand the importance of these parameters the texture coding of colors must be considered. For example, a monochrome pixelmap may support various "grey" patterns to simulate intensities. However, as soon as the color parameter controls more than one pixel it is important to have an alignment pin-point. Otherwise, it is not possible to avoid distortions after scrolling or

when drawing a complex area by appending multiple rectangular areas. Usually, the pin-point is related to some fixed point in model space and therefore invariant under scrolling in device space.

> The Oberon module Display provides an operation ReplPattern which allows to fill a rectangular area by replicating an arbitrary pattern. ReplPattern does not support a pin-point, instead it aligns patterns to the screen origin. Hence, whenever scrolling a view the whole view needs to be redrawn to avoid distortions. If a pin-point is present it is possible to use faster block-move plus partial re-drawing techniques to implement scrolling.

3.4.7.2 The Childmap Concept

Besides supporting a set of output primitives, pixelmaps provide for intra- and inter-pixelmap block copying. For inter-pixelmap block copies an additional problem is introduced, since source and destination pixelmaps in general have different color models, resolutions, and storage mappings. This problem has been "solved by avoidance": Instead of copying (part of) a view from one pixelmap to another, applications are expected to redraw (part of) the view when a copy between pixelmaps of different types would be necessary. However, completely preventing inter-pixelmap copying is too restrictive. For example, popup-menus or double-buffering techniques need a background pixelmap that is not related to a device, but used as a buffer to copy to and from.

To support background pixelmaps without introducing the complexity of arbitrary inter-pixelmap block copies, the concept of *childmaps* has been introduced. Every pixelmap has an operation that on demand allocates a childmap of given minimal width and height. Such a childmap has the same internal structure (color, resolution, and storage model) as the creating pixelmap. Thus, copying between a pixelmap and its childmaps, and also between childmaps of the same pixelmap, is easy to understand and easy to implement. As a useful side-effect, childmaps remove the resource allocation problem that occurs when providing a shared "background pixelmap", as is done for example in the Oberon system. Since the Oberon background pixelmap is shared, no application can rely on its context and therefore all uses must be restricted to short-lived temporary buffering.

3.4.7.3 The Font Subsystem

A particularly hard aspect of supporting pixelmaps of multiple resolutions is the representation of typographically acceptable characters. A common approach is to support outline-fonts, often called meta-fonts, that describe the geometric characteristics of characters without referring to a particular device resolution. Typical examples are the TEX Metafont [Knu86], PostScript

[Ado85], and TrueType [App90a] font description and rastering systems. Besides rastering a character specifically for a certain device resolution the outline-font needs to be tailored for graceful simplification of character shapes when rastering to devices with rather low resolutions (like current displays have). It is noteworthy that the problem of character mapping to devices occurs already at a lower level of abstraction: Even patterns cannot be mapped to devices of varying resolutions without introducing significant distortions. (The exception being primitive patterns, especially grey shades.) In general, it can be said that it is a conceptual mistake to allow the definition and application of arbitrary patterns when dealing with generalized pixelmap models. Therefore, the Ethos pixelmap model only supports colored (and potentially grey-shaded) block fills plus placement of characters specified by character code and font.

A pixelmap either passes the character/font pair to some remote service, or asks the font object to return a specific pattern for the pixelmap's resolution. By replacing the implementation of the font object, varying font machineries can be installed without disturbing clients of the pixelmap abstraction. (Since the pattern is indirectly retrieved by the pixelmap, instead of being retrieved by the client and passed through the interface, the problem of optimizing pattern transfer to a remote service is avoided.)

Often, an application needs only the bounding box information for a character, e.g., if the character rendering and placing happens completely on a remote machine. To optimize this situation, font objects have an additional operation returning only the box of a character.

The Oberon font system contains several misconceptions. First of all, fonts are named using the file names of pre-rastered *screen* fonts, e.g. "Syntax10.*Scn*.Fnt". In turn, models like texts contain names of device specific fonts. (Even Oberon's Printer module takes names of screen fonts to control what *printer* fonts to be used!) Secondly, the font system – realized in modules Display and Fonts – does not allow to separate abstract fonts from device specific resolutions. Thirdly, the Display module directly operates on patterns. This complicates implementations that support remote devices, as is done by Oberon ports for X Window [BCF*92][SG86].

3.4.7.4 Managing Multiple Screens

The predominant implementation of the pixelmap abstraction is the raster screen used to interact with the machine. Often, it is useful to have a support for multiple screens, e.g. a monochrome and a color screen. To simplify applications using screen coordinates, it is useful to place the screens within a single coordinate system, thus avoiding the application-level distinction to what screen a certain coordinate belongs. This was done in QuickDraw [App85] for the Macintosh, and is also done in the Oberon system.

Screen number i has an origin $(x_i, 0)$ with $x_i \geq 0$. The first screen is always

installed at origin (0, 0). Different screens may have different resolutions and color models. Thus, the unified coordinate space is meant to help in organizing a user-interface spanning multiple screens, but it does not introduce some kind of metrics spanning multiple screens. For example, in the picture, it is possible that $x_2-x_1 < x_1-x_0$, while at the same time the display area of screen 1 is physically wider than that of screen 0. This may happen, if screen 1 has less pixels per row but also if it has a smaller resolution than screen 0.

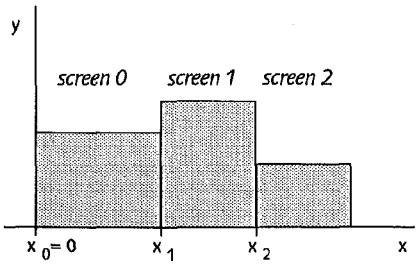


Figure 3.23 – Placement of Multiple Screens in a Single Coordinate System.

3.4.7.5 Module Frames – Carrier/Link/Rider Scheme for Pixelmaps

Collecting the concepts discussed above into a single unified module Frames leads to the following interface. The entities giving module Frames its name are frames: Rectangular areas attached to a pixelmap. Frames interact with pixelmaps by means of a bottleneck interface. The bottleneck interface supports low-level clipping to rectangular bounds by means of so-called ports. A port is again a rectangular area of a pixelmap, and typically a subset of the area covered by the frame using the port.

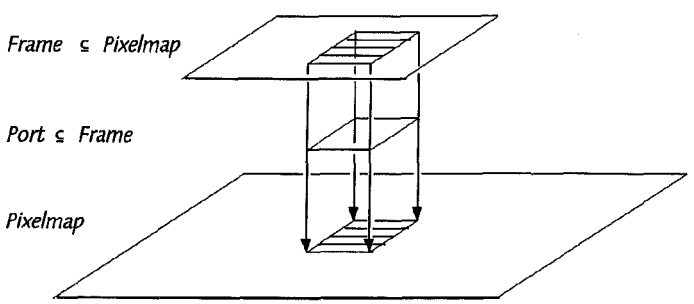


Figure 3.24 – Frames, Ports, and Pixelmaps.

Again, the Carrier/Link/Rider scheme has been applied to achieve an orderly decomposition. Pixelmaps are carriers, ports are links, and frames are riders. The idea is that a frame may add higher-level operations and abstractions. For example, a frame may introduce a model space coordinate system with operations to zoom, pan, or rotate the model coordinates before mapping them to device coordinates. Of course, many different frames may reside on the same pixelmap. The frame-to-pixelmap bottleneck is represented by ports which also are clipping windows onto pixelmaps. The SetFrame operation of a pixelmap prevents the setting of a port outside of the pixelmap's bounds. Hence, low-level clipping is limited to the bounds actually presented by the port without risking invalid access. It should be noted that it would not suffice to use the frame bounds for clipping: Often it is required to restore part of a frame, e.g. after scrolling the displayed contents. At the same time the frame coordinates should be preserved (cf. 3.4.8). The solution is to setup a port that clips to a subset of the frame area.

The interface of module Frames is listed below. It is noteworthy that there is a pattern descriptor, but that there are no operations on ports that directly make use of patterns. Instead, the only means to define arbitrary patterns and apply them to a pixelmap is by means of defining an appropriate font that encapsulates the device-specific pattern to raster mapping. For example, Ethos uses a font called EthosPatterns that contains system patterns, e.g. cursor, pointer, and caret patterns.

As for most Ethos modules, Frames also contains a default implementation. It implements at least one screen driver for a screen located at coordinate (0, 0). By means of procedure ThisScreen the default screen and any other installed screen can be located within the screen coordinate system. Screen allocation is done in ascending order along the x-axis (Figure 3.23). Hence, ThisScreen may also be used to enumerate all screens available at any one time.

```
DEFINITION Frames;
   IMPORT Objects;

   CONST mm = 36000;

   TYPE
      Pattern = POINTER TO PatternDesc;
      PatternDesc = RECORD
         w, h, mw: INTEGER;   (*metrics in carrier space*)
         map-: POINTER TO ARRAY OF SET
      END;
```

```
Char = POINTER TO CharDesc;
CharDesc = RECORD (PatternDesc)
   dx, x, y: INTEGER   (*metrics in carrier space*)
END;

Font = POINTER TO FontDesc;
FontDesc = RECORD (Objects.CarrierDesc)
   PROCEDURE (F: Font) ThisChar (ch: CHAR; unit: LONGINT): Char;
   PROCEDURE (F: Font) ThisBox (ch: CHAR; unit: LONGINT): Char;
      (*request box information only ⇒ may return ch.map = NIL*)
END;

NormalColor = LONGINT;   (*device independent*)
Color = LONGINT;

Port = POINTER TO PortDesc;
PortDesc = RECORD (Objects.ObjectDesc)
   x–, y–, w–, h–: LONGINT;
   PROCEDURE (P: Port) Resize (x, y, w, h: LONGINT);
   PROCEDURE (P: Port) Fit (F: Frame);
   PROCEDURE (P: Port) GetDot (x, y: LONGINT): Color;
   PROCEDURE (P: Port) Dot (x, y: LONGINT; col: Color);
   PROCEDURE (P: Port) Block (x, y, w, h: LONGINT; col: Color; px, py: LONGINT);
   PROCEDURE (P: Port) Char (x, y: LONGINT; col: Color; fnt: Font; ch: CHAR);
   PROCEDURE (P: Port) String
      (x, y: LONGINT; col: Color; fnt: Font; VAR str: ARRAY OF CHAR);
   PROCEDURE (P: Port) CopyBlock (sx, sy, w, h, dx, dy: LONGINT);
   PROCEDURE (P: Port) CopyBlockFrom (src: Port; sx, sy, w, h, dx, dy: LONGINT);
END;

PixelMap = POINTER TO PixelMapDesc;
PixelMapDesc = RECORD (Objects.CarrierDesc)
   x–, y–, w–, h–: LONGINT   (*allocated carrier space*);
   unit–, colors–: LONGINT;
   bkgnd–, tone–, forgnd–, invert–: Color;
   PROCEDURE (P: PixelMap) Init (x, y, w, h, unit, colors: LONGINT);
   PROCEDURE (P: PixelMap) NormColor (col: Color): NormalColor;
   PROCEDURE (P: PixelMap) MapColor (col: NormalColor): Color;
   PROCEDURE (P: PixelMap) SetFrame (VAR f: Frame; x, y, w, h: LONGINT);
   PROCEDURE (P: PixelMap) NewChild (w, h: LONGINT): PixelMap;
END;

Screen = POINTER TO ScreenDesc;
ScreenDesc = RECORD (PixelMapDesc)
   next–: Screen
END;
```

```
    Frame = RECORD (Objects.ObjectDesc)
       x–, y–, w–, h–: LONGINT;   (*rectangular region in carrier space*)
       base–: PixelMap; link–: Port;
       PROCEDURE (VAR F: Frame) ConnectTo
          (base: PixelMap; link: Port; x, y, w, h: LONGINT);
       PROCEDURE (VAR F: Frame) Dot (x, y: LONGINT; col: Color);
       PROCEDURE (VAR F: Frame) Block
          (x, y, w, h: LONGINT; col: Color; px, py: LONGINT);
       PROCEDURE (VAR F: Frame) Char
          (x, y: LONGINT; col: Color; fnt: Font; ch: CHAR);
       PROCEDURE (VAR F: Frame) String
          (x, y: LONGINT; col: Color; fnt: Font; str: ARRAY OF CHAR);
       PROCEDURE (VAR F: Frame) CopyBlock (sx, sy, w, h, dx, dy: LONGINT);
       PROCEDURE (VAR F: Frame) Restore (x, y, w, h: LONGINT);
       PROCEDURE (VAR F: Frame) Resize (x, y, w, h: LONGINT);
       PROCEDURE (VAR F: Frame) Broadcast (VAR msg: Objects.NotifyMsg);
       PROCEDURE (VAR F: Frame) Handle (VAR msg: Objects.NotifyMsg);
    END;

  VAR
    bkgnd–, tone–, forgnd–, invert–: NormalColor;

  (* Geometric Utilities *)
  PROCEDURE Contains (F: Frame; x, y: LONGINT): BOOLEAN;
  PROCEDURE Clip (VAR F: Frame; x, y, w, h: LONGINT);   (* F := F ∩ (x, y, w, h) *)
  PROCEDURE ClipPort (P: Port; x, y, w, h: LONGINT);   (* P↑ := P↑ ∩ (x, y, w, h) *)
  PROCEDURE SetClippedPort (VAR F: Frame; x, y, w, h: LONGINT);
                                                (* F.link := F ∩ (x, y, w, h) *)

  (* General *)
  PROCEDURE InstallScreen (S: Screen);
  PROCEDURE ThisScreen (x: LONGINT): Screen;

END Frames.
```

A few standard colors in global color space are exported by Frames. Applications using only these can expect to achieve good results on most pixelmaps. Otherwise, pixelmaps contain a field *colors*, giving the number of different colors available. This can be used by applications to switch from color to texture filled areas when outputting to monochrome instead of color pixelmaps.

Frames have methods Handle and Broadcast to deal with message records. The intention is that operations typically handled by broadcasting to all subframes are coded using message records instead of methods. Thus, the default implementation of Handle calls Broadcast for the given message, while the default implementation of Broadcast ignores the message, i.e. does nothing.

Frames.Frame is declared as a record, not as a pointer. This allows for using light-weight frames directly without allocating a separate object. Examples might be frames used by an application to implement buttons or the like placed directly within some other frame. However, more heavy-weight frames are usually requested from directory objects or installed within generic container frames, making a pointer to a frame necessary. Such a pointer type can be introduced when needed, and indeed is introduced in module Viewers (cf. 3.4.8.1).

3.4.8 User Interface Concepts

The user interface of a system is the way it presents itself to the "world". A careful design of the elements and style of interaction is important to arrive at a masterable system. A set of clearly worked-out conventions is generally a good idea (e.g. [App85]). A more direct way has been taken for Ethos: The major user interface abstractions are captured in appropriate frameworks. Typical applications will just provide some concrete implementations derived from the abstractions of the frameworks. Hence, as long as no defaults are overridden, the system behaves consistently since shared behavior is implemented by shared code. This is similar to the use of application frameworks in traditional systems (e.g. MacApp [App90b]). (Of course, in Ethos almost all pre-defined abstractions can be partially or completely replaced, creating a potentially very different system.)

3.4.8.1 Low-Level Organization of Screens

User-Interfaces built within Ethos are expected to rely on two primary resources: raster displays and direct manipulation devices, like mice. To simplify the task of higher-level user-interface components, it is useful to have some low-level organization of screen-areas into sub-areas. Basically, there are two well-known techniques of organizing a screen, one based on screen *tiling* [Tei84], the other on *overlapping windows*. (For a general discussion cf. [Wil89]). The former is based on subdividing a screen into non-overlapping areas, while the latter is based on the so-called "desktop metaphor" simulating the chaos of overlapping sheets of paper on a desktop. It is felt that the overlapping windows approach is well suited for small screens, where essentially only one sufficiently large work-area can be presented. However, on modern workstation hardware, this limitation is no longer present. Just to the opposite, if the *usable* screen area is rather large, overlapping window systems tend to *use* only a portion of it, while (by definition) a tiling system uses all the usable space, cf. Figure 3.25.

It has been observed that users arrange their overlapping windows on large screen by hand such that they effectively do *not* overlap! To cope with this paradox situations, many overlapping window applications today provide tiling-like options, i.e. are able to automatically place their windows in a tiling-like fashion.

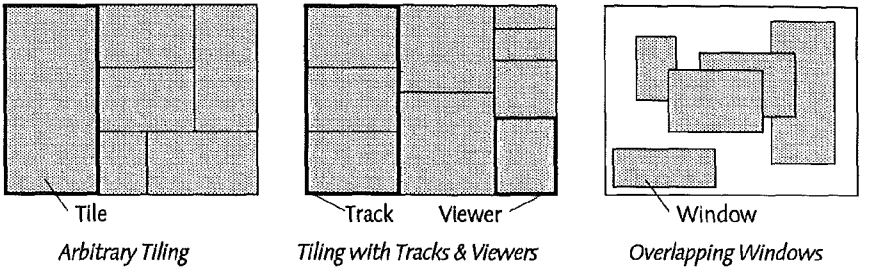


Figure 3.25 – Tiling vs. Overlapping Windows.

In Ethos the module Viewers introduces three organizational levels of abstraction: layouts, tracks, and viewers. A *layout* adds a logical organization to a screen. The decoupling of screens and layouts helps separating concerns: A screen is concerned about displaying pixels, while a layout is concerned about organizing a screen into logical sub-areas. Standard layouts are vertically tiled into a series of *tracks*. Tracks are horizontally tiled into a series of *viewers*. (Tracks and viewers with almost identical semantics already exist in Oberon. The layout abstraction is approximated by the hard-wired features of the Oberon Viewers module.)

To compensate for the no-overlaps commitment of tiling systems, Oberon introduced the possibility to *cover* one or more tracks completely by overlaying a new track. This has proven to be a powerful and versatile feature of the Oberon viewer system and therefore has been retained in Ethos. Furthermore, Oberon allows to move the separation line between two viewers, effectively resizing the two involved viewers. This has been generalized in Ethos to also allow moving the separation line between two tracks. (This generalization introduces slight technical complications as tracks, like viewers, have a guaranteed minimal size. A track to be resized may cover several tracks below. Then, the minimal sizes of the covered tracks need be added and used to constrain the resizing of the covering track.)

Figure 3.26 shows how viewers, tracks, and layouts are interrelated. Viewers and tracks are kept in rings containing a filler-trailer. A track ring is maintained under preservation of the invariant that the tracks in the ring do not overlap, have the same y-coordinate and height as their underlying layout, and that the sum of the track widths equals the width of the underlying layout. Likewise, the invariant of a viewer ring is that the viewers do not overlap, have the same x-coordinate and width as their underlying track, and that the sum of the

viewer heights equals that of the underlying track. Except for filler tracks, all tracks have a guaranteed minimal width. Also, all viewers that are not filler viewers have a guaranteed minimal height.

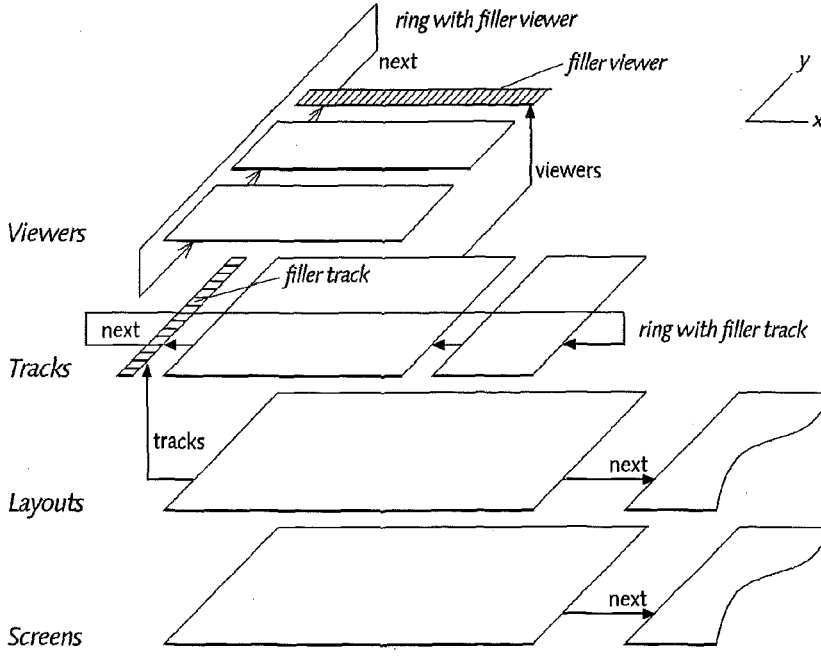


Figure 3.26 – Screen Organization using Layouts, Tracks, and Viewers.

Since tracks may cover one or more tracks below, each track has a (potentially empty) list of covered tracks. Of course, each covered track may again cover tracks even further below, leading to a push-down stack structure, cf. Figure 3.27. The invariant here is that a track is always as wide as the sum of the widths of the covered tracks. Therefore, it needs to be at least as wide as the sum of the minimal widths of the covered tracks.

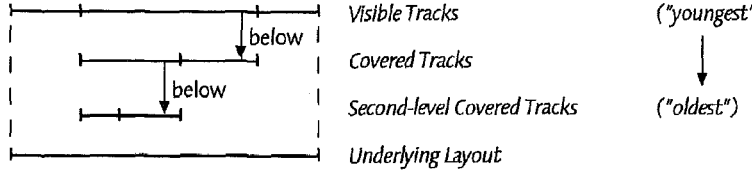


Figure 3.27 – Covered Tracks, Handled as a Push-Down Stack.

Another abstraction introduced by module viewers are *markers*. A marker maintains visible interaction states, e.g. the mouse cursor and the pointing marker, and the caret and selection of the text editor. Markers with an iconic appearance are supported by subtype CharMarker: A character out of a font is used to represent the marker on the screen. (Examples: cursor, pointer, caret; all using the font EthosPatterns.) Markers with varying looks are covered by subtype AreaMarker. There, the bounding box of the actual markings is used to define the marker. (Example: selections.)

Markers are derived from frames and have a pin-point (sometimes called "hot-spot") (*pinX*, *pinY*) and a field *on* indicating their visibility. Using the operation Draw(*x*, *y*), the pin-point of a marker may be moved to the point (*x*, *y*), also making the marker visible. A marker may be made invisible by invoking operation Fade. A special operation is provided to remove all markers within a certain rectangular area. Usually, this is done before applying updates to that area.

Two standard markers are defined in Viewers: The mouse *cursor* and the *pointer*. The former serves to track the current mouse position, while the latter is used to mark a specific position on the screen. The state of the pointer is evaluated by auxiliary procedures MarkedFrame and MarkedViewer to retrieve the currently marked (innermost) frame and the currently marked viewer, respectively.

```
DEFINITION Viewers;
   IMPORT Objects, Frames, Fonts;

   CONST resume = 1; suspend = 0;

   TYPE
      Marker = POINTER TO MarkerDesc;
      MarkerDesc = RECORD (Frames.Frame)
         on–: BOOLEAN; pinX–, pinY–: LONGINT;   (*pinX, pinY in screen coordinates*)
         PROCEDURE (M: Marker) Draw (x, y: LONGINT);
         PROCEDURE (M: Marker) Fade;
      END;

      CharMarker = POINTER TO CharMarkerDesc;
      CharMarkerDesc = RECORD (MarkerDesc)
         fnt–: Fonts.Font; ch–: CHAR;
         PROCEDURE (M: CharMarker) SetLooks (fnt: Fonts.Font; ch: CHAR);
      END;

      AreaMarker = POINTER TO AreaMarkerDesc;
      AreaMarkerDesc = RECORD (MarkerDesc)
         areaW–, areaH–: LONGINT;
         PROCEDURE (M: AreaMarker) SetArea (w, h: LONGINT);
      END;
```

```
    Frame = POINTER TO Frames.Frame;

    Viewer = POINTER TO ViewerDesc;
    ViewerDesc = RECORD (Frames.Frame)
       next–: Viewer; state–: INTEGER;
       PROCEDURE (V: Viewer) Open (x, y: LONGINT);
       PROCEDURE (V: Viewer) MinH (): LONGINT;
       PROCEDURE (V: Viewer) Close;
    END;

    Track = POINTER TO TrackDesc;
    TrackDesc = RECORD (Frames.Frame)
       next–, below–: Track; viewers–: Viewer;   (*below is NIL-terminated list*)
       PROCEDURE (T: Track) Open (x: LONGINT; filler: Viewer);
       PROCEDURE (T: Track) Cover (x, w: LONGINT; filler: Viewer);
       PROCEDURE (T: Track) MinW (): LONGINT;
       PROCEDURE (T: Track) AllocViewer (): LONGINT;
          (*returns "good" y-coordinate for next viewer to be opened in T*)
       PROCEDURE (T: Track) ThisViewer (y: LONGINT): Viewer;
       PROCEDURE (T: Track) Close;
    END;

    Layout = POINTER TO LayoutDesc;
    LayoutDesc = RECORD (Frames.Frame)
       next–: Layout; tracks–: Track;
       PROCEDURE (L: Layout) Open (filler: Viewer);
       PROCEDURE (L: Layout) AllocTrack (): LONGINT;
          (*returns "good" x-coordinate for next track to be opened in L*)
       PROCEDURE (L: Layout) ThisTrack (x: LONGINT): Track;
    END;


    LocateMsg = RECORD (Objects.NotifyMsg)
       x, y: LONGINT; f: Frame
    END;
    StateMsg = RECORD (Objects.NotifyMsg)
       id: INTEGER; x, y, w, h: LONGINT
    END;

VAR
    all–: RECORD x–, y–, w–, h–: LONGINT END;   (*bounding box of all screens*)
    cursor, pointer: Marker;

PROCEDURE RemoveMarks (x, y, w, h: LONGINT);
PROCEDURE RemoveMarksIn (port: Frames.Port);
PROCEDURE MarkerDefaults;

PROCEDURE Broadcast (VAR msg: Objects.NotifyMsg);
PROCEDURE Restore (x, y, w, h: LONGINT);
PROCEDURE ThisLayout (x: LONGINT): Layout;
```

```
    PROCEDURE ThisTrack (x: LONGINT): Track;
    PROCEDURE ThisViewer (x, y: LONGINT): Viewer;
    PROCEDURE ThisFrame (x, y: LONGINT): Frame;
    PROCEDURE NotifierDefaults;

    PROCEDURE MarkedFrame (visiblyOnly: BOOLEAN): Frame;
    PROCEDURE MarkedViewer (visiblyOnly: BOOLEAN): Viewer;

    PROCEDURE RecallViewer (): Viewer;   (*viewer that got closed or replaced last*)
END Viewers.
```

It is noteworthy that module Viewers introduces completely independent abstractions for layouts, tracks, and viewers. This differs significantly from the Oberon model where a single abstraction *frame* is used for all organizational purposes. In turn *every* frame in the Oberon system contains fields *dsc* and *next*, just in case that the frame has subframes (*dsc*), or is part of a list of frames (*next*). This was felt too heavy-weight to be part of the frame abstraction. In Ethos the sub-structure of a frame (if present) is completely private to that frame. Instead of allowing a generic traversal of the hierarchical frame structure, each frame has standard methods for broadcasting and handling messages. In turn, Ethos frames are conceptually more light-weight than Oberon frames.

A positive result of not over-generalizing the nesting structure of frames is the possibility to declare pointers with strongest possible types. For example, a track contains a rings of viewers. Since a track is not a special case of some general frame having sub-frames, the track contains a pointer *viewers* of type Viewer. Likewise, viewers are threaded into the ring using pointers *next* of type Viewer. This makes the data-structure much more comprehensible, while at the same time saving many type guards in the actual code.

> The argument that one might save code by introducing one general frame nesting concept, as was done in the Object Oberon library [MTG89], did not hold in practice. The implementations of, say, layouts and tracks, are actually quite different and a generalization would simply merge both, leading to a less comprehensible design.

Another interesting issue is the use of Oberon-style message records instead of Oberon-2-style methods in module Viewers. The idea is that all messages that are forwarded (often broadcasted) to sub-frames without explicitly distinguishing among different messages are implemented using message records. In other words, whenever a message needs to be interpreted by the receiving frame, instead of being forwarded, methods are used. The mixture of the two approaches may seem a little awkward. However, the resulting achievement of having interfaces as expressive as possible was felt more important. (Using only methods would lead to an awkward design indeed: Instead of having a single broadcast mechanism for abstract messages, each

abstraction would have to add a special broadcast for each broadcast-message newly defined.)

3.4.8.2 Command Invocation Conventions

The text system plays a crucial role in both, the Oberon and Ethos systems. In Oberon it is predetermined to be the standard and primary data model used for command invocation and command parameter passing. This is reflected by having it located just below module Oberon, while all other data models are implemented in modules not known to Oberon. In Ethos, the text system is not as predetermined to play such a central role, but the standard configuration of Ethos relies just as heavily on it as Oberon does. The following (partial) module diagrams compare the Oberon and Ethos structures, as far as the placement of data models is concerned. All modules below the separation line belong to the core of Oberon or Ethos, respectively. (In principle, Texts could import Graphics, or vice versa. Hence, on the level of data models, a certain asymmetry may exist even in the Ethos system.)

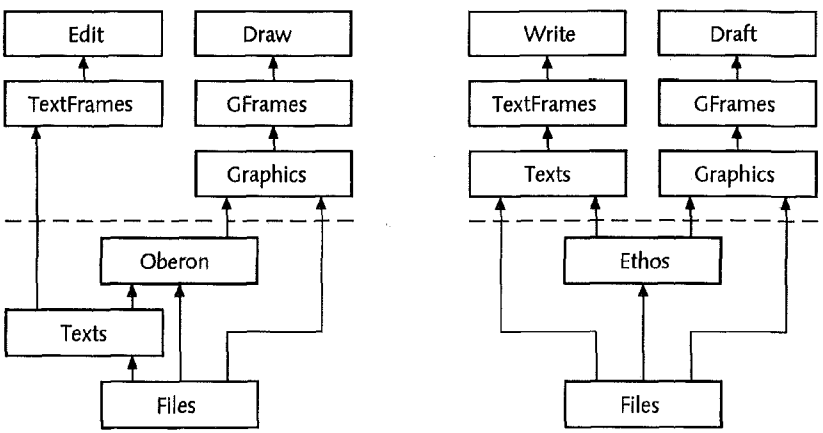


Figure 3.28 – Placement of Module Texts. Left: Oberon System – Right: Ethos System.

In Oberon, a command is an arbitrary (parameterless and exported) procedure. It can be invoked *by name*, i.e. by using the procedure name and the name of the implementing module. Usually, commands are called by means of a command interpreter. In Oberon, every part of a text visible on the screen can be interpreted as a command by applying a certain mouse click combination to it. If the text clicked at follows the form *M.P*, where *M* is a module and *P* a procedure name, the system tries to call command *M.P*. Most commands expect some parameters. For example, a command to open a text

for editing expects the name of the particular text to be opened. Since commands are parameterless procedures, parameters are passed by convention: The command interpreter sets some global variables such that they indicate the context of the command. These global variables are then, again by convention, interpreted by called commands.

This is the spot, where Oberon introduces some predetermination of module Texts: Module Oberon contains a global variable of type ParList that already contains fields *text* and *pos*, indicating the textual context of the command and the position just after the command's name. Every Oberon command can expect that these fields are set to something meaningful by the calling command interpreter. The following code fragment shows the typical interaction of caller (command interpreter) and callee (command).

```
PROCEDURE Caller (name: ARRAY 32 OF CHAR; VAR res: INTEGER);
   VAR par: Oberon.ParList;
      frame: Display.Frame; text: Texts.Text; pos: LONGINT;   (*actual parameters*)
BEGIN ...
   NEW(par); par.frame := frame; par.text := text; par.pos := pos;
   Oberon.Call(name, par, FALSE, res);
   ...
END Caller;

PROCEDURE Callee*;
   VAR S: Texts.Scanner;
BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
   Texts.Scan(S);   (*scan actual parameters*)
   ...
END Callee;
```

Since Ethos retained the Oberon concept of *commands*, the question arises how Ethos performs parameter passing from command interpreters to commands. Before going into the details, a small discourse on the extensibility of calling conventions for commands may be in order. In Oberon, the standard parameter record may be extended to add additional parameters. However, the extension must be known to caller and callee.

In Ethos, a different approach has been taken. Every module providing a new data model (texts, graphics, pictures, etc.) also provides a global parameter variable. By time-stamping each of these plus having a standard time-stamp variable in module Ethos, commands can find out, whether certain parameters are present, or not. This way, Ethos decouples callers and callees completely: Every data model known to the caller can be used to deposit corresponding parameters, and every data model known to the callee can be used to check for valid parameters. Again, the following code fragment shows the typical interaction of caller and callee – this time using the Ethos model. For the sake of easy comparison, texts are again used as parameters.

```
PROCEDURE Caller (name: ARRAY 32 OF CHAR; VAR res: INTEGER);
   VAR frame: Viewers.Frame; text: Texts.Text; pos: LONGINT;   (*actual parameters*)
BEGIN ...
   Texts.par.time := Ethos.par.time + 1;
   Texts.par.text := text; Texts.par.beg := pos; Texts.par.end := text.Length();
   Ethos.Call(name, frame, FALSE, res);   (*increments Ethos.par.time*)
   ...
END Caller;

PROCEDURE Callee*;
   VAR S: Texts.Rider;
BEGIN
   IF Texts.par.time = Ethos.par.time THEN
      Texts.par.text.SetRiderAt(S, Texts.par.beg);
      S.Scan;   (*scan actual parameters*)
      ...
   END
END Callee;
```

The additional check for validity of a particular parameter makes the Ethos command slightly more complicated. On the other hand, commands that need a text as parameter, but that are called by command interpreters that cannot provide a meaningful textual context are in trouble when using the Oberon conventions: All commands simply rely on having a textual parameter available. An example is a graphics system that supports invocation of commands bound to some kind of button. Then it is unnatural to add an artificially created, empty text-parameter just to avoid run-time errors when a command is called that expects a text-parameter. Instead, appropriate graphical parameters should be passed. This way a command can check whether it is called with appropriate parameters and, if not, use a default behavior or signal an error message.

```
DEFINITION Ethos;
   IMPORT Objects, Viewers;

   TYPE
      Param = RECORD
         time: LONGINT; frame: Viewers.Frame; viewer: Viewers.Viewer
      END;
      InputMsg = RECORD (Objects.NotifyMsg)
         ch: CHAR
      END;
      CursorMsg = RECORD (Objects.NotifyMsg)
         x, y: LONGINT;
         keys: SET
      END;
```

```
    PointerMsg = RECORD (Objects.NotifyMsg)
      x, y: LONGINT
    END;
    DefocusMsg = RECORD (Objects.NotifyMsg) END;

  VAR
    par: Param;
    focus–: Viewers.Frame;   (*always valid*)

  PROCEDURE Collect (cost: INTEGER);
    (*decrements internal counter by cost, when counter < 0 cause garbage collection*)
  PROCEDURE Separate (cmd: ARRAY OF CHAR; VAR mod, ident: ARRAY OF CHAR);

  PROCEDURE Install (mod: ARRAY OF CHAR);
    (*if module mod is not present, it is tried to load it*)
  PROCEDURE Call
    (cmd: ARRAY OF CHAR; frame: Viewers.Frame; new: BOOLEAN; VAR res: INTEGER);
    (*automatically sets par.viewer if frame # NIL*)

  PROCEDURE PassFocus (f: Viewers.Frame);   (*passing NIL is valid*)

  PROCEDURE SystemKey (ch: CHAR);
    (*standard handling of system keys, e.g. ESC to defocus and remove all marks*)
  PROCEDURE Loop;
END Ethos.
```

3.4.8.3 Standard Look and Feel

Based on modules Viewers and Ethos, module Looks defines standard viewers and standard viewer components that can be used to compose viewers with a common "look and feel". The standard viewers – called *twin-viewers* – contain two sub-frames, a head and a body frame, cf. Figure 3.29. The separation line between head and body frame is interactively adjustable while the viewer is open. A twin-viewer also implements the interactive adjustment of track and viewer separation lines that coincide with one of the viewer's borders.

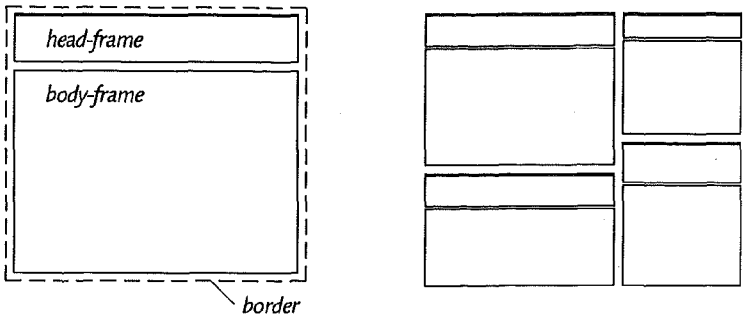


Figure 3.29 – Twin-Viewer – Screen Organization using Tracks and Twin-Viewers.

Head and body frames are displayed the same way to help placing arbitrary concrete frames into both. In Oberon, the head frames are displayed using inverse video, requiring all frames installed as head frames to display their contents using inverse mode operations. This was felt to restrictive.

The second set of abstractions provided by Looks are components used to create scrollable views with interactive scrollers. Looks defines Scrollable to be a frame containing a contents frame and two scroller frames. (A scroller may have zero width or height, effectively making both scrollers optional.) The contents frame has a simple interface hiding the details of concrete scroller components. The idea is that a frame displaying some view should only be concerned with the view itself. Outer borders and scroll-bars are added by the Scrollable abstraction.

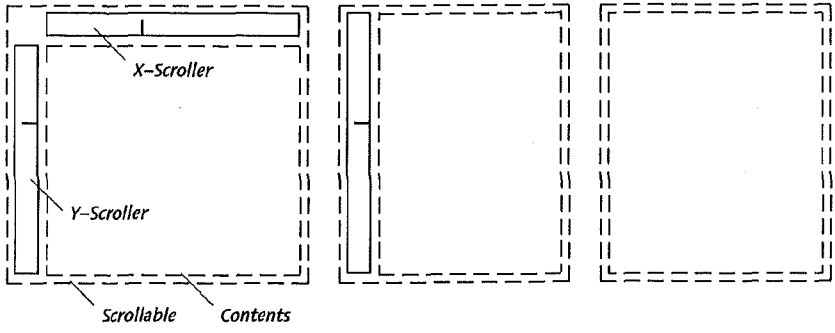


Figure 3.30 – Scrollable with two Scrollbars – with one Scrollbar – Framed (without Scrollbars)

In the Oberon system, no generic support of the Scrollable kind exists. Borders and scroll-bars are implemented over and over again for every new viewer class.

Scrollable frames without scroll-bars are normally used for the head-frames of twin-viewers, where typically only a border but no scroll-bars are wished.

```
DEFINITION Looks;
   IMPORT Objects, Frames, Viewers;

   TYPE
      Contents = POINTER TO ContentsDesc;
      ContentsDesc = RECORD (Frames.Frame)
         PROCEDURE (VAR C: ContentsDesc) GetRanges (VAR w, h: LONGINT);
            (*extent of the model space*)
         PROCEDURE (VAR C: ContentsDesc) XPos (): LONGINT;
         PROCEDURE (VAR C: ContentsDesc) YPos (): LONGINT;
            (*logical positions in model space corresponding to current scrolling position*)
         PROCEDURE (VAR C: ContentsDesc) TrackXPos
            (VAR x, y: LONGINT; VAR keysum: SET; VAR pos0, pos1: LONGINT);
            (*interactive scrolling feedback requested by x-scroller*)
         PROCEDURE (VAR C: ContentsDesc) TrackYPos
            (VAR x, y: LONGINT; VAR keysum: SET; VAR pos0, pos1: LONGINT);
            (*interactive scrolling feedback requested by y-scroller*)
         PROCEDURE (VAR C: ContentsDesc) SetXPos (pos: LONGINT);
         PROCEDURE (VAR C: ContentsDesc) SetYPos (pos: LONGINT);
            (*scroll view to the given logical positions in model space*)
      END;

      XScroller = POINTER TO XScrollerDesc;
      XScrollerDesc = RECORD (Frames.Frame)
         contents–: Contents; tickH–: LONGINT;
         PROCEDURE (VAR S: XScrollerDesc) Init (c: Contents; tickH: LONGINT);
      END;
      YScroller = POINTER TO YScrollerDesc;
      YScrollerDesc = RECORD (Frames.Frame)
         contents–: Contents; tickW–: LONGINT;
         PROCEDURE (VAR S: YScrollerDesc) Init (c: Contents; tickW: LONGINT);
      END;

      Scrollable = POINTER TO ScrollableDesc;
      ScrollableDesc = RECORD (Frames.Frame)
         xscroller–: XScroller; yscroller–: YScroller; contents–: Contents;
         barW–, barH–, left–, right–, bottom–, top–: LONGINT;
         PROCEDURE (VAR S: ScrollableDesc) Init
            (sx: XScroller; sy: YScroller; c: Contents;
               barW, barH, left, right, bottom, top: LONGINT);
      END;
```

```
    Viewer = POINTER TO ViewerDesc;
    ViewerDesc = RECORD (Viewers.ViewerDesc)
      headH–, headDefH–: LONGINT;
      head–, body–: Viewers.Frame
      PROCEDURE (VAR V: ViewerDesc) Init
        (head, body: Viewers.Frame; headH: LONGINT);
      PROCEDURE (VAR V: ViewerDesc) ShiftGap (headH: LONGINT);
      PROCEDURE (VAR V: ViewerDesc) TitleOf (): Viewers.Frame;
      PROCEDURE (VAR V: ViewerDesc) ContentsOf (): Viewers.Frame;
    END;

    Directory = POINTER TO DirectoryDesc;
    DirectoryDesc = RECORD (Objects.ObjectDesc)
      tickW, barW, tickH, barH: LONGINT;   (*dimensions of scroll-bars, tick marks*)
      scrlLeft, scrlRight, scrlBottom, scrlTop: LONGINT;   (*borders for scrollable case*)
      frmLeft, frmRight, frmBottom, frmTop: LONGINT;   (*borders for framed case*)
      headH: LONGINT;
      PROCEDURE (D: Directory) NewLayout (): Viewers.Layout;
      PROCEDURE (D: Directory) NewTrack (): Viewers.Track;
      PROCEDURE (D: Directory) NewViewer (head, body: Viewers.Frame): Viewer;

      PROCEDURE (D: Directory) NewFiller (): Viewers.Frame;
      PROCEDURE (D: Directory) NewFillerViewer (): Viewer;

      PROCEDURE (D: Directory) NewXScroller (c: Contents): XScroller;
      PROCEDURE (D: Directory) NewYScroller (c: Contents): YScroller;
      PROCEDURE (D: Directory) NewFramed (c: Contents): Viewers.Frame;
      PROCEDURE (D: Directory) NewScrollable
        (sx: XScroller; sy: YScroller; c: Contents): Scrollable;

      PROCEDURE (D: Directory) AllocNormalViewer (VAR x, y: LONGINT);
        (*determine track to open "normal" viewers in – à la Oberon user track*)
      PROCEDURE (D: Directory) AllocSpecialViewer (VAR x, y: LONGINT);
        (*determine track to open "special" viewers in – à la Oberon system track*)
    END;

    ScrollerMsg = RECORD (Objects.NotifyMsg)
      (*notify scrollers that f has scrolled*)
      f: Viewers.Frame
    END;
    LocateMsg = RECORD (Objects.NotifyMsg)
      (*locate significant sub–frame in corresponding subtree*)
      f: Viewers.Frame
    END;

  VAR
    dir, stdDir–: Directory;
```

```
    PROCEDURE OpenViewer (D: Directory; v: Viewers.Viewer; x, y: LONGINT);
      (*shorthand to alloc and create track (if necessary), alloc and open viewer,
        and restore screen area*)
    PROCEDURE SetDirDefaults;
    PROCEDURE SetLayoutDefaults;
    PROCEDURE SetMouseDefaults;
END Looks.
```

Looks follows the usual Ethos directory object conventions. Hence, it is easy to change the common "look & feel" of the system partially or in total.

3.4.8.4 Coupling of Models and Views – An Example: TextFrames

Following the MVC principle (cf. 2.3.2), data models and data views in Ethos are strictly separated. A data model is changed by invoking operations private to the model. Then, a model may send a notification message to its views informing them that it has changed. Such notification mechanisms are handled by a general mechanism in module Objects. There, it is possible to register recipients for notification messages. For example, the viewer system installs a single notifier that broadcasts every notification message that it receives to all layouts. (Layouts in turn broadcast to all their tracks, tracks to viewers, viewers to sub-frames, etc.)

Interesting examples of structures in Ethos that notify upon state changes are the file directories, module directories, texts, and any other data models provided as extensions. The notifications sent by the file and module directories can be used to display up-to-date views showing lists of files or modules fulfilling certain criteria.

Under controlled conditions it is sometimes desirable to turn the notification mechanism off. For example, a transactional modification of a model involving many small steps may take too long or cause screen flicker due to frequent updates, or both. Individual data models may take care of this problem, either by introducing some transaction concept, or – less robust, but far simpler – by providing a flag to turn notification on or off.

The typical model-view relation in standard Ethos is manifest in the text system. Just as in standard Oberon, texts are data models that notify after each modification. The typical receiver of text update notifications are text frames. In Ethos, module TextFrames defines and implements a functionally rather complete view to texts. TextFrames supports all features found in the frames of Write [Szy92a], the standard document editor of the Oberon system. As in Oberon, a frame also implements direct manipulation features, making a frame a combination of a view and a controller component.

TextFrames also defines elements derived from the text system's element type (cf. 3.4.6). Elements defined in Texts are strictly limited to the model level, i.e. have no capabilities to display themselves in a view. TextFrames extends these elements to allow for self-representation in a text frame. The protocol used to display an element has two phases. In a first phase elements are requested to compute their actual size (PrepareDisplay). The second phase then requests actual display of the element within some allocated space. The two-phase protocol allows elements to have varying sizes, e.g. depending on their contents or on the output medium.

A special element type is defined to support paragraph formatting. These elements are called paragraph controls (*parcs* for short) and affect the formatting of the text following them up to the next parc or the end of the text. Parcs and their attributes must be known to text frames if paragraph formatting should be visible. However, the exact *behavior* of parcs is irrelevant for text frames. Hence, a separate module ParcElems defines an extension and provides direct-manipulation functionality for parcs.

Module TextFrames is also interesting for the problems it causes in the context of the Ethos extension policy. Like most other Ethos services, TextFrames defines an abstract text frame and provides a directory object to request some concrete implementation of it. To extend a text frame one has to implement one anew, perhaps forwarding most interface activities to the standard implementation.

> This is just what happens when extending an Oberon implementation: A handler picks up some messages and forwards others to the standard handler implementation.

Since text frames are rather heavy-weight (have many methods and fields), it is a little tedious to implement extensions. It has been observed that common extensions to text frames do not deal with the *view component*. Instead, the *controller component* is extended to support different mouse-click or keyboard handling features. Hence, for text frames the controller component has been opened for separate extension. This is done by routing all invocations (dealing with the controller component of a frame) via a controller object installed in the text frame. The standard controller object just forwards all controller activities to the frame, while an application-specific one might filter some of the controller invocations and react upon them in a specific way. For example, a small Lisp interpreter written for Oberon [Gri91] has been ported to Ethos. The only extension made to text frames is a special interpretation of the Linefeed key: It is used to evaluate the expression to the left of the caret. All that was required to do so, was to implement a small controller and install it into the used text frames.

The typical interaction of a frame and its controller is illustrated by the following example showing the events taking place when a mouse click into

the frame is detected. (The sequence below assumes the use of a standard, i.e. inactive controller.)

1. user clicks with mouse	*module Ethos sends an input message M to the frame containing the mouse cursor*
2. F.Handle(M)	*text frame F receives input message M*
3. F.controller.Edit(F, x, y, keys)	*text frame calls controller to handle mouse input*
4. C.Edit(F, x, y, keys)	*a standard controller just forwards to its frame*
5. F.Edit(x, y, keys)	*standard edit routine locates mouse click position*
6. F.controller.TrackCaret(F, x, y, keys)	*click should cause placement of caret*
7. C.TrackCaret(F, x, y, keys)	*again, standard controller just forwards to F*
8. F.TrackCaret(F, x, y, keys)	*caret is tracked until mouse has been released*

The text frame invokes its controller whenever a control activity on the next finer level of granularity is required. For example, the message handler calls the controller to handle a mouse click. The mouse click handler calls the controller to handle a track placement operation.

The efficiency of this scheme is by no means a problem: User interaction happens at rather low frequencies. Also the response time visible when producing some tracking feedback is not affected while tracking a certain marking. In the example above, most time is spent waiting for the user in routine F.TrackCaret.

```
DEFINITION TextFrames;
   IMPORT Objects, Input, Frames, Viewers, Looks, Texts;

   CONST
      (*Parc.opts*)
         gridAdj = 0; leftAdj = 1; rightAdj = 2; pageBreak = 3; twoColumns = 4;
      (*Frame.opts*)
         scrollOpt = 0; fontOpt = 1; showParcsOpt = 2; changeMarkOpt = 3;
         flexWidthOpt = 4;
      (*SelectionMsg.id*)
         getsel = 0; setsel = 1;

   TYPE
      Context = RECORD (Objects.ObjectDesc)
         (*block of parameters repeatedly passed to elements*)
         attr: Texts.Attributes; pos: LONGINT; x0, y0, indent: LONGINT
      END;

      Elem = POINTER TO ElemDesc;
      ElemDesc = RECORD (Texts.ElemDesc)
         PROCEDURE (E: Elem) PrepareDisplay (VAR con: Context);
         PROCEDURE (E: Elem) Display
            (VAR f: FrameDesc; VAR con: Context; VAR eFrame: Viewers.Frame);
            (*by returning eFrame # NIL, element E may install a subframe*)
```

```
    PROCEDURE (E: Elem) Edit
        (VAR f: FrameDesc; VAR con: Context; VAR x, y: LONGINT; VAR keysum: SET);
    PROCEDURE (E: Elem) Focus
        (VAR f: FrameDesc; VAR con: Context; eFrame: Viewers.Frame);
        (*user focussed subframe eFrame of E for editing*)
    PROCEDURE (E: Elem) Defocus
        (VAR f: FrameDesc; VAR con: Context; eFrame: Viewers.Frame);
        (*user defocussed subframe eFrame of E*)
END;

Parc = POINTER TO ParcDesc;
ParcDesc = RECORD (ElemDesc)
    first, left, width, lead, lsp, dsr: LONGINT;
    opts: SET;  nofTabs: INTEGER;
    tab: ARRAY 32 OF LONGINT
END;

Location = POINTER TO LocationDesc;
LocationDesc = RECORD (Objects.ObjectDesc)
    (*a location in model space and its related position in view space*)
    org, pos: LONGINT; x, y, dx, dy: LONGINT    (*origin org of line containing pos*)
END;

Controller = POINTER TO ControllerDesc;
ControllerDesc = RECORD (Objects.ObjectDesc)
    PROCEDURE (C: Controller) ConnectTo (VAR f: FrameDesc);
    PROCEDURE (C: Controller) TrackLine
        (VAR f: FrameDesc;
            VAR x, y: LONGINT; VAR org: LONGINT; VAR keysum: SET);
    PROCEDURE (C: Controller) TrackWord
        (VAR f: FrameDesc;
            VAR x, y: LONGINT; VAR pos: LONGINT; VAR keysum: SET);
    PROCEDURE (C: Controller) TrackCaret
        (VAR f: FrameDesc; VAR x, y: LONGINT; VAR keysum: SET);
    PROCEDURE (C: Controller) TrackSelection
        (VAR f: FrameDesc; VAR x, y: LONGINT; VAR keysum: SET);
    PROCEDURE (C: Controller) Call
        (VAR f: FrameDesc; pos: LONGINT; new: BOOLEAN);
    PROCEDURE (C: Controller) Write
        (VAR f: FrameDesc; ch: CHAR; VAR attr: Texts.AttributesDesc);
    PROCEDURE (C: Controller) EditElem
        (VAR f: FrameDesc; VAR x, y: LONGINT; VAR keysum: SET);
    PROCEDURE (C: Controller) Edit
        (VAR f: FrameDesc; x, y: LONGINT; keysum: SET);
END;
```

```
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Looks.ContentsDesc)
   text–: Texts.Text; org–: LONGINT; opts–: SET;
   focus–: Viewers.Frame; controller–: Controller;
   hasCar–, hasSel–: BOOLEAN; carLoc, selBeg, selEnd: Location;
   defParc: Parc; time: LONGINT;
   PROCEDURE (VAR F: FrameDesc) InstallController (c: Controller);
   PROCEDURE (VAR F: FrameDesc) Init (t: Texts.Text; org: LONGINT; opts: SET);

   PROCEDURE (VAR F: FrameDesc) ThisPos (x, y: LONGINT): LONGINT;
   PROCEDURE (VAR F: FrameDesc) ThisSubFrame
      (x, y: LONGINT): Viewers.Frame;
   PROCEDURE (VAR F: FrameDesc) PassSubFocus (f: Viewers.Frame);
   PROCEDURE (VAR F: FrameDesc) RemoveSelection;
   PROCEDURE (VAR F: FrameDesc) SetSelection (beg, end: LONGINT);
   PROCEDURE (VAR F: FrameDesc) RemoveCaret;
   PROCEDURE (VAR F: FrameDesc) SetCaret (pos: LONGINT);

   PROCEDURE (VAR F: FrameDesc) ParcBefore
      (pos: LONGINT; VAR parc: Parc; VAR pbeg: LONGINT);
   PROCEDURE (VAR F: FrameDesc) LineOrg
      (text: Texts.Text; pos: LONGINT; VAR org: LONGINT);
   PROCEDURE (VAR F: FrameDesc) ParagraphOrg
      (text: Texts.Text; pos: LONGINT; VAR org: LONGINT);

   PROCEDURE (VAR F: FrameDesc) ShowFrom (pos: LONGINT);
   PROCEDURE (VAR F: FrameDesc) Update (VAR msg: Texts.NotifyMsg);

   PROCEDURE (VAR F: FrameDesc) TrackLine
      (VAR x, y: LONGINT; VAR org: LONGINT; VAR keysum: SET);
   PROCEDURE (VAR F: FrameDesc) TrackWord
      (VAR x, y: LONGINT; VAR pos: LONGINT; VAR keysum: SET);
   PROCEDURE (VAR F: FrameDesc) TrackCaret
      (VAR x, y: LONGINT; VAR keysum: SET);
   PROCEDURE (VAR F: FrameDesc) TrackSelection
      (VAR x, y: LONGINT; VAR keysum: SET);

   PROCEDURE (VAR F: FrameDesc) Call (pos: LONGINT; new: BOOLEAN);
   PROCEDURE (VAR F: FrameDesc) Write
      (ch: CHAR; VAR attr: Texts.AttributesDesc);
   PROCEDURE (VAR F: FrameDesc) WriteElem
      (e: Texts.Elem; VAR attr: Texts.AttributesDesc);

   PROCEDURE (VAR F: FrameDesc) EditElem
      (VAR x, y: LONGINT; VAR keysum: SET);
   PROCEDURE (VAR F: FrameDesc) Edit (x, y: LONGINT; keysum: SET);
END;
```

```
        Directory = POINTER TO DirectoryDesc;
        DirectoryDesc = RECORD (Objects.ObjectDesc)
            tickW, barW, left, right, bottom, top: LONGINT;
            altLeft, altRight, altBottom, altTop: LONGINT;
            frmOpts, scrlOpts: SET;
            PROCEDURE (D: Directory) New (t: Texts.Text; org: LONGINT; opts: SET): Frame;
            PROCEDURE (D: Directory) NewFramed
                (t: Texts.Text; org: LONGINT): Viewers.Frame;
            PROCEDURE (D: Directory) NewScrollable
                (t: Texts.Text; org: LONGINT): Looks.Scrollable;
            PROCEDURE (D: Directory) NewMenu
                (name, menu: ARRAY OF CHAR): Texts.Text;
        END;


        SelectionMsg = RECORD (Objects.NotifyMsg)
            (*get, set selection text[beg, end); get valid if time ≥ 0*)
            id: INTEGER; text: Texts.Text; beg, end, time: LONGINT
        END;
        CopyOverMsg = RECORD (Objects.NotifyMsg)
            (*ask receiver to copy over text[beg, end) to its caret position*)
            text: Texts.Text; beg, end: LONGINT
        END;
        ExtendMsg = RECORD (Objects.NotifyMsg)
            (*look for an extension frame candidate to extend selection over frame boundaries*)
            src, ext: Frame
        END;

    VAR
        dir, stdDir-: Directory;
        defParc: Parc;

    PROCEDURE GetSelection (VAR text: Texts.Text; VAR beg, end, time: LONGINT);
    PROCEDURE SetSelection (text: Texts.Text; beg, end: LONGINT);
    PROCEDURE SetDirDefaults;
END TextFrames.
```

An interesting detail is the capability of TextFrames to *set* selections in all views displaying a certain text. This can be used to implement simple feedback mechanisms for operations modifying parts of a text.

The implementation of standard parcs is contained in the separate module ParcElems listed below. ParcElems also extends the type defined in TextFrames by adding methods to set and get the attributes of a parc. These methods are designed to be extensible such that extended parcs may add new attributes.

```
DEFINITION ParcElems;
   IMPORT Objects, Input, Frames, Viewers, Texts, TextFrames;

   TYPE
      Parc = POINTER TO ParcDesc;
      ParcDesc = RECORD (TextFrames.ParcDesc)
         PROCEDURE (P: Parc) SetAttr
            (VAR f: TextFrames.FrameDesc; pos: LONGINT; VAR r, s: Texts.Rider);
            (*scan attribute settings using s; produce log output using r*)
         PROCEDURE (P: Parc) GetAttr
            (VAR f: TextFrames.FrameDesc; VAR r, s: Texts.Rider);
            (*scan attribute requirements using s; produce output using r*)
      END;

   PROCEDURE ParcExtent (t: Texts.Text; beg: LONGINT): LONGINT;
END ParcElems.
```

3.4.9 System Configuration and The Bootstrap Process

The potential of the Ethos system is based to a large extent on its ability to be configured on all levels. The ubiquitous approach taken is that of replacable default objects. Default objects are often directory objects that deliver objects of a certain type. Of course, a configuration should be automatically re-established whenever booting the system. To avoid introducing some kind of control language, a special module Config is used: During the boot-strap process (cf. below), Config gets control and establishes the wished configuration.

Often it is required to construct a configuration by using a certain extension to handle special cases, while another (or the default) should handle all other cases. A typical approach to this problem is what might be termed the *"piggy-back" style of programming*: All functions are implemented as an open *list of handlers*, perhaps even with priorities. The resulting configuration depends on the order of handler invocations which is not explicit. Also, in general, the automatic chaining of handlers is not flexible enough. Instead, each handler should have a single hook used to refer to a default handler. Then it is under control of that handler when it hands over to the default. Module Config is then used to construct explicit chains of extensions, where this is required. The following example lists an almost minimal module Config, leading to a standard configuration with a module Busy installed. The only other thing that Initial does is loading module System and causing a default viewer setting to be opened.

Procedure Initial is known *by name* to module Ethos. Invoking Initial as a command allows to re-establish the initial configuration as long as correct

command invocation is still possible. This is quite helpful when experimenting with new configurations. (To be complete, Initial should invoke all SetDefault procedures of modules where the configuration has been changed.)

```
MODULE Config;
   IMPORT Ethos;

   PROCEDURE Initial*;
      VAR res: INTEGER;
   BEGIN
      Ethos.Call("System.DefaultViewers", NIL, FALSE, res);
      Ethos.Call("Busy.Start", NIL, FALSE, res)
   END Initial;

END Config.
```

Bootstrap.
Booting the Ethos system happens in a series of stages. The idea behind introducing stages is an increase in flexibility: Each boot-stage needs a certain burned-in knowledge about the next higher boot-stage. The finer grained the boot-stages are, the less knowledge needs to be burned into the low stages. The following list describes the actions performed during each of the stages. After boot-stage 1 the module loader is present and initialized. Thus, all higher boot-stages are simple and essentially ask the loader to install some higher module known only by name at that stage.

- *Boot-Stage 0.* A mechanism outside of Ethos loads modules Devices, Objects, Files, and Modules. Usually, these four modules are prelinked into a special *boot-file.* Afterwards, control passes to a special entry of the first module, i.e. the low-level module Devices.
- *Boot-Stage 1.* Module *Devices* initializes the four initial modules in the mentioned order. Devices contains a mechanism that performs an up-call to an installed main procedure whenever the system starts or restarts. As part of its initialization actions, Modules installs a first main procedure. After initializing all four modules, Devices starts the system, i.e. calls the installed main procedure.
- *Boot-Stage 2.* Module *Modules* gets control. Modules is the standard module loader and hence is now able to load and initialize arbitrary modules. Modules loads module Ethos. Ethos is the central scheduling instance for all user interactions: It contains the *main loop.* As part of its initialization Ethos installs a main procedure and returns.

- *Boot-Stage 3.* Module *Ethos* gets control. Ethos installs a new main procedure, the main loop, and uses Modules to load module Config. (If loading of Config fails, Ethos loads module System directly and establishes the minimal startup configuration.)
- *Boot-Stage 4.* Module *Config* gets control. The purpose of Config (which is available in source form to every user) is to establish an initial configuration. Usually, Config loads module System and invokes a command System.DefaultViewers to open a startup set of viewers available to the user after the system has booted. Also, Config may load any number of other modules required to establish the startup configuration.
- *Boot-Stage 5.* Unless Config caused another main procedure to be installed, control is passed to the main loop in module Ethos. The system is now up and running, awaiting user input.

At first glance it is not clear why a module first installs a main procedure, then returns to Devices, only to immediately get control again. The reason is robustness. If the initialization of a higher boot-stage module terminates exceptionally, the system falls back into module Devices, removing all modules loaded during the current loader invocation (cf. 3.4.3.1). Since each boot-stage completes by returning to Devices, an exception leaves the so far established system intact. This is used to fall back to a standard configuration when the user defined configuration is inconsistent.

3.5 Examples of Extending Ethos

A quick recapitulation of the Ethos extension model follows, for a full discussion cf. Section 2.4.4. The guiding principle is to avoid extension of concrete classes [JF88]. In general, if a class should be based on the implementation of an existing concrete class, this is done by means of *forwarding*. The more straight-forward approach of using method overriding in conjunction with super-calls has many disadvantages. While it seems to improve code re-use, this misses the point [Mag91]. It is felt that the far more important effect of code re-use is that of *re-use of client code*: The dynamic binding of methods makes it possible to implement client-code that works generically on all sorts of implementations of a certain abstract class.

Using direct inheritance to extend a concrete class abolishes part of a system's extensibility. For example, one might introduce an extension of text frames that is a significant improvement over the standard implementation. Then one installs a directory object returning these new text frames. This should have an immediate effect on all applications that thereafter use the directory object to get a text frame. However, if an application uses a concrete text frame implementation as a basis for an extension, this application will not profit from the new text frame implementation. Instead, one has to replace the extended implementation.

The method put forward in Ethos to minimize extension of concrete classes leads to a demand for an easy delegation construct. The proposed forwarding technique uses a record field to refer to an object of the class methods should be forwarded to. An abstract *nut-shell class* can be provided, where a nut-shell class implements all methods such that invocations are forwarded to an instance variable. Such a nut-shell class can be used to minimize the coding-effort when implementing concrete forwarding classes. The arising problem are self-invocations: If the object forwarded to performs a recursive invocation of one of its own methods, this cannot be intercepted by the forwarding object. The more general concept that can be used to solve such problems is called *delegation*: Other than with forwarding, self-invocations return control from the delegatee to the delegator. (For a definition of delegation as opposed to forwarding, cf. 1.1.1 or [JZ91]). However, a clean use of delegation requires language support not present in Oberon-2 (cf. 4.4).

This section discusses a wide variety of extensions of the Ethos system. Some of the extensions have been implemented to validate the extension model and the interface definitions. However, often an extension is supposed to closely follow the pattern of an existing part of the system. Then, for the sake of total project time, the extension is only discussed but has not been carried

through to its very end.

3.5.1 An Alternate File Directory Model

Typical directory objects in Ethos are file directories. Whenever an old file is to be retrieved or a new file to be generated, a file directory object is consulted. The standard directory object maintains a single, flat name space on a single device (the boot disk). However, the constraints to having a flat directory or to supporting just one device are *not* built into Ethos, but are mere restrictions of the standard implementation.

This section covers two extensions of the file directory: A hierarchical name space and installable file-systems. To make things more interesting, the implemented service follows the "no new languages" paradigm. Instead of having a language (or an interactive facility) to define hierarchical directories and to mount file systems, Oberon-2 is used to set-up the wished configuration. (Of course, in practice one could easily add some interactive facility dealing with the same programming interface.)

Introducing a non-flat name space seems simple at first hand. Indeed, it is simple to replace the flat name space by a hierarchical one. The problems wait at a different end: When introducing a hierarchical name space one also has to introduce a "current directory", such that applications not aware of the hierarchy will function in a meaningful way. Also, one has to introduce a list of search-paths, such that the system will find certain files, independent of their placement in the hierarchy. The latter point can be illustrated by considering that Ethos is based on dynamic module loading. Hence, it is important that object files are found even if they are not located in the current directory. However, forcing all object files into a single, known place renders the hierarchical name space useless (simple prefixing conventions would do then). For example, one might want to maintain two different versions of a system. Then it is important that the *same* module exists in two different sub-directories and that it is easy to select which one to inspect when this module is sought.

Besides modules, there are many other objects that depend on availability of files. For example, the font directory object seeks for pre-rastered fonts and the text directory object seeks for existing texts. Hence, having a single "current search path" will not do. Also, it is desirable to scan multiple search-paths in a certain known order. Then it is possible to create a new version of something by copying it to a search-path searched before the one leading to the old version. If the new version shall be dropped for some reason, it suffices to delete it: The old version will again be visible.

The implementation of a hierarchical name space pursued in this section is based on a new directory type derived from the directory type defined in module Files. Such a directory contains a list of sub-directories and a reference to a host directory. The sub-directories are used to form the hierarchy, while the host directories are used to access actual file-systems. Figure 3.31 illustrates the situation: Directory *d* has sub-directories *d0*, *d1*, and *d2*; directory *d1* has sub-directories *d10* and *d11*; all directories share a common host directory.

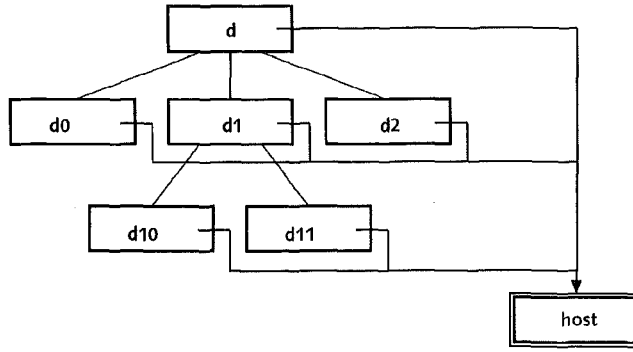


Figure 3.31 – A Tree of Directories mounted to a Single Host Directory.

Hierarchical names are resolved by matching certain patterns within file names (see below) and forwarding the request to the appropriate sub-directory. Once a name has native form, i.e. does not denote a path leading to one of the sub-directories, it is forwarded to the host directory. Hence it is possible to resolve to final names on every level of the directory tree.

To fully utilize a hierarchical name space it is useful to introduce some naming conventions such that native, absolute, and relative naming of files are possible. Native naming means that the file name should be taken as is and used to access the host directory. Relative naming means relative to some path prefix that is set for a particular directory. Path prefixes are used to create search paths when retrieving existing files, or to place new files using some "current" prefix. The following simple naming conventions have been chosen, where relative and absolute paths follow the conventions of UNIX:

$/d_0/d_1/\ldots/d_n$/name	*absolute path*
$d_i/d_{i+1}/\ldots/d_n$/name	*relative path*
%name	*native path*

The "%"-prefix for a native path is required to disambiguate it from a relative path having no directory prefixes (i.e. "$d_i/d_{i+1}/...$ $/d_n$/name" degenerates to "name"). This is useful if the host directory again uses structured file names, e.g. to allow for path names like "$/d_{00}/d_{01}\%/d_{10}/d_{11}$/name".

When a directory request to a directory d has the absolute form "$/d_0/d_1/...$ $/d_n$/name", "/d0" is removed and the request forwarded to the sub-directory d_0 of d. A request to d based on a native name is directly forwarded to d's host directory. The treatment of relative names depends on the directory operation performed. If an existing file is sought, a directory uses a list of search prefixes (called the directory's *context prefixes*) to expand the relative name. If a new file is to be registered, a different prefix (called the directory's *current prefix*) is used for expansion. Finally, absolute names of the form "/name" are expanded using a directory specific prefix (called the directory's *prefix*) and forwarded to the host directory. Expanding a path-name means prefixing it with some other partial path-name.

In a common set-up the hierarchy of directories is established by the configuration module. The only interactive operation usually applied is the modification of the current prefix of the top-level directory. (The operation of changing this prefix may be compared to the UNIX *cd* shell command.) All operations of a hierarchical directory are based on string manipulation of path names and request forwarding to either other hierarchical directories or to host directories. This makes the hierarchical directory extension flexible. For example, using different host directories for different sub-directories of a directory tree enables the support of multiple file-systems. In fact, adding a new sub-directory with some file-system's directory installed as its host is all that need be done to "mount" a file-system. Unmounting the file system is then simply done by removing the sub-directory again. An example for the usage of an alternate host directory is given in the next section.

> One of the hazards of unmounting file-systems in traditional systems is the possibility that references to files of that file-system are still active. In the model proposed in this section this is not a problem. The unmount process described above simply removes the file-system from the active name space. Thereafter it is no longer possible to retrieve files from or allocate files in that file-system. A physical removal of the file-system is also possible, but requires more care: Since files are self-contained objects they exist until finalization. The finalization of the last open file of a file-system can then be used to safely remove the file-system itself. Also, the list of opened files is available through the file-system's identity directory. Hence it is also possible to explicitly flush and invalidate all existing file references. Then the file-system can be removed immediately.

The definition of directories is not restricted to tree-like structures. In general, any directed graph is possible. For example, if d1 is a sub-directory of d it is well possible that d is again one of the sub-directories of d1. Also, d1 may be one of its own sub-directories. Using such network relations among directories can be quite convenient to ease navigation in a larger directory

structure. (In UNIX, '.' and '..' paths refer to the current and the current's father directory, respectively. This is just a special case of cyclic sub-directory relations.)

Module Directories implements the hierarchical naming strategies explained above. Its interface is shown below. The context prefix list has been implemented as an open array of names (*context*) that is increased by doubling its length each time all entries have been used up. It may be interesting to note that the complete implementation takes only about 200 lines of code.

```
DEFINITION Directories;
   IMPORT Files;

   CONST
      (*DirNotifyMsg.op*)
         init = 10; setCurrent = 11;
         addContext = 12; removeContext = 13;
         installDir = 14; removeDir = 15;

   TYPE
      Contexts = POINTER TO ARRAY OF Files.PathName;

      Directory = POINTER TO DirectoryDesc;
      DirInfo = POINTER TO DirInfoDesc;
      DirInfoDesc = RECORD (Objects.ObjectDesc)
         next: DirInfo;
         dirname: Files.PathName;
         dir: Directory;
      END;

      DirectoryDesc = RECORD (Files.DirectoryDesc)
         nofCon–: INTEGER; context–: Contexts;
         current–: Files.PathName; prefix–: Files.PathName;
         host–: Files.Directory;
         PROCEDURE (D: Directory) Init (prefix: ARRAY OF CHAR; host: Files.Directory);
         PROCEDURE (D: Directory) SetCurrent (path: ARRAY OF CHAR);
         PROCEDURE (D: Directory) AddContext (context: ARRAY OF CHAR);
         PROCEDURE (D: Directory) RemoveContext (context: ARRAY OF CHAR);
         PROCEDURE (D: Directory) InstallSubDir
            (dirname: ARRAY OF CHAR; dir: Directory);
         PROCEDURE (D: Directory) ThisSubDir (dirname: ARRAY OF CHAR): Directory;
         PROCEDURE (D: Directory) RemoveSubDir (dirname: ARRAY OF CHAR);
         PROCEDURE (D: Directory) GetSubDirs (prefix: ARRAY OF CHAR): DirInfo;
      END;
END Directories.
```

The possibility to remove context prefixes and sub-directories is usually not used. Instead, the whole directory structure is constructed at once using some Oberon-2 code typically found in module Config. Having the inverse operations of adding a new context prefix and installing a new sub-directory is useful, though. For example, one might add an interactive interface that allows for direct graphical manipulation of the directory structure. To support such applications, Directory extends the DirNotifyMsg defined in Files such that all new directory operations cause proper notifications.

3.5.2 A Simple Remote File System

The discussion of flexible directory schemes (previous section) already indicated the possibility of having multiple file-systems available at any one time. Figure 3.32 illustrates a simple setup, where a common main directory contains a sub-directory. These directories refer to two different host directories. A simple example using such a topology is a diskette file system temporarily mounted into the main directory structure. In turn, files on the diskette can be maintained and accessed just as usual files on the disk. (On many machines one has to take care of the possibility of diskettes being removed by the user at any one time.)

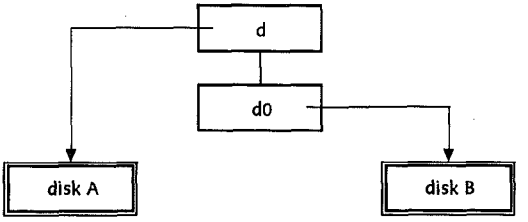


Figure 3.32 – Mounting a Second Disk Using a Subdirectory.

Since host directories are treated like standard file directories, they can again handle structuring conventions. For example, *disk B* might hold a Macintosh file system, where hierarchical naming is usually indicated by means of the separator ":" (instead of "/"). Then the path name "/d/d0/a:b:c" refers to the Macintosh path "a:b:c". Of course, using the prefix scheme explained in the previous section one could easily have multiple directories pointing to the same host directory, but referring to different folders within that host directory. This way it is possible to create a homogeneous naming scheme covering a variety of underlying host naming schemes. In the example, the Macintosh naming conventions would become invisible and the file "c" could be reached by using the path "/d/d0/a/b/c".

More interesting than the support of additional disks is the support of file-systems of a totally different nature. As an example, a simple remote file-system is studied. To avoid the subtleties of consistent updates the implementation is based on a trivial copy/copy-back scheme: If a remote file gets requested a copy is sent over the network. Whenever a remote file is flushed or registered it gets sent back. If concurrent copy-backs happen, there is no guarantee on the order of updates, except that a copy-back will either win, or lose completely, i.e. the surviving update is consistent.

A remote directory is created upon request by a special directory object of module RemoteFiles. Method This takes a remote address as its parameter and tries to establish a network connection. If this is done successfully, a file-directory object is returned that gives access to files on the remote machine. As at the time of completion of this thesis the network support for Ethos was not yet completed, RemoteFiles has not yet been worked out completely. The definition below serves as an illustration.

```
DEFINITION RemoteFiles;
   IMPORT Objects, Files, NetLink;

   TYPE
      Directory = POINTER TO DirectoryDesc;
      DirectoryDesc = RECORD (Objects.ObjectDesc)
         res: INTEGER;
         PROCEDURE (D: Directory) This (site: NetLink.SocketAdr): Files.Directory;
      END;

   VAR
      dir, stdDir–: Directory;
END RemoteFiles.
```

As another interesting application, directory objects may be used to implement *semantic file systems* as suggested in [GJS91]. The idea is to allow certain contents-related queries for files, where *virtual directory names* are used to achieve compatibility with existing directories. As Ethos directory objects have full control over the evaluation of search strings when retrieving files, semantic file systems should be straight-forward to integrate into Ethos.

3.5.3 Adding Threads

Many modern operating system introduce threads (cf. section 3.4.4) as the most fundamental concept of multi-programming. However, it has been observed before (e.g. [Wir77]), that co-routines are a much more fundamental construct that can be used to create thread-supporting libraries. In fact, threads can be seen as "co-routines plus pre-emption".

In Ethos, not even co-routines have been made available on a low level. (This has been done for the same reasons as in Oberon: Multi-programming on single-user machines introduces far more problems than it solves.) Yet there are examples where a functional thread package would be helpful. The most important area are services offered to other machines over some network. There is no direct way to detect that a remote machine has crashed. Hence, timeouts are used to guarantee termination of functions waiting for a remote reply. However, if a machine has no thread-concept, it becomes hard to guarantee that the machine will reply within some bounded time (e.g. [Szy90a]). If such a guarantee cannot be made (with some acceptable probability), the use of timeouts becomes questionable: For a remote machine there is no way to distinguish between a machine that is busy for a long time and one that has crashed.

In Ethos all provisions have been taken to allow introduction of co-routines. Since a co-routine has as its essential structure a private stack, it must be possible to announce the existence of such a stack to the garbage collector. This can be done by registering the stack as a reference-block (cf. section 3.4.4). The garbage collector (cf. section 3.4.2) treats reference-blocks as arrays of potential pointers. Each of these potential pointers is checked for plausibility and then verified against the actual heap-structure. Surviving pointer candidates are traced to mark the then potentially reachable objects. This is just the technique used by the garbage collector to trace references contained in the standard stacks, e.g. the user and supervisor stacks.

The only other hurdle when implementing co-routines as an "after-thought", i.e. when adding co-routines as an extension to the system, is the protection against stack overflows. For the standard stacks provisions have been taken to detect stack overflows. In principle, the processor should guard a stack against overflows by means of a stack-fence register. Second best, a memory protection scheme, e.g. a memory management unit, can be used. (The Chameleon machine [HP92] has a simple page protection scheme that is used for this purpose.) If no hardware support is available, the compiler could issue stack check instructions – a relatively expensive thing to do. In any case, if provisions for flexible stack-overflow detection are given, it is easy to implement co-routines as an extension to Ethos.

Having co-routines, multi-programming based on cooperative scheduling is possible. For the timing critical problems discussed above, this is not enough, though. Instead, a scheduler needs to be installed that performs pre-emptive scheduling among a set of co-routines (then becoming *threads*). Such a pre-emptive scheduler can be implemented by means of an engine provided by module Tasks (cf. section 3.4.4).

Module Threads does just this; it introduces a new scheduler type, derived

from that defined in module Tasks. A task set-up to execute under such a scheduler will be executed in a threaded fashion, pseudo-concurrently with other threads, other engines, and the main system's thread. To make module Threads a little bit more interesting, another module has been added that implements signals (similar to those defined in [Wir84]). Using the simple monitor construct of module Tasks, it is easy to implement other synchronization constructs, such as Semaphores [Dij65].

Note that the creation of new threads is done by merely setting a task (derived from Tasks.Task) to a thread scheduler. The scheduler then creates a new dispatcher and allocates resources (like the calling stack) for the thread. Control is passed to the new thread by calling the Do method of the attached task. (If Do ever returns, the task is suspended.)

```
DEFINITION Threads;
   IMPORT Tasks;

   TYPE
      Scheduler = POINTER TO SchedulerDesc;
      SchedulerDesc = RECORD (Tasks.SchedulerDesc)
         PROCEDURE (S: Scheduler) Pass;
      END;

   VAR sched, stdSched–: Tasks.Scheduler;

   PROCEDURE SetDefaults;
END Threads.

DEFINITION Signals;
   IMPORT Objects, Tasks;

   TYPE
      Signal = POINTER TO SignalDesc;
      SignalDesc = RECORD (Objects.ObjectDesc)
         PROCEDURE (S: Signal) Send;
         PROCEDURE (S: Signal) Enqueue (t: Tasks.Task);
         PROCEDURE (S: Signal) QueueLen (): INTEGER;
      END;

      Directory = POINTER TO DirectoryDesc;
      DirectoryDesc = RECORD (Objects.ObjectDesc)
         PROCEDURE (D: Directory) New (): Signal;
      END;

   VAR dir, stdDir–: Directory;

   PROCEDURE SetDefaults;
END Signal.
```

The following example for creating and using threads shows an implementation of the Sleeping Barber algorithm.

```
IMPORT
   Signals, Threads;

TYPE
   Consumer = POINTER TO RECORD (Tasks.TaskDesc) END;
   Producer = POINTER TO RECORD (Tasks.TaskDesc) END;

VAR
   nonempty, nonfull: Signals.Signal;
   in, out, n: INTEGER;
   buf: ARRAY 100 OF CHAR;

PROCEDURE (C: Consumer) Do;
   VAR base: Threads.Scheduler; dst: CHAR;
BEGIN base := C.base(Threads.Scheduler);
   LOOP DEC(n);
      WHILE n < 0 DO nonempty.Enqueue(C) END;
      dst := buf[out]; (*consume dst*) out := (out + 1) MOD LEN(buf);
      IF n = LEN(buf) THEN nonfull.Send END
   END
END Do;

PROCEDURE (P: Producer) Do;
   VAR in1: INTEGER; src: CHAR;
BEGIN
   LOOP INC(n);
      WHILE n > LEN(buf) DO nonfull.Enqueue(P) END;
      (*produce src*) buf[in] := src; in := (in + 1) MOD LEN(buf);
      IF n = 0 THEN nonempty.Send END
   END
END Do;

PROCEDURE Kick;
   VAR p: Producer; c: Consumer;
BEGIN n := 0; in := 0; out := 0;
   nonempty := Signals.dir.New(); nonfull := Signals.dir.New();
   NEW(p); Threads.sched.SetTask(p, FALSE); p.Resume;
   NEW(c); Threads.sched.SetTask(c, FALSE); c.Resume
END Kick;
```

The standard thread scheduler implemented in Threads follows a 50% heuristics: At most 50% of the processor cycles are spent executing threads, while the rest is reserved for the main thread. This ensures sufficient reactivity of interactive applications. Also, the use of WHILE loops to establish conditions is necessary for two reasons. Firstly, the implementation of Signals is simple-minded in that it does not seize the processor when a thread calls Enqueue, but simply returns after ensuring that the thread is in the queue of the corresponding signal. Secondly, the semantics of the implemented Send

operation is that it does wake up the longest waiting thread, but that it does not affect scheduling. Hence, live locks due to priority inversions are avoided, but there is no guarantee that a resuming thread finds the condition that it was waiting for still established.

3.5.4 Extending the Text Model

Module Texts, defining the standard text model, has been introduced in section 3.4.6. The standard text model can be extended in several ways, e.g. using

- new element implementations
- new text attributes
- overlayed structures maintained by text operations

The implementation of new elements is the easiest way. Adding a new element does not interfere with any of the existing elements, and all elements can be used in any combination to form a document. This is the recommended way of extending the Ethos text model.

Introducing new attributes is more involved. It is necessary to implement a new concrete text class that understands and maintains the new attributes. Also, it is not possible to use two different attribute extensions in conjunction. (This would lead to a multiple-inheritance problem to re-join the two separate attribute record definitions.) Often it is better to introduce new attributes by means of *control* elements. The paragraph controls defined in module TextFrames (cf. section 3.4.8) are a good example. Then the convention is applied that an attribute extends from a control element to the next of the same kind. If no element controls a certain text stretch, a default value is chosen.

More general than added attributes are overlayed structures. An overlayed structure maintains some additional information relative to the edited text. For example, such a structure may be used to implement a folding editor. The overlayed structure then indicates beginning and ending positions of a text fold. An extended editor then can "open" and "close" such a fold, by displaying or hiding it. Again, the Ethos text model allows for such extensions but one must be aware that it is rather hard to re-combine two different extensions of that kind. As for attribute extensions it is preferable to use special elements to solve the problem. For example, to implement folding one can use pairs of fold-elements that delimit fold stretches.

More interesting extensions of the text model involve extensions of the editing model. A standard text can be edited freely by means of arbitrary insertions and deletions. However, it may be interesting to provide an extended text implementation that defines some kind of protocol for elements affected by editing operations. For example, one could allow elements to prevent deletion, or to modify deletions to consistent ranges. Then it is possible to have pairs of elements that cannot be deleted individually. Also, it may be interesting to notify all elements in a text of all changes applied to the text. This is useful when elements need to re-calculate their contents upon certain text changes. Finally, a text may be modelled as a non-sequential structure, e.g. a network of linked text portions. Then, an extension of the text model may support such non-linear structures and provide a (modifyable) linear projection onto the structure. This may be used to define repeatedly occuring text fragments only once, automatically maintaining consistency when one of the exemplars is edited.

All these extensions tend to degrade performance of the text implementation significantly. However, opening possibilities for experimenting with clever implementations of such extensions is at the very heart of the Ethos project.

As the Ethos text system was the seed for developing the Oberon Write system [Szy92a], the extensions available for Write are considered shortly. For a more detailed description of the extensions and their applications consult the Write documentation (e.g. [Szy91]). The equivalent extensions are expected to be easily applicable to Ethos. However, the large user base of Oberon lead to far more useful extensions for Write than could be re-implemented for Ethos in the course of this project.

3.5.5 Changing the User Interface Model

The standard user interface of Ethos is implemented in modules Viewers (3.4.8.1) and Looks (3.4.8.3). Viewers defines the primary organization of screen space using layouts, tracks, and viewers. Looks defines refinements of the viewer concept and introduces a directory object. This directory object is the anchor for changing the Ethos user interface.

On the one hand, the graphical looks may be changed. This is done by re-implementing the Looks abstractions Viewer, Scroller, Scrollable, and the like. On the other hand, the "feel" may be changed by interpreting mouse and keyboad input in a different way. For the latter kind of modification, no framework is present within Ethos. Indeed, the proper decoupling of user interaction from program models is still an open research issue (e.g. [Mar91]).

3.5.6 Remote Pixelmaps – Printing

Pixelmaps have been introduced in 3.4.7.1 as the bottleneck interface to all two-dimensional pixel-oriented devices. The bottleneck has been designed with efficient decoupling of remote devices in mind. For example, it was anticipated from the beginning that printing is merely drawing to a frame connected to a pixelmap representing a printer. Whether this "printer" is actually a virtual one mapped to a screen (previewing), a true local device, or a remote printing service is fully transparent to the implementor and user of a frame class.

Figure 3.33 illustrates the decoupling of a remote device using a pair of special pixelmaps and frames. The idea is that an arbitrary client frame is connected to a special pixelmap. The pixelmap in turn communicates with a special frame, possibly located on a different machine. Finally, the special frame is connected to the target pixelmap. As a result, the client frame seems to operate directly on the target pixelmap. (This holds, as long as neither the client frame, nor the target pixelmap use type tests to query the actual type of the underlying pixelmap or the connected frame, respectively.)

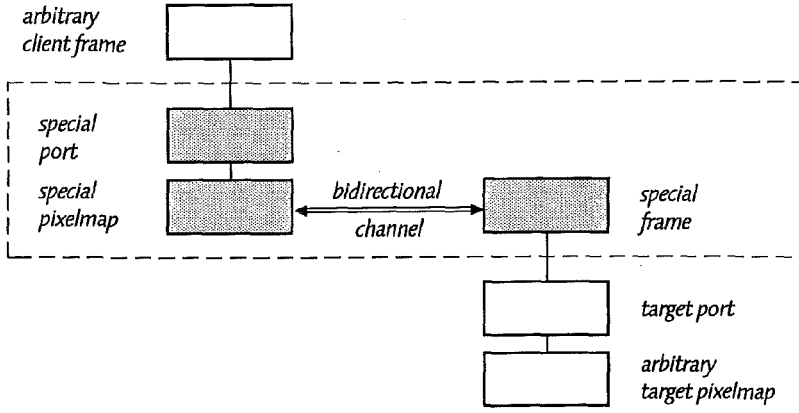


Figure 3.33 – Decoupling (Possibly Remote) Pixelmap Devices.

The bidirectional channel is used to transport coded requests and replies corresponding to the operations of the pixelmap bottleneck interface. The channel needs to be bidirectional, since pixelmaps offer query functions, e.g. used to map colors between universal and device-specific codes.

The decoupling toolkit consisting of the special frames and pixelmaps can also be used to decouple parts coexisting on the same machine. For example, a background thread (3.5.3) may produce results by drawing into an arbitrary frame. Then, to avoid synchronization conflicts with the single-threaded user

interface of Ethos, an idle task (3.4.4) may be used to periodically update the screen.

3.5.7 Adding a New Abstraction – Graphics

The only data model fully supported in the minimal Ethos configuration are texts. However, adding a new data model to the system is rather easy. For example, to add a graphics subsystem it is merely necessary to provide modules Graphics, GraphicFrames, and Draft. These modules have similar functionality as the ones of the text system (Texts, TextFrames, Write).

The implementation of a graphics subsystem for Ethos has almost been completed, starting with the Draft editor for Oberon (by Robert Griesemer).

3.6 Porting Ethos

One of the earliest goals set when defining the Ethos project was portability. From the experiences with the Oberon system it was clear that portability needs to span both, source code and data files. The former to allow quick ports to new machines, the latter to enable data exchange among versions running on various machines. While the former goal was met quite well by the Oberon system [BCF+92], the latter has been almost ignored when designing Oberon.

3.6.1 What is Portable? What is Not?

In principle, porting Ethos means rewriting hardware-dependent parts. Most of these are concentrated in the foundation modules Devices, Objects, Files, and Modules. Module Devices needs be rewritten almost completely, while modules Objects, Files, and Modules contain only few hardware-dependent details. The loader in module Modules needs to be adapted if the target machine has a different processor-architecture or if the used compiler generates a different object-file format. The remaining dependencies may be found in the modules residing in the Device Abstraction Layer (cf. section 3.3.1) where default implementations of device drivers are located. This includes modules Frames, Input, and NetLink. Again, these modules contain clearly separated hardware-dependencies and are otherwise portable.

The interfaces of all modules have been carefully designed to avoid hardware dependencies. This holds also (within limits) for the module Devices at the bottom of the Ethos system. In other words, it should be

possible to rewrite Devices for other machines without changing its interface. It is clear that such a goal is difficult to achieve and it must be admitted that the expressed optimism is only supported by the study of a few contemporary workstation architectures. However, while Devices is imported by some modules in the Ethos system, it is never re-exported. Hence, changing its interface is a mere matter of inconvenience but keeps higher-level interfaces intact.

3.6.2 Effort Required when Porting Ethos

Ethos has been ported within five days to a DECstation running Ultrix (the port requires further polishing though). The porting scheme applied follows the one used for the Oberon system [BCF*92]. Major difficulties have not been encountered for Ethos. A port to the Chameleon machine [HP92] is planned, but has not reached a definite state. A port of the Oberon system to Chameleon has been completed, and the low-level parts implemented to port Oberon have already been designed with specific demands of the Ethos port in mind.

3.6.3 Experiences with Porting Ethos to a New Machine

The abstractions provided by Ethos proved easily portable. A single minor exception is the pattern pointer contained in the Frames.Pattern descriptor. For a port to X-Windows this pointer has been replaced by a longint (pointing to a C structure defined by X).

3.7 History of the Ethos Project

Within the tight time-line of the Ethos project – less than two man-years – strict planning of certain project phases was necessary. At the same time the original project goals required as many decisions as possible to be left open as long as possible. The resulting project phases can be seen as a mixture of milestones and evolutionary refinements. Some of the strategies applied to prototype the system from its earliest stages up to a version running on a bare machine are interesting in themselves. Also, the process of turning vague ideas into trusted implementations is of some interest.

3.7.1 Phases of Project Evolution

The Ethos system has been developped in five major steps (often called *milestones*), each being subdivided into many smaller refinement steps. The major steps are characterized by a re-engineering of the whole system: Each time the complete system was frozen in its current state, and development of code started over again. Of course, whenever a design situation reoccured that was found to be solved by implementations of an earlier step, source code was carefully copied, instead of being rewritten. The following graphics shows the modules involved in each of the first four major steps; the modules that resulted from the fifth and (so far) last step have been described at length in subsection 3.3.1.

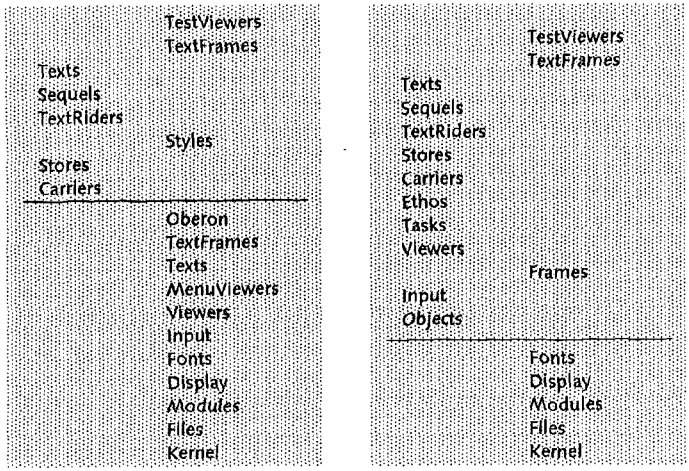


Figure 3.34 – Ethos Evolution: Bootstrapping from Oberon.

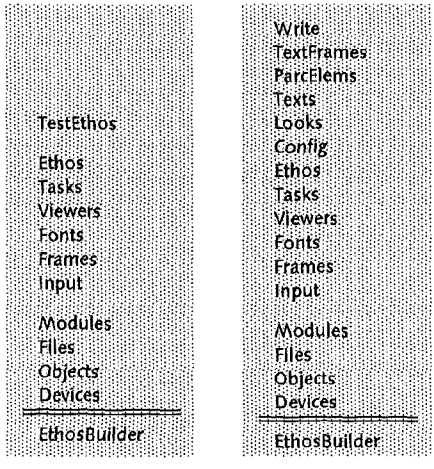


Figure 3.35 – Ethos Evolution: Self-contained system.

The first two versions were built on top of the Oberon system. For this purpose, the Oberon system can be divided into two parts: One part – the Oberon *core* – contains hard-wired abstractions like the memory management, the file system, and the module loader. The other part, forming the majority of the Oberon system, contains replaceable and (to some extent) extensible abstractions, like application specific models, views, and command packages.

To prototype early versions of Ethos, the Oberon core was maintained, while replacing everything on top. This was done by stepwise reducing the dependency on Oberon modules. During these steps, the requirements for the lower parts became more precise, and in the fourth step the Oberon system was completely replaced by low-level Ethos modules. The third step led to the first bootable, but minimal version of Ethos. This required implementing a tool to make boot files, called EthosBuilder and executing under Oberon (cf. 3.7.2 below).

Re-introducing the higher-level parts of the second step into the bootable version led to step four. Finally, all the lessons learned while engineering this version were used to improve some of the design principles and to redesign the entire system following the improved principles.

3.7.2 Bootstrapping the System on a Bare Machine

The boot mechanism of Ethos on a bare machine has been described at length in subsection 3.4.9. The important point is that the boot strap starts by loading a file of fixed and simple format from an disk or diskette into the main memory. This is done by a small boot loader program burned into a read-only memory (ROM) of the machine. The ROM has not been changed over the one used to boot Oberon. Hence, the same boot file format was used, and Oberon and Ethos could be booted interchangably on the same machine.

The boot file consists of a sequence of blocks, each prefixed by an absolute memory address. The boot loader copies each of the blocks to the given absolute address. Finally, the boot file contains an entry point – again an absolute address – that is used by the boot loader to jump to the loaded program.

BootFile = {Address Length {byte}$_{Length}$} StartAddress 0.

To construct a boot file the tool EthosBuilder is used. EthosBuilder takes a sequence of object file names, reads each of these files and creates a boot file.

After each major change to the system, EthosBuilder was used to assemble a self-contained boot file containing scaffolded modules. Then, by testing and accepting modules from the bottom individually, a running system was re-established. It is important that EthosBuilder can combine any number of modules into a boot file. Hence, it is easy to test components like the file system or the module loader, since they need not function just to have testing modules available.

In an early step the Devices module was designed and implemented. Its most essential features were tested by directly writing test patterns to the bitmapped display. As soon as these were up and running, minimal input and output primitives were implemented to support test scaffolding of other modules. Already in an early version simple support for character output to the display was provided. This was used to produce readable traces of the stack and processor state after catching hardware exceptions (traps). It was crucial to have readable and informative output in early stages to avoid the burden of "low-level debugging" whenever possible.

It may be noteworthy that the garbage collector was not installed until after most of the system was completed. Instead, an early version of the collector was implemented and tested within the running Oberon system. It was found that for a garbage collector there are mainly two possibilities: Either it runs

perfectly, or it is quite easy to cause some significant defect in the running system. This is only true, if the system has reached sufficient complexity to produce a complete set of test cases. Hence, it was easy to test the garbage collector by installing an almost correct version into the otherwise complete system.

> Testing the garbage collector took about two days. Almost all the time was spent searching for a mysterious error that caused collection of objects that were still reachable from the stack, but not otherwise. To trace pointers in the stack (cf. subsection 3.4.2), all values that look like heap references are sorted into an array. Then, this array is compared against all allocated blocks, marking blocks that are reachable from the stack.
>
> Finally, the error was located in a Heapsort procedure that was copied from another program, implemented by someone else. While this sorting procedure was about the only part of the collector that was trusted, it was also the only part that contained a significant error!

3.7.3 Acquiring Confidence Into New Implementations

While it is obvious that one has to acquire confidence into a new implementation, the methods used to achieve that goal are often left unmentioned. The scientifically rather weak field of "Software Engineering" led to the impression that these methods are imprecise, repetitions of common knowledge, or the like. However, it is felt that the importance of the topic legitimates a short discussion. In this section it is tried to cover some of the important points in a rather general setting. The discussion may be seen as a closing remark on the Ethos project.

There is one guiding principle of utmost importance:

Have a single bug at a time.

In other words, it is important to gain confidence in small steps. This is by no means a new rule, but the one that is most often violated. The combinatorial difficulties of a system can only be mastered when the components seen individually are trustworthy. If one follows the rule thoroughly the location of errors is almost always clear, i.e. in the moment an error gets detected its cause is likely to be found within a small part of the whole system.

In order to trust individual subsystems two things need to be done. Firstly, it must be clear what the subsystem is *supposed* to do, i.e. some kind of specification is required. Secondly, the actual implementation must be validated for actually implementing the specification. In principle, a formal specification and a formal proof would do the job best. This is at least true for

small subsystems with tricky semantics.

For larger systems with modest complexity the formal approach often leads to specifications longer than the actual implementations. Also, there is *no* general way of completely validating a program: For example, the equivalence of a program and its formally proven counterpart is undecidable in the general case. A method to attack this problem is the hand-in-hand development of a program and its proof [Dij65][Dij76][DS90]. For delicate parts of programs, such as subtle algorithms applied to a certain data structure, this can be regarded part of the state-of-the-art.

The covered area is often referred to as "programming in the small", compared to "programming in the large", where the requirement is added to abstract sub-systems into compact units of reasoning. In the area of side-effect free languages (e.g. functional languages) this has been achieved. In languages based on the imperative paradigm, including most existing object-oriented languages, no sufficiently mighty methods exist. At the heart of the problem is the concept of procedural abstraction: A procedure – other than a function – leaves side-effects in the global state. For "pure" object-oriented languages the global state is clearly separated into objects, enabling a better control over side-effects. The same is true for modularized systems, though. Still, the change of global state *is visible* to other objects and thus the underlying paradigm remains imperative.

Besides formal specification a weaker tool exists: A sufficiently powerful type-system. The typing information added to a program can be seen as a subset of a formal specification. Constraints enforced by the type system are predicated over the identifiers used in the various signatures. A well typed program carries a large part of its specification in-place, i.e. closely related to the program code. It is important to have a *sound* type-system, i.e. one where the predicates implied by assigning certain signatures to components are guaranteed by a combination of compile-time and run-time checking. Also, all predicates implied by the type-system should be *decidable*. Having a well typed program based on a sound and sufficiently strong but still decidable type-system allows the compiler to *validate* the part of the program specified by the typing information!

A general warning may be in order at this place: Many existing languages add type information that is not sound, not decidable, or simply not completely checked. Examples are covariance problems in Eiffel [Mey88] or unsafe pointer arithmetic in C [KR78]. For such languages, the type-system adds the *illusion* of having a compiler checking part of a program's correctness, while this is actually not done or even impossible. It is doubtful whether such typing information is better than no type information at all.

Obviously, a type system has its limits. In theory, the limits are reached at the point of undecidability. In practice, they are reached far earlier, as a type system must remain masterable by the programmer. Of course, the constructs of a type system should be orthogonal. (The existence of separate record and class constructs in many object-oriented languages is a clear violation of this principle.) Designing a good type-system is a trade-off between expressiveness (what can be specified?) and conciseness (how many separate type concepts exist?). Hence, the most important question to be asked is: Is the set of expressible predicates worth adding another type concept? There is no easy answer to this question, and the evolution of typed language families exhibits a constant addition and removal of type features. For example, the languages ALGOL 60 [Nau60], EULER [Wir63], SIMULA [DN66], SIMULA 67 [DMN68], ALGOL 68 [WMP*69], PASCAL [Wir71], Modula [Wir77], Ada [DoD80], Modula-2 [Wir82], Oberon [Wir88b], and Modula-3 [Nel91] reveal the interesting play of forces between opposing answers to the before mentioned design question.

Run-time checks should be added by the language compiler wherever the static checking does not suffice to guarantee certain properties. Of course, run-time checks cannot guarantee that a program is correct, but they can guarantee that it is *safe*. Safe means that the program cannot possibly invalidate certain global system invariants. Among these are arbitrary side-effects due to dangling pointers, stack overruns, type casts, and out of bounds array accesses. Since in many systems such conditions are neither caught by the compiler nor by run-time checks, the amount of "debugging" caused by introducing only a few such errors is enormous [Gri91b]. For example, a system supporting arbitrary pointer structures should either disallow explicit deallocation (and therefore needs garbage collection), or it should guarantee the absence of dangling references (as can be done for certain functional languages). To conclude, the language and its implementation are an important point to consider when trying to write correct programs efficiently. For the short implementation and testing times taken by the Ethos project, the choice of the language Oberon-2 and the use of a highly reliable compiler [Cre91] were absolutely crucial.

Run-time checks are often misunderstood. Generally, a run-time exception should be avoidable by the programmer. To do so, the offending statement needs be guarded properly. For example, to prevent index out-of-bounds exceptions, an indexed access can be guarded with a range condition. Also, if such a guard is present a smart compiler can optimize the code by not issueing a redundant run-time check. The same holds for all the run-time checks listed above. However, there are problems where run-time checks are absolutely unacceptable to solve a problem. A typical example is the co-variance problem: Often one wants to claim that a certain method parameter of a sub-class has a sub-type of the type declared for that parameter in the super-class. If a type-system does not allow to express covariant type changes, e.g. to allow static

checkability, it is tempting to simply use a type guard in the method. If the sub-class method is then called with a parameter of the type defined for the super-class method, a run-time exception results. This is not the way to do it, as there is no way for the *caller* of the method to prevent this situation: Late binding of methods makes this impossible. For the sake of an appropriate language construct it has been done in Ethos at a few places. Possibilities of a clean solution to this problem are discussed in the conclusions, Section 4.4.

Despite program validation, *testing* a program is important: Unless the program got generated fully automatically (and specification *and* transformer are known to be correct), there is always the chance of trivial errors like typing mistakes or identifier misnomers. In practice, testing is even more important as the feat of treating large systems in a largely formal manner is still a pipe-dream.

However, testing is by definition (and for non-trivial programs even in principle) non-exhaustive. Hence, it cannot be used to validate a program, but only to falsify it. The quality of the testing process depends highly on the attitude taken when testing a program. Testing should be done in a destructive way, i.e. the goal should be to find some way to falsify the program. Often it is claimed that some testing instance independent from the original programmer should be consulted. This is found only partially helpful: In principle, no other person than the original programmer has such a deep understanding where things can go wrong. Testing ones own programs then requires the attitude of not letting a single error pass through. This goal in turn leads to a different attitude when writing programs: abstractions on all levels are conceived such that they are testable. (The most important rule is to minimize interdependencies, such that individual parts can be tested individually.)

Bearing invariants in mind becomes the guiding programming principle. Only if the invariants maintained by a certain abstraction (a procedure, a class, a module, a sub-system) are clear, it is possible to fulfill the task of implementing the abstraction. This is best accomplished if the concept captured by a certain abstraction follows from the meaning of the name given to the abstraction. (Badly chosen names are one of the biggest hurdles when trying to master a complex system.) Testing the implementation of an abstraction is best done in a fashion where the tester knows the implementation details. (Such tests are sometimes referred to as *white-box* tests.) By testing implementations bottom-up spotted errors are most likely to be located in the implementation level currently under suspicion.

Testing programs can be considered an art in itself. While it is obvious that one actually does not want to master the subject of testing but would prefer to replace it by verification, the practical needs of testing are ubiquitous. In the following, a few test methods are summarized that proved successful during

the development of Ethos.

The simplest and still most effective method is *program scaffolding* [Bro75]. Scaffolding means the addition of otherwise unneeded code just to support testing of code under development. In its essence, this is meant when answering the question for the need of having "debugging" tools with "essentially we all use print statements to do the debugging". Scaffolding is best done on the module level: A special code section is appended to construct test scenarios and invoke the various procedures and methods of that module. Then the module to be tested is used as temporary top-level module, i.e. by implementing a trivial user-interface for interactive testing. Later, when the module gets mature and its implementation is trusted, the scaffolding code can be removed again. (Sometimes one might prefer to just comment it out for later use.)

A difficulty arises when it becomes necessary to recheck a module, say *M*, that has already been trusted and built upon. One could re-scaffold that module, but the resulting change of the module interface invalidates the client modules. It is likely that in such a situation the scaffolding code of the module *M* should be invoked by the scaffolding code of some module higher up in the import hierarchy, say *S*. A simple solution to this problem is the addition of an auxiliary module *A* that then is imported by *M* and *S*. *A* is used to couple the scaffolding parts of *M* and *S*, cf. Figure 3.36. This way a module low in the hierarchy can be tested using all the functionality available (and trusted) far up in the hierarchy *without* changing any of the existing interfaces, hence avoiding massive recompilations and therefore shortening the test cycle time.

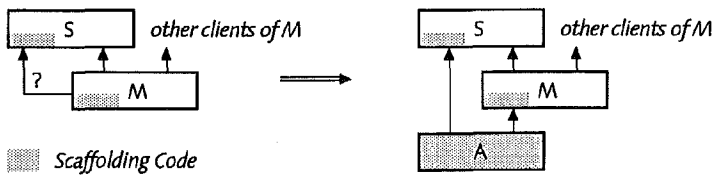


Figure 3.36 – Auxiliary Scaffolding Module.

Finally a note on so-called "Debuggers" may be in order. From the experience gained during the Ethos project it became apparent that traditional debuggers do not help. Traversing heap structures is tedious and simple inspection routines written on the fly but geared towards the inspected structure are often more flexible. What is required however is an indication where a program failed if an exception caused its termination. Experience showed that a simple stack dump (essentially the sequence of procedure activations) suffices. This is what has already been provided in the Oberon system.

4 Conclusions

The questions at the core of every thesis are:

- What has been learned?
- What can be learned?

In this section it is tried to answer these two questions by assessing the Ethos project and by drawing conclusions that point into the future of Object-Oriented Programming Languages and Operating Systems.

4.1 Was it Worth the Effort?

Since the early Lisp and Smalltalk systems, designs were driven by the idea to get rid of the traditional separation of operating system and application. The goal has been and still is the extensibility on all levels of a system. However, all early systems where built on the basis of untyped languages. Untyped programs have only weak static semantics, i.e. semantics derivable from the program text without executing the program. This gets worse if the untyped language is purely object-oriented. Then every operation is potentially subject to late binding. Also, most Lisp dialects and Smalltalk fail to support a proper module concept. The absence of types and modules makes these systems complex and hard to manage, i.e. basically unsuitable for designing large systems.

Such problems of the early approaches can be mostly overcome by using a language supporting modules and a strong type system. However, only recent language developments fulfill these requirements. Therefore, it was felt justified to develop the *new* operating system *Ethos* using the recent programming language *Oberon-2* [MW91]. To avoid re-inventing too many wheels, Ethos follows many of the concepts of the *Oberon* system.

The extension potential of Ethos is better compared to systems like Smalltalk than to traditional operating systems, where a strict boundary between applications and operating system exists. Other than most existing operating systems, including Oberon, Ethos supports prototyping of low-level components within the running system. For example, it is possible to extend the heap manager or the module loader, where the latter may be useful when exploring new object file formats. The organization of all services into classes allows *extensions on all levels*, while the use of polymorphic typing in

conjunction with late bound procedures and dynamic integration facilities enables the *integration of extensions* into *the running system.* For parts of the Oberon system, in particular for viewers and for extensible objects defined in the text and graphics subsystems, this is also possible. However, other parts of Oberon – especially the low-level core – are realized in a traditional, non-extensible fashion.

A possible point of criticism is the lack of support for access protection, multi-programming, multiple processors, and distributed computing. These are interesting fields, and they are tightly related to the way a system is designed. Still, it is felt that the concentration on a single-user workstation operating system was justified, as the narrowed scope allowed for a more substantial treatment of the remaining problems. Also, some of these aspects can be added as extensions to the existing system, including a certain degree of multi-programming and transparent support for remote resources. However, for a future project it would be interesting to reconsider aspects of concurrency and protection.

4.2 Is the Price Paid Justified?

Is the price paid in terms of reduced performance and increased complexity justified? The most crucial decision made when designing Ethos was to use a single language and a single extension model on all levels of the system. This places a heavy burden on the language but simplifies implementation drastically. On the one hand, the language had to support low-level programming for foundation modules and device drivers. On the other hand, higher-level modules had to be implemented in an expressive and type-safe way. For example, in the original Ceres implementation of the Oberon system [WG88], Assembly language was used to implement the lowest foundation module Kernel, as well as to implement the critical device driver for the bitmapped display. However, the increased speed of recent hardware and the improved code quality achieved by using the Oberon-2 compiler [Cre91] justified the coding of the whole system using the single language Oberon-2.

> Although the Ethos raster operations support clipping to rectangular bounds, which the Oberon version does not, the Ethos raster operations coded using Oberon-2 are between 20% slower and 40% faster (!) than the Oberon version coded in Assembly language. (Precise measurements were found almost impossible due to caching effects.) The lesson learned was that the use of a high-level language permitted easy re-arrangements of code without introducing subtle errors, hence making it far easier to tune frequent cases than when using Assembly language. (Another lesson learned is that for typical raster operations the overhead of low-level rectangular clipping can be neglected.)

The second decision, i.e. to use a single extension model, was found to be more critical. Using an object-oriented approach even for low-level services of the system introduces significantly more indirections into the Ethos system, than present in, say, the Oberon system. The most critical interface is that provided by the Stream type and its subtypes, as typical invocations on objects of type Stream ask for reading or writing a single byte or a few bytes. Hence, the overhead paid for using the fully general Carrier/Rider scheme applied to byte streams is significant (about 20 to 80% compared to Oberon). However, it was felt tolerable in the execution context of the running system (often less than 20%), which normally does more than just reading and writing single bytes.

For current language technology it is indeed critical to support extensibility by means of late binding on the level of primitive system functionality. The results are tolerable, but in order to achieve a performance that compares well to less extensible systems, a different language implementation technique is required. For example, in-line caches might be used to automatically detect and optimize the case where a late-bound operation is invoked frequently with the same binding. This technique is essential for implementations of pure object-oriented languages where even integers are objects [DS84] [HCU91][CU91], but it could as well be applied to hybrid object-oriented languages.

4.3 How Difficult are Extensions to do? Where are the Limits?

The level of difficulty experienced when trying to extend a system depends on both the complexity perceived when trying to comprehend the system, and on the degree of structure imposed on extensions of the system. The former aspect is a matter of size and structure of the original system, where it helps to carefully use modules and types to organize a system. The latter aspect is of more fundamental nature: How can a system be designed to allow for a maximal extension potential? Of course, the empty system has the largest extension potential, as it does not enforce anything. However, for a system to be meaningfully extensible, it is necessary that extensions are (or can be) just as extensible as the original system. Hence, the extension problem has an inductive nature and aspects of orthogonality and composability are to be considered. To manage such problems it is necessary to find and define principles that can be used to create extensible structures in a uniform way.

The Ethos project aimed at the two-fold goal of investigating and finding useful design principles while at the same time validating these principles by actually applying them to the design of the Ethos system. Whether this approach was successful can only be judged by a few experiences made when

actually extending the system. For a meaningful assessment of the simplicity and potential of extending Ethos, a longer period of study and a broader range of users would be necessary. (The Ethos system has mainly been extended by the author, but additional experiences stem from porting some applications from Oberon to Ethos, where the ports were jointly done by the author and one of his colleagues (Robert Griesemer), who also implemented these applications.) The experience gained so far is nevertheless encouraging: The complexity of the overall system seems masterable due to the two-level structure imposed by modularization and typing.

The Ethos extension model (cf. 2.3, 2.4) is quite rigid and departs from the tradition of object-oriented systems in *not* promoting code inheritance, and thereby in hindering direct code re-use. Instead, the emphasis is on subtyping and the re-use of client code due to sharing of polymorphic interfaces. The limits of this extension model are set by the limits of message *forwarding*: Whenever resorting to another implementation to handle default cases, forwarding instead of superclass invocation is recommended. However, forwarding to another object changes the identity of the receiver object, and therefore severely restricts the potential of self-invocations. A way out of this dilemma is the support of *delegation* instead of forwarding. However, simulating delegation by means of forwarding in conjunction with an additional auxiliary parameter [JZ91] leads to tedious interfaces and calling conventions. A support for delegation in the language would ease the situation significantly. However, it is not at all clear how this can be done in an efficient yet type-safe manner (cf. 4.4.3).

4.4 What can be Learned?

The design principles developed in the course of the Ethos project, as well as the actual structure chosen for the Ethos system may be used as a starting point for a similar project. While these points have been covered extensively in the previous chapters, this section adds concrete conclusions drawn for various components of a system, covering the processor and machine architectures, the language design and implementation, and the software modules. Of course, it is not claimed that everything that, say, a processor should provide is listed. Instead, features are mentioned that are typically missing in current solutions. Naturally, this section does not present definite results, but hints at possible future work.

4.4.1 What a Processor Architecture should Provide

The processor should support the checking of properties that need be checked to guarantee safety, and that need be checked at run-time due to static undecidability. Such conditions are: Stack overflows; dereferencing of unbound pointers (value NIL); numbers out of range when used to index arrays, to execute a case switch, or coerce to types defined over a smaller range. Stack overflows can be caught using a stack limit register. (This is so simple that it is unbelievable that nearly no processor supports it; notable exceptions being the Lilith [Ohr84] and Hyperstone [Hyp87] processors.) NIL dereferencing should be caught when computing an address base [Hyp87], e.g. by means of a hint to the used compute instruction that a certain value should not become zero, or by means of a test and trap instruction that avoids conditional branching and may be implemented using imprecise exceptions [Dec92].

The processor should support an unstructured and unconstrained (flat) address space to avoid special cases in the software. Likewise, the processor should not introduce special protection schemes like distinguishing between user and supervisor modes. (A small set of privileged instructions used to actually extend the processor architecture within a trusted micro-kernel may be acceptable, though.) It should be possible to define domains within the single global address space and assign certain access tags to such domains. Then, the processor traps when accessing an address within a domain which holds a tag that does not match the one currently held by the processor.

4.4.2 What a Machine Architecture should Provide

The machine should support single address space architectures, where everything available is mapped to unique addresses. To implement protection in hostile environments, it is useful to support protection ranges within the linear address space. In fact, recent work shows that the traditional protection schemes based on memory management units introduce unnecessary costs into the critical path of every memory access [KLC*91][WS92][Pou91]. The remaining purpose of a memory management unit is the mapping of a (single) large logical address space to a smaller physical one.

4.4.3 What a Language Should Allow / must not Allow

Systems. A language should support strong typing and explicit support for modularization [Szy92b]. To allow for nested structuring of larger systems, it would be helpful if a module can be used to encapsulate a group of modules without introducing efficiency problems. The concept of separating systems and modules – where systems are kind of super-modules used to group modules and other systems – proposed by Cardelli [Car89] is not recommended. Instead of introducing another concept, the module concept itself should be made flexible enough. For example, a module should be able to re-export arbitrary parts of other modules.

Subtyping/Subclassing. To avoid conflicts between subtyping and subclassing hierarchies – where subclassing is merely used to inherit code, while subtyping expresses conformance relations – the language should distinguish these two concepts (as is done in Emerald [Hut87][Bla91]). For example, a plain subtyping hierarchy, as present in the language Oberon, could be extended to allow binding of operation signatures to types. However, concrete implementations of such operations are not bound to the type, but to a class. A class then is nothing but the implementation of a specific type, and multiple classes may implement a single type. This avoids the misuse of subtyping to introduce implementation alternatives.

In a next step, the subclassing mechanism may be introduced. For a class *C* implementing a type *T* derived from the base type *T0*, this can be done by allowing *any* implementation of *T0* to be bound to become the base class of *C*, cf. Figure 4.1. The concept can be made more powerful, by supporting *semi-dynamic superclasses* bound at object creation time, or *dynamic superclasses* changable throughout an objects lifetime.

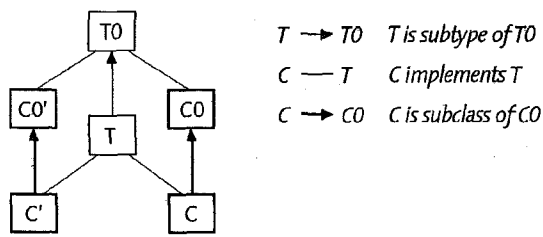


Figure 4.1 – Subtyping vs. Subclassing.

If a class C' implements a type T' being an arbitrary subtype of T0, it would be especially interesting to allow C' to be used as a semi-dynamic or even dynamic base class of C, cf. Figure 4.2. Doing so requires separate instances of class C' and to delegate messages from an instance of C to an instance of C'. This way, the advantages of delegation could be combined with those of strong typing. However, it is not clear how or if this proposal can be realized in a type-safe yet efficient manner.

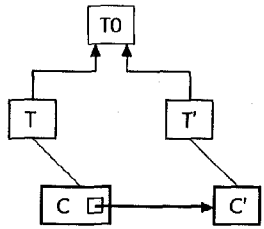


Figure 4.2 – Fully Dynamic Subclassing using Delegation.

Covariant Subtyping. Another crucial aspect is the correct treatment of *covariant* subtyping of in- and in/out-parameters of operations. A trivial solution is to forbid covariance in these situations by using a type system based on contravariance, as has been done for Oberon-2 [MW91]. However, such a type system is not expressive enough to describe certain requirements. Another solution is the separate introduction of *genericity* to express certain covariant subtyping conditions in the case of homogeneous collection types. Another – but incorrect – possibility has been chosen for Eiffel [Mey88], where the construct *like current* allows to express that an argument of a method should be of the *same* type as the receiver. This is merely a way to express genericity in an unsafe manner [Coo89] since it does introduce the covariance problem. A global type check of a program can avoid covariance problems at run-time [Mey92], but is not a feasible solution for extensible systems (cf. 1.1.2).

Loopholes. To be a replacement for most hardware-based protection mechanisms, a language implementation should be type-safe. To make this possible, the language itself must not define constructs that cannot be implemented in a type-safe manner without introducing an unacceptably high run-time overhead, i.e. constructs that are doomed to introduce loopholes into the implementation.

In the case of the languages Oberon and Oberon-2, a few loopholes are known that should be eliminated from the language definitions. First of all, the WITH construct applied to designators of pointer type is unsafe, as illustrated in the following code example.

```
TYPE
   P = POINTER TO R;
   R = RECORD END;
   P1 = POINTER TO R1;
   R1 = RECORD (R) x: INTEGER END;

PROCEDURE F;
   VAR p: P; p1: P1;
   PROCEDURE G;
   BEGIN NEW(p) END G;
BEGIN NEW(p1); p := p1;
   WITH p: P1 DO
      p.x := 0;  (*legal, since p has been guarded to P1*)
      G;         (*this destroys the assertion of the WITH guard*)
      p.x := 42 (*havoc!!*)
   END
END F;
```

A possible solution would be to disallow the application of WITH to pointers, to remove WITH from the language altogether, or to introduce additional type guards wherever the compiler cannot statically guarantee that the regional guard still holds. The first alternative seems to be the best and the third the worst choice.

Another loophole of current versions of Oberon and Oberon-2 is the special compatibility rule for formal reference parameters of type ARRAY OF SYSTEM.BYTE. As explained earlier (2.4.1), this feature should at least not be used within module interfaces, unless the corresponding modules are declared low-level and unsafe. Even better, the special compatibility rule could be eliminated from the language definitions. (Recent versions of the Oberon and Oberon-2 compilers at least emit warnings when a module uses this unsafe compatibility rule.)

Finally, a loophole introduced by the system implementation results from the ability to unload a module. If this is done in a careless way, dangling procedure references may result, cf. 3.4.3.2.

4.4.4 What a Compiler should do

Assuming that the language definition includes a strong type system that *allows* for a type-safe implementation, the compiler (plus the run-time system, if necessary) should actually *provide* for a type-safe implementation. To do so, the compiler should detect the use of uninitialized variables, especially in the case of reference types (pointers, procedure variables). If for some variables this is statically undecidable – or the compiler complexity gets unacceptable – these variables should be initialized to some safe value, e.g. NIL for reference

types.

The original Oberon compiler/system [WG88] causes global and heap allocated variables to be initialized, but does not initialize variables allocated on the stack. This was done for efficiency reasons: The machines used for the first Oberon implementation were to slow to tolerate the increased procedure activation costs. Not initializing stacked variables was possible as the garbage collector was only invoked when the stack was empty and that the garbage collector implementation assumes that all pointers either point to a valid object, or are unbound (NIL). However, severe side-effects can result, if a program erroneously dereferences an uninitialized pointer. Therefore, the Oberon-2 compiler [Cre91] improved the situation by also initializing stack allocated pointer variables.

In conjunction with conservative garbage collection techniques applied to the stack [Tem91] (cf. 3.4.2) a loophole in the current Oberon and Ethos implementations exists (detected by Marc Brandis), which could be fixed by the compiler. The following code fragment illustrates the situation. (For the original Oberon implementation this is not a problem, as the garbage collector is only invoked when the stack is empty.)

```
PROCEDURE F;
   VAR p: POINTER TO RECORD x, y: INTEGER END;

   PROCEDURE G (VAR x: INTEGER);
   BEGIN
      p := NIL;  (*eliminate last pointer to object holding the actual parameter bound to x*)
      ...        (*cause garbage collection*)
      x := 42    (*havoc!!*)
   END G;

BEGIN NEW(p); G(p.y)
END F;
```

A possible fix would be the maintenance of an invisible ("shadow") pointer to the endangered object. Such a pointer needs to be set only if the compiler cannot guarantee that a pointer will survive anyway. Another possibility would be to loosen the rules to decide whether a value on the stack is potentially a pointer by marking a block even if a value points *into* the block, instead of to the block origin address. Experiments showed that this approach leads to a significant increase in pointer candidates, thereby slowing garbage collection down. Finally, the problem can be solved by not using conservative collection at all. Then, stack frame descriptors must be used that are either anchored by pushing an additional tag (likely too costly), or by using the return addresses stored in each stack frame [Gol91].

4.4.5 What Proper Modules should provide

Modules are a language issue, as far as the definition (or absence) of the module construct is concerned. Besides its availability, the proper use of a module construct is equally important. For a system to be trustworthy, all interfaces of common use should be safe, where it is helpful to have a way to express safety of an interface in terms of the programming language (e.g. using a pseudo-module SYSTEM, as is done in Oberon).

Another issue are provisions for extensibility, where especially modules providing extensible services should have a safe interface. Having a safe interface, a module can maintain and guarantee certain invariants that no possible client of the module can invalidate. This is achieved by encapsulation, e.g. using abstract data types. On the other hand, to allow for extensions, a module cannot encapsulate everything, as fully encapsulated abstractions (like abstract data types) are just not extensible. The conflict can be circumvented if a module provides means for extensions that again can establish and maintain (enforce) invariants, while being forced to respect invariants of the extended abstraction.

A possible approach utilizes type- or class-bound operations in conjunction with read-only variables: An extension can override the implementations of these operations to enforce its own invariants, while it has to call the original versions of the base implementation to modify the read-only variables. Therefore, invariants of base and extension are maintained simultaneously. Applying this scheme repeatedly leads to *safely extensible extensions* (sic!). This is precisely what a proper module should provide: safely extensible abstractions, that potentially are themselves extensions.

4.5 The Future

Many hints on what could be done in future projects are given in the previous sections of this chapter. Hence, this final section concentrates on the possible future of the Ethos project itself, by asking two questions:

- What is missing?
- How could one proceed?

The main neglect of the Ethos design is the missing support for object protection beyond the static limits that can be set using the module and type system of the language. This hinders the robust support for multiple clients sharing parts of the objects of an Ethos system. Likewise, concurrency and especially distribution aspects are only partially supported.

A possible way to proceed would be the improvement of the meta-programming support, allowing to intercept individual accesses to objects, or to all objects of a certain class. A problem is how this can be done efficiently if support for objects on a medium granularity is to be continued. A possible approach could be the introduction of *proxy objects* [Sha86] that take the place of objects in specific situations. A proxy object looks and behaves as if it were the represented object, while actually filtering and forwarding to it. Wherever a normal object does the job, no extra costs are introduced, but whenever an object needs to be accessed across certain boundaries (machines, protection domains), a proxy can be passed instead. Hence, proxy objects could be used to solve protection issues and to transparently support concurrency. A proper integration of the proxy concept might require (partial) integration into the language. Also, a useful and extensible coding scheme for access rights needs to be established, where the capability concept [TMR86] might serve as a starting point.

Another issue are the default implementations of services in Ethos. Some of these are rather heavy-weight by means of algorithmic complexity and source code size. Introducing new implementations that in principle could build on the default implementations are sometimes hindered by the strong modular encapsulation. However, simply exporting all building blocks can easily violate safety. For example, the file system implementation contains a B-tree implementation. Exporting the B-tree on the same level as the file system could allow to manipulate the file directory without respecting file system invariants. A better way would be to introduce modules with non-public interfaces that can be provided as building blocks for certain extensions. (For a possible language support cf. 4.4.3.) Thereby implementations can be structured into frameworks, as has been done in the Choices project [CRJ87].

A critical point of the Ethos design is the reliance on forwarding for extensions in conjunction with a programming language that was designed with a different extension model in mind (i.e. inheritance). A future Ethos version might be based on a refined language that directly supports concepts of forwarding and delegation.

Finally, Ethos needs further steps of evolutionary refinement and concept validation. Many more applications should be implemented, and the set of design principles should be taken further. For certain concepts a cleaner foundation should be sought. An example is the abstraction from the physical processor, which is essentially not present in the current Ethos system. The integration of a micro-kernel design [Gie90] could be helpful, where it remains a somewhat open issue what to put into such a kernel.

Bibliography

The bibliography follows several goals. First of all, it is a complete list of material referenced from within the body text. (The sections referencing a particular entry are listed in curly brackets.) Secondly, the bibliography itself is partially commented. Comments have been added when the plain title of a referenced work does not indicate why it is interesting in the scope of this thesis. (An uncommented but rather complete bibliography may be found in [Sha92].)

ABB*86 M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanlan, M. Young. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer USENIX Conference*, Atlanta, GA. Jul. 1986. {2.2, 3.4.4}

Ado85 Adobe Systems, Inc. *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA. 1985. {3.4.7, 3.4.7.3}
PostScript as an example of a "bottleneck" interface to graphical output devices.

AH87 G. Agha, C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In: B. Shriver, P. Wegner [eds.]. *Research Directions in Object-Oriented Programming*, 49–74. The MIT Press, Cambridge, MA. 1987. {3.4.4}

Ame87 POOL-T – A Parallel Object-Oriented Language. In: A. Yonezawa, M. Tokoro [eds.]. *Object-Oriented Concurrent Programming*, 199–220. The MIT Press, Cambridge, MA. 1987. {3.4.4}

AN88 M. C. Atkins, L. R. Nackman. The Active Deallocation of Objects in Object-Oriented Systems. *Software – Practice and Experience*, 18:11, 1073–1089. Nov. 1988. {2.5.3, 3.4.2.4}

App85 Apple Computer, Inc. *Inside Macintosh.* Addison-Wesley, Reading, MA. 1985. {3.4.4, 3.4.7.4, 3.4.8}

App90a Apple Computer, Inc. *The TrueType Font Format Specification.* Version 1.0, APDA M0825LL/A. Apple Program Developers Association, Cupertino, CA. 1990. {3.4.7.3}
TrueType as an example for a device-independent font-rasterization system.

App90b Apple Computer, Inc. Speed your software development with MacApp. *Develop – The Apple Technical Journal*, 2, 155–171. Apr. 1990. {3.4.8}
Describes the application framework MacApp.

Atk89 M. C. Atkins. *Implementation Techniques for Object-Oriented Systems.* Ph.D. Thesis. University of York, UK. Jun. 1989. {2.5.3, 3.4.2.4}
Additionally to [AN88]: Contains discussion of weak pointers to be used instead of finalization being built into a system.

B*85 F. L. Bauer et al. The Wide Spectrum Language CIP-L. *Lecture Notes in Computer Science*, 183. Springer-Verlag, Berlin, D. 1985. {3.1.2}

Bac86 M. J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, NJ. 1986. {3.4.4}

Bar88 J. F. Bartlett. Compacting Garbage Collection with Ambigous Roots. *Report*, 88/2. DEC Western Research Laboratory, MA. Feb. 1988. {3.4.2.3}
Conservative garbage collection.

BCF*92 M. Brandis, R. Crelier, M. Franz, J. Templ. The Oberon System Family. (Submitted for Publication). Institute for Computer Systems, ETH Zurich, CH. 1992. {3.4.7.3, 3.6, 3.6.2}

Ben88 J. Bentley. *More Programming Pearls – Confessions of a Coder.* Addison-Wesley, Reading, MA. 1988. {3.1.2}
Contains "Column 9: Little Languages".

Bla91 A. Black. Types and Polymorphism in Emerald. In: J. Palsberg, M.I. Schwartzbach [eds.]. Types, Inheritance and Assignments – A Collection of Position Papers from the ECOOP'91 Workshop W5, Geneva, Switzerland, July 1991. Appeared as *Technical Report* DAIMI PB–357. Computer Science Department, Aarhus University, DK. Jun. 1991. {1.1.1, 4.4.3}

BM72 R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:3, 173–189. 1972. {3.4.5.4}
Introduces B-Trees.

Bro75 F. P. Brooks. *The Mythical Man-Month – Essays on Software Engineering.* Addison-Wesley, Reading, MA. 1975. {3.7.3}
Hints at the importance of code scaffolding.

BW88 H.-J. Boehm, M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18:9, 807–820. Sep. 1988. {3.4.2.3}
Conservative garbage collection.

Cam92 R. H. Campbell. Private Communication. Mar. 1992. {2.2}
Campbell admitted that the Choices project was hindered by the missing run-time type and garbage collection support in C++; changing C++ was not an option for the Choices project.

Car89 L. Cardelli. Typeful Programming. *Research Report*, 45. DEC Systems Research Center, Palo Alto, CA. May 1989. {1.1.1, 2.4.1, 2.5.1, 4.4.3}
Makes strong point for static typing and type checking ("typeful programming"), in favour of separate compilation; introduces language Quest having a second-level type system.

CCH*89 P. S. Canning, W. R. Cook, W. L. Hill, J. Mitchell, W. G. Olthoff. F-bounded quantification for object-oriented programming. *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, 273–280. 1989. {1.1.1}

CHC90 W. R. Cook, W. L. Hill, P. S. Canning. Inheritance is not subtyping. *Proceedings of the ACM Conference on Principles of Programming Languages (POPL'90)*, 125–135. ACM Press. Addison-Wesley, Reading, MA. Jan. 1990. {1.1.1}

CHO92 K. Chen, P. Hudak, M. Odersky. Parametric Type Classes. *Proceedings of the ACM Conference on LISP and Functional Programming*. Jun. 1992. {1.1.1}

Cha92 C. Chambers. Object-Oriented Multi-Methods in Cecil. *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June, 1992. Lecture Notes in Computer Science, 615, 33-56. Springer-Verlag, Berlin, D. Jun. 1992. {1.1.1, 1.1.2}

Che84 D. R. Cheriton. The V Kernel, a software base for distributed systems. *IEEE Software*, 1:2, 19–42. Apr. 1984. {2.2}
Example for an early "micro kernel"-like architecture masking aspects of distribution; extensions by adding new service modules and performing inter-process communications (IPCs).

Che86 D. R. Cheriton. VMTP: A Transport Protocol for the Next Generation of Communication Systems. *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'86)*. Aug. 1986. {2.2}
Describes the VMPT transport protocol used in the V kernel to perform remote inter-process communications (IPCs); interesting part of the V system [Che84].

CJM*89 R. H. Campbell, G. M. Johnston, P. W. Madany, V. F. Russo. Principles of object-oriented operating system design. *Technical Report*, R-89-1510. Department of Computer Science, University of Illinois at Urbana Champaign, IL. Apr. 1989. {2.2, 3.4.3.4}
Choices OS; mimics fairly conventional OS's but uses frameworks inside to gain structure and flexibility.

CL90 P. R. Calder, M. A. Linton. Glyphs: Flyweight Objects for User Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'90)*. Oct. 1990. {3.4.6}
Introduces concept of objects being lightweight enough to use one for every character but flexible enough to implement complex objects.

Coh81 J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13:3, 341–367. Sep. 1981. {2.5.3, 3.4.2.3}
Survey of garbage collection techniques.

Col60 G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3:12, 655–657. Dec. 1960. {2.5.3}
Proposes reference counters to detect garbage.

Coo89 W. R. Cook. A Proposal for Making Eiffel Type Safe. *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP'89)*, Nottingham, England, 57–70. Cambridge University Press, UK. Jul. 1989. {1.1.2, 4.4.3}
Isolates the contravariance violation of the *like current* construct found in Eiffel and proposes a solution.

Cox87 B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA. 1987. {2.2}

CRJ87 R. H. Campbell, V. F. Russo, G. Johnston. Choices: The Design of a Multiprocessor Operating System. *Proceedings of the USENIX C++ Workshop*, 109–123. Santa Fe, NM. Nov. 1987. {2.2, 4.5}

Cre91 R. Crelier. OP2: A Portable Oberon-2 Compiler. *Proceedings of the Second International Modula-2 Conference*. Loughborough University, UK. Sep. 1991. {3.4.3.4, 3.7.3, 4.2, 4.4.4}
The compiler used to develop Ethos.

CS91 R. Carr, D. Shafer. *The Power of PenPoint.* Addison-Wesley, Reading, MA. 1991. {2.2}

CU91 C. Chambers, D. Ungar. Making Pure Object-Oriented Languages Practical. *Proceedings of the Sixth Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'91),* Phoenix, AZ. SIGPLAN Notices 26:11, 1-15. ACM Press. Addison-Wesley, Reading, MA. Oct. 1991. {1.1.3, 1.1.4, 4.2}
Demonstrates that a pure object-oriented language (Self) can be implemented in a way reaching about 50% of the performance of optimized C code.

Dec92 *Preliminary Alpha Architecture Handbook.* (Special Announcement Edition). Digital Equipment Corporation, Maynard, MA. Feb. 1992. {4.4.1}

DG87 L. G. DeMichiel, R. P. Gabriel. The Common Lisp Object System: An Overview. *Proceedings of the First European Conference on Object-Oriented Programming (ECOOP'87),* Paris, F. Lecture Notes in Computer Science, 276. Springer-Verlag, Berlin, D. Jun. 1987. {1.1.1}
CLOS introduction.

Dij65 E. W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technological University, Eindhoven, NL. 1965. In: F. Genuys [ed.]. *Programming Languages,* 43–112. Academic Press, London, UK. 1968. {3.4.4, 3.5.3, 3.7.3}
Introduction of semaphores.

Dij68 E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM,* 11:5, 341–346. May 1968. {1.2, 1.2.1, 2.3.2, 3.3.1}
The THE operating system was a forerunner of systems designed using a strict layering of abstractions.

Dij76 E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ. 1976. {3.4, 3.7.3}
Introduces concept of guarded commands (used in *if* and *do* constructs).

DLA88 P. Dasgupta, R. J. LeBlanc Jr., W. F. Appelbe. The Clouds distributed operating system: Functional description, implementation details, and related work. *Proceedings of the Eighth International Conference on Distributed Computer Systems (ICDCS'88),* San José, CA. IEEE, New York, NY. Jun. 1988. {2.2}
Clouds OS; concentrates on integrating support for reliable objects into low levels of the OS.

DLM*78 E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21:11, 966–975. Nov. 1978. {3.4.2.3}
Early example for an asynchronous garbage collector; problem is its inefficiency.

DMN68 O.-J. Dahl, B. Myrhaug, K. Nygaard. *SIMULA 67 Common Base.* Norwegian Computing Center, Oslo, N. 1968. {1.1.1, 2.4.2, 3.7.3}

DN66 O.-J. Dahl, K. Nygaard. Simula – An ALGOL–Based Simulation Language. *Communications of the ACM*, 9:9. Sep. 1966. {3.7.3}

DoD80 *Ada Reference Manual: Proposed Standard Document.* Department of Defence, USA. Jul. 1980. {1.1.1, 3.7.3}

Dod92 M. Dodani, C.-S. Tsai. ACTS: A Type System for Object-Oriented Programming Based on Abstract abd Concrete Classes. *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June, 1992. Lecture Notes in Computer Science, 615, 309-328. Springer-Verlag, Berlin, D. Jun. 1992. {2.4.3}

Don90 C. Dony. Exception Handling and Object-Oriented Programming: Towards a Synthesis. *Proceedings of the Joint Conference of the European Conference on Object-Oriented Programming, and the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (ECOOP/OOPSLA'90)*, Ottawa, Canada. SIGPLAN Notices, 25:10. ACM Press. Addison-Wesley, Reading, MA. Oct. 1990. {2.5.4}

DS84 L. P. Deutsch, A. Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the Eleventh Symposium on the Principles of Programming Languages (POPL'84)*, Salt Lake City, UT. ACM Press. Addison-Wesley, Reading, MA. 1984. {1.1.2, 1.1.3, 4.2}

DS90 E. W. Dijkstra, C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY. 1990. {3.7.3}
Most recent coverage of program semantics based on predicate transformers.

ES90 M. A. Ellis, B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA. 1990. {1.1.1, 2.5.4}
Contains the latest language revisions over the original language defined in [Str86].

FH88 A. J. Field, P. G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, Reading, MA. 1988. {2.3.2}

Gie90 M. Gien. Micro-Kernel Architecture – Key to Modern Operating Systems Design. *UNIX Review*, 8:11. Nov. 1990. {1.2.2, 4.5}

GJS91 D. K. Gifford, P. Jouvelot, M. A. Sheldon, J. W. O'Toole, Jr.. Semantic File Systems. 16-25. {3.5.2}

Gol83 A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA. 1983. {2.2, 4.5}
Describes the Smalltalk programming environment.

Gol91 B. Goldberg. Tag–Free Garbage Collection for Strongly Typed Programming Languages. *Proceedings of the Conference on Programming Language Design and Implementation (SIGPLAN'91)*, Toronto, Ontario, CDN. SIGPLAN Notices, 26:6, 165–176. {4.4.4}

GR83 A. Goldberg, D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA. 1983. {1.1.1, 2.2, 2.5.1}
Describes the large Smalltalk class library.

Gri91b L. Griffiths. Modula-2 is Three Times less Error Prone than C. *Proceedings of the Second International Modula-2 Conference*, 332-338. Loughborough University, UK. Sep. 1991. {3.4.2.2, 3.7.3}
Quantitative analysis of error classes as they occur when programming in C or in Modula-2, respectively.

Gri91 R. Griesemer. Oberon Lisp. Internal Memo. Institute for Computer Systems, ETH Zurich, CH. Jun. 1991. {3.4.2.3, 3.4.8.4}
Example for an application successfully ported from Oberon to Ethos.

Gut85 J. Gutknecht. Concepts of the Text Editor Lara. *Communications of the ACM*, 28:9, 942-960. Sep. 1985. {3.4.6}
Introduces piece lists as a data structure to carry both, attribute runs and file runs.

GWB91 R. P. Gabriel, J. L. White, D. G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34:9, 27–38. Sep. 1991. {2.5.1}
Contains a useful definition of reflection (pp. 36–37).

HAR*88 F. Herrmann, F. Armand, M. Rozier, M. Gien, V. Abrossimov, I. Boule, M. Guillemont, P. Leonard, S. Langlois, W. Neuhauser. Chorus, a new technology for building UNIX systems. *Proceedings of the European Unix User Group Autumn Conference (EUUG Autumn)*, Cascais, Portugal. Oct. 1988. {2.2}
Describes the Chorus OS.

Har91 W. Harris. Contravariance for the rest of us. *Journal of Object Oriented Programming*, 4:3, 10-18. Nov. 1991. {1.1.1}

Explains the contravariance phenomenon. Makes a point for "inheritance is not subclassing". Looks at multi-methods to cure contravariance problems. Two ways of creating safe polymorphic code: (a) separating subtyping and subclassing, (b) parametric polymorphism. (b) is further split into simple (unquantified) parametric, bounded, and f-bounded quantification.

HCU91 U. Hölzle, C. Chambers, D. Ungar. Optimizing Dynamically–Typed Object–Oriented Languages with Polymorphic Inline Caches. *Proceedings of the Fifth European Conference on Object-Oriented Programming (ECOOP'91)*, Geneva, Switzerland. Lecture Notes in Computer Science, 512, 268-287. Springer-Verlag, Berlin, D. Jun. 1992. {1.1.2, 4.2}

Describes how to do on the fly compilation at run-time utilizing observed polymorphisms.

Hee88 B. Heeb. Private Communication. 1988. {3.4.3.1}

Beat implemented an early experimental Oberon loader for the Macintosh. He hinted at the problems of recursive loaders and suggested a non-recursive variant.

Hoa74 C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17:10, 549–557. (Erratum in: Communications of the ACM, 18:2, 95. February 1975.). Oct. 1974. {3.4.4}

Systematic coverage of monitors.

Hoa84 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ. 1984. {3.4.4}

HP92 B. Heeb, C. Pfister. Chameleon – A Workstation of a Different Colour. *(Submitted for Publication.)*. Institute for Computer Systems, ETH Zurich, CH. 1992. {3.5.3, 3.6.2}

Hut87 N. Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming*. Ph.D. Thesis. Department of Computer Science and Engineering, University of Washington, Seattle, WA. Jan. 1987. {1.1.1, 4.4.3}

Hyp87 *hyperstone 32-Bit-Microprocessor User's Manual*. hyperstone electronics GmbH, Konstanz, D. 1987. (Issue April 1990). {4.4.1}

Ing81 D. H. H. Ingalls. Design Principles Behind Smalltalk. *BYTE*, 6:8, 286-298. Aug. 1981. {1.2.2}

JF88 R. E. Johnson, B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1:2. Jun. 1988. {2.2, 3.5}
Defines notion of frameworks; makes claim that concrete classes should not be extended.

JZ91 R. E. Johnson, J. M. Zweig. Delegation in C++. *Journal of Object Oriented Programming*, 4:3, 31-34. Nov. 1991. {1.1.1, 3.5, 4.4}
Shows a simple (but not really natural) way of simulating delegation using C++. Points out the subtle difference between forwarding and delegation. Concludes in the question whether future languages will be class or delegation base. (Does not ask the question why one does not add both to a single language ...)

KL89 W. Kim, F. H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications*. ACM Press. Addison-Wesley, Reading, MA. 1989. {3.4.4}
Coverage of various OO concurrency approaches.

KLC*91 E. J. Koldinger, H. M. Levy, J. S. Chase, S. J. Eggers. The Protection Lookaside Buffer. *Technical Report*, 91–11–05. Department of Computer Science and Engineering, University of Washington, Seattle, WA. Nov. 1991. {4.4.2}

KMN*90 S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3:3, 11–22. Sep. 1990. {2.2}
Language *Guide* used for the single-language operating system *Commandos*.

Knu68 D. E. Knuth. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA. 1968. Second edition: 1973. {2.5.3}

Knu84 D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA. 1984. {3.4.6}

Knu86 D. E. Knuth. *The METAFONTbook*. Addison-Wesley, Reading, MA. 1986. {2.5.3, 3.4.7.3}
Describes meta-fonts as a device-independent description of fonts.

KP88 G. E. Krasner, S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1:3, 26-49. Aug. 1988. {2.3.2}
Good explanation of the MVC design principle.

KR78 B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ. 1978. {3.7.3}

KRB*91 G. Kiczales, J. des Rivieres, D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA. 1991. {2.5.1}

Lac91 S. Lacourte. Exceptions in Guide, an Object-Oriented Language for Distributed Applications. *Proceedings of the Fifth European Conference on Object-Oriented Programming (ECOOP'91)*, Geneva, Switzerland. Lecture Notes in Computer Science, 512, 268-287. Springer-Verlag, Berlin, D. Jun. 1992. {2.5.4}
Contains a survey of approaches to exception handling in typed object-oriented languages.

Lam83 B. A. Lampson. A Description of the Cedar Language – A Cedar Language Reference Manual. *Technical Report*, CSL-83-15. Xerox Palo Alto Research Center, Palo Alto, CA. 1983. {2.2}

Mad91 P. W. Madany. *An Object-Oriented Framework for File Systems.* Ph.D. Thesis (Draft). Department of Computer Science, University of Illinois at Urbana-Champaign, IL. Nov. 1991. {2.2}
Explains Choices file system framework in detail (cf. [MCR*89]).

Mag91 B. Magnusson. Code Reuse Considered Harmful (Guest Editorial). *Journal of Object Oriented Programming*, 4:3, 8. Nov. 1991. {1.1.1, 2.4.4, 3.5}
Makes the claim that using inheritance for code re-use is as bad as using procedures for code re-use in classical programs. Just as a procedure should represent an algorithmic abstraction, a class should represent an abstraction in the modeling space.

Mar91 The GADGETS User Interface Management System. *Structured Programming*, 12:12, 75-89. Dec. 1991. {3.5.5}

MCR*89 P. W. Madany, R. H. Campbell, V. F. Russo, D. E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP'89)*, Nottingham, England, 311–328. Cambridge University Press, UK. Jul. 1989. {2.2}
Introduces Choices file system framework (cf. [Mad91]).

McC60 J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3, 184–195. 1960. {3.4.2.3}

Mey88 B. Meyer. *Object–Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ. 1988. {1.1.3, 2.5.2, 3.7.3, 4.4.3}
Introduces language *Eiffel* to explain aspects of object-oriented software construction.

Mey92 B. Meyer. *Eiffel – The Language*. Prentice Hall, Englewood Cliffs, NJ. 1992. {1.1.1, 1.1.2, 2.4.2, 2.5.2, 4.4.3}
Reference to most recent version of language *Eiffel*; introduces "System-Level Type Checking" to fix safety problems with the type system (cf. [Coo89]).

MG89 J. A. Marques, P. Guedes. Extending the Operating System to Support an Object-Oriented Environment. *Proceedings of the Fourth ACM Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'89)*, New Orleans, LO. SIGPLAN Notices, 24:10. Oct. 1989. {2.2}
Introduces operating system *Commandos*: Concentrates on mapping of objects into domains (address spaces) spanning over multiple machines; language oriented: Guide [KMN*90].

MMS79 J. G. Mitchell, W. Maybury, R. Sweet. Mesa Language Manual, Version 5.0. *Technical Report*, CSL–79–3. Xerox Palo Alto Research Center, Palo Alto, CA. Apr. 1979. {2.2, 3.4.8.1}
Mesa is the language used throughout the *Cedar* system [Tei84].

MN91 M. S. Manasse, G. Nelson. Trestle Reference Manual. *Research Report*, 68. DEC Systems Research Center, Palo Alto, CA. Dec. 1991. {2.3.3.2}
The Trestle *oracles* used to adapt to screen-dependent resources are examples for special directory objects.

MT86 S. J. Mullender, A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:4, 289–300. 1986. {2.2}

MTG89 H. Mössenböck, J. Templ, R. Griesemer. Object Oberon – An Object-Oriented Extension of Oberon. *Technical Report*, 109. Institute for Computer Systems, ETH Zurich, CH. Jun. 1989. (Revised April 1990). {3.1.2, 3.4.8.1}

MW91 H. Mössenböck, N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12:4. 1991. {2.2, 3.1.2, 4.1, 4.4.3}

Nau60 P. Naur. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 3:5. May 1960. {3.7.3}

Nel91 G. Nelson [ed.]. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ. 1991. {2.4.3, 2.5.4, 3.4.5.2, 3.7.3}

Ode89 M. Odersky. Extending Modula–2 for Object–Oriented Programming. Proceedings of the First International Modula-2 Conference, Bled, Yugoslavia. Oct. 1989. {2.4.2}

Ohr84 R. S. Ohran. *Lilith: A Workstation Computer for Modula-2.* Dissertation ETH Zurich, No. 7646. 1984. {4.4.1}

Omo91 S. M. Omohundro. The Sather Language. *Technical Report*, TR–91–34. International Computer Science Institute, Berkeley, CA. Jun. 1991. {2.4.2}

OPS92 N. Oxhoj, J. Palsberg, M. I. Schwartzbach. Making Type Inference Practical. *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June, 1992. Lecture Notes in Computer Science, 615, 329-349. Springer-Verlag, Berlin, D. Jun. 1992. {2.3.2}

Org72 The Multics System: An Examination of Its Structure. The MIT Press, Cambridge, MA. 1972. {3.4.4}

Par72 D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:12, 1053–1058. Dec. 1972. {2.3.2}

Pes89 F. V. Peschel. *Vamos – Entwurf und Realisierung eines erweiterbaren Betriebssystems für Arbeitsplatzrechner.* Dissertation ETH Zurich, No. 8713. Verlag der Fachvereine, Zurich, CH. 1989. {2.2}

PHT91 C. Pfister [ed.], B. Heeb, J. Templ. Oberon Technical Notes. *Technical Report*, 156. Institute for Computer Systems, ETH Zurich, CH. Mar. 1991. {3.4.2.2, 3.4.2.3, 3.4.2.6}

Pik91 R. Pike. $8\frac{1}{2}$, the Plan 9 Window System. *Proceedings of the USENIX Summer Conference*, Nashville, TE. Jun. 1991. {2.2}

Pou91 D. Pountain. ARM600: RISC Goes OOP. *BYTE*, 16:13, 84IS-49–60. Dec. 1991. {4.4.2}
The ARM600 uses a MMU that distinguishes between address and protection mappings, the former called pages, the latter domains.

PPT*90 R. Pike, D. Presotto, K. Thompson, H. Trickey. Plan 9 from Bell Labs. *Proceedings of the Summer UK Unix User Group Conference (UKUUG'90)*, London, GB, 1–9. Jul. 1990. {2.2}

PS83 J. L. Peterson, A. Silberschatz. Operating System Concepts. Addison-Wesley, Reading, MA. 1983. (Corrected Reprint of Second Edition: 1986). {1.2}

PS90 J. Palsberg, M. I. Schwartzbach. Object-Oriented Type Inference. *Proceedings of the Sixth Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'91)*, Phoenix, AZ. SIGPLAN Notices 26:11, 146–161. ACM Press. Addison-Wesley, Reading, MA. Oct. 1991. {1.1.2}

RAA*88 M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guiellemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Overview of the Chorus Distributed Operating System. *Computing Systems Journal*, 1:4, 305–370. The Usenix Association. Dec. 1988. {2.2}

RBF*89 R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, R. Sanzi. Mach: A Foundation for Open Systems. *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS–II)*, Pacific Grove, CA. Sep. 1989. {2.2}

Rov84 P. Rovner. On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language. *Technical Report*, CSL-84-7. Xerox Palo Alto Reaserach Center, Palo Alto, CA. Jul. 1984. {2.5.3, 3.4.2.4}

RST89 R. van Renesse, H. van Staveren, A. S. Tanenbaum. The Performance of the Amoeba Distributed Operating System. *Software – Practice and Experience*, 19:3, 223–234. Mar. 1989. {2.2}

RW92 M. Reiser, N. Wirth. *Programming in Oberon. Steps Beyond Pascal and Modula*. Addison-Wesley, Reading, MA. 1992. {2.2}

SG86 R. W. Scheifler, J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5:2. 1986. {3.4.7, 3.4.7.3}

Sha86 M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. *Proceedings of the Sixth International Conference on Distributed Computer Systems (ICDCS'86)*, Cambridge, MA. IEEE, New York, NY. May 1986. {4.5}

Sha89 M. Shapiro. Prototyping a Distributed Object-Oriented Operating System on UNIX. *Research Report*, 1082. Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, F. Aug. 1989. {2.2}
SOS operating system: Concentrates on fragmented objects – support of remote objects via local proxy objects.

Sha92 M. Shapiro. Object-Oriented and Operating Systems. *Bibliography file*. Maintained by the SOR project. Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, F. May 92 (latest version, periodically updated). {bibliography}
Fetchable by anonymous ftp from ftp.inria.fr, directory INRIA/bib, file ooos.bib.Z.

Spa86 E. H. Spafford. *Kernel Structures for a Distributed Operating System*. Ph.D. Thesis. Georgia Institute of Technology, Atlanta, GA. May 1986. {2.2}

SS91 R. Sharma, M. L. Soffa. Parallel Generational Garbage Collection. *Proceedings of the Sixth Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'91)*, Phoenix, AZ. SIGPLAN Notices 26:11, 16–32. ACM Press. Addison-Wesley, Reading, MA. Oct. 1991. {3.4.2.3}

Sta89 B. Stamm. Algorithms for Drawing Thick Lines and Curves on Raster Devices. *Technical Report*, 107. Institute for Computer Systems, ETH Zurich, CH. May 1989. {3.4.7}

Ste87 L. A. Stein. Delegation is Inheritance. *Proceedings of the Second ACM Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA'87)*, Orlando, FL. SIGPLAN Notices, 22:12. Oct. 1987. {1.1.1}

Str86 B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA. 1986.
The latest language revision may be found in [ES90].

SW67 H. Schorr, W. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10:8, 501–505. Aug. 1967. {3.4.2.3}

Szy90a C. A. Szyperski. Network Communication in the Oberon Environment. *Technical Report*, 126. Institute for Computer Systems, ETH Zurich, CH. Feb. 1990. {3.4.4, 3.5.3}

Szy90b C. A. Szyperski. Towards Object-Oriented Structures for Open Operating Systems. *First Workshop on Object-Orientation in Operating Systems*. In: Addendum to the Proceedings of the Joint Conference of the European Conference on Object-Oriented Programming and the Conference on Object-Oriented Programming Systems, Languages, and Applications (ECOOP/OOPSLA'90), Ottawa, Canada. ACM Press. Addison-Wesley, Reading, MA. Oct. 1990. {2.3.3.1}

Szy91 C. A. Szyperski. Write: An Extensible Text Editor for the Oberon System. *Technical Report*, 151. Institute for Computer Systems, ETH Zurich, CH. Jan. 1991. {3.4.5.2, 3.4.6, 3.5.4}

Szy92a C. A. Szyperski. Write-ing Applications: Designing an Extensible Text Editor as an Application Framework. *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92)*, Dortmund, D. Prentice Hall, Englewood Cliffs, NJ. Mar. 1992. {2.2, 3.4.6, 3.4.8.4, 3.5.4}
The *Write* text editor has been derived from the *Ethos* text system. This paper explains many of the ideas behind the extension model of both the *Write* editor and the Ethos text system.

Szy92b C. A. Szyperski. Import is not Inheritance. Why we need both: Modules and Classes. *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June, 1992. Lecture Notes in Computer Science, 615, 19-32. Springer-Verlag, Berlin, D. Jun. 1992. {1.1.3, 1.1.4, 2.2, 2.4.1, 2.4.2, 4.4.3}

Tan87 A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ. 1987. {1.2}

Tan92 A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ. 1992. {1.2}

Tei84 W. Teitelman. A Tour Through Cedar. *IEEE Software*, 1:2, 44–73. Apr. 1984. {1.1.2, 2.2, 3.4.8.1}
The *Cedar* system is an evolutionary predecessor of the *Oberon* system. Many of the design aspects of *Cedar* have been carried through to *Oberon*, from where still a considerable amount reached *Ethos*. *Cedar* is based on the language *Mesa* [MMS79].

Tem91 J. Templ. Design and Implementation of SPARC-Oberon. *Structured Programming*, 15:12, 197–205. Dec. 1991. {3.4.2.3, 4.4.4}
Contains the description of a technique for conservative garbage collection to handle references in untaged stack frames.

Tes85 L. Tesler. Object–Pascal Report. *Structured Programming*, (formerly: Structured Language World), 9:3, 10–17. Mar. 1985. {1.1.1}

TMR86 A. S. Tanenbaum, S. J. Mullender, R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. *Proceedings of the Sixth International Conference on Distributed Computer Systems (ICDCS'86)*, Cambridge, MA, 558–563. IEEE, New York, NY. May 1986. {2.2, 4.5}

UJ88 D. Ungar, F. Jackson. Tenuring Policies for Generation-Ba: Storage Reclamation. *Proceedings of the Third ACM Conference Object-Oriented Programming Systems, Languages, and Applicati (OOPSLA'88)*, San Diego, CA. SIGPLAN Notices, 23:11, 1–17. N 1988. {3.4.2.5}

Ung84 D. Ungar. Generation Scavenging: A Non-disrupt High-performance Storage Reclamation Algorithm. *Proceedings of ACM Symposium on Practical Software Development Environme* Pittsburg, PA. SIGPLAN Notices, 19:5, 157–167. Apr. 1984. {2. 3.4.2.5}

US87 D. Ungar, R. B. Smith. Self: The Power of Simplicity. *Proceeding the Second ACM Conference on Object-Oriented Programming, Syste and Applications (OOPSLA'87)*, Orlando, FL. SIGPLAN Notices, 22: Oct. 1987. {1.1.1}

Vet91 C. Vetterli. *OPUS: Entwurf und Realisierung eines erweiterba objektorientierten Dokumentenverarbeitungssystems.* Dissertation E Zurich, No. 9456. Verlag der Fachvereine, Zurich, CH. 1991. {3.4.

WDH89 M. Weiser, A. Demers, C. Hauser. The Portable Common Runti Approach to Interoperability. *Proceedings of the Twelfth A Symposium on Operating Systems Principles (SOSP'89)*, Litchfield P. AZ. SIGOPS Operating System Reviews 23:5, 114–122. Dec. 19 {2.5.3, 3.4.2.4}

Web89 B. F. Webster. *The NeXT Book.* Addison-Wesley, Reading, MA. 19 {2.2, 3.4.7}
Application Framework *NextStep* using *Display-Postscript.*

Weg90 P. Wegner. Concepts and Paradigms of Object-Orien Programming. *OOPS Messenger*, 1:1, 7–87. Aug. 1990. {1.1.1}

WG88 N. Wirth, J. Gutknecht. The Oberon System. *Technical Report*, Institute for Computer Systems, ETH Zurich, CH. Jul. 1988. {4.4.4

WG89 N. Wirth, J. Gutknecht. The Oberon System. *Software – Practice Experience*, 19:9. Sep. 1989. {1.1.1, 2.2, 3.1.1, 3.1.2, 3.3.1, 3.4.2.2}

WG92 N. Wirth, J. Gutknecht. *Project Oberon. The Design of an Opera System and Compiler.* Addison-Wesley, Reading, MA. 1992. { 3.1.2, 3.4.2.2, 3.4.2.3, 3.4.3, 3.4.5.2, 4.2}

WH81 P. H. Winston, B. K. P. Horn. *Lisp.* Addison-Wesley, Reading, M 1981. (Corrected Reprint of Third Edition: 1989). {3.4.2.3}

Wil83 G. Williams. The Lisa Computer System. *BYTE*, 8:2. Feb. 1983. {3.4.6}
Introduces the *Lisa* computer, including the *LisaWrite* editor with its characteristic rulers.

Wil89 M. Wille. *Overview: Entwurf und Realisierung eines Fenstersystems für Arbeitsplatzrechner.* Dissertation ETH Zurich, No. 8771. Verlag der Fachvereine, Zurich, CH. 1989. {2.2, 2.4.3, 3.4.8.1}

Wil92 P. R. Wilson. Uniprocessor Garbage Collection Techniques. *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, F. Lecture Notes in Computer Science. Springer-Verlag, Berlin, D. Sep. 1992. {2.5.3}
Contains a thorough comparison of the most recent garbage collection techniques for uniprocessors, including mark-and-scan, copying, and treatmill collectors.

Wir63 N. E. Wirth. *A Generalization of Algol.* Ph.D. Thesis. University of California at Berkeley, CA. 1963. {3.7.3}
Introduces the language *Euler.*

Wir71 N. Wirth. The Programming Language PASCAL. *Acta Informatica*, 1, 35–63. 1971. {3.7.3}

Wir77 N. Wirth. Modula: A Programming Language for Modular Multiprogramming. *Software – Practice and Experience*, 7:1, 37–52. Jan. 1977. {3.5.3, 3.7.3}
Multi-programming based on conditions and monitors.

Wir82 N. Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, D. 1982. (Fourth edition: 1988). {1.1.1, 2.4.2, 3.2.1, 3.7.3}

Wir84 N. Wirth. Schemes for Multiprogramming and their Implementation in Modula-2. *Technical Report*, 59. Institute for Computer Systems, ETH Zurich, CH. Jun. 1984. {3.4.4, 3.5.3}
Coroutines and their use to implement higher multi-programming schemes.

Wir88a N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10:2, 204–214. Apr. 1988. {2.3.2, 3.1.2, 3.2.1, 3.4.2.1}

Wir88b N. Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18:7, 671-690. Jul. 1988. {1.1.1, 2.2, 3.7.3}

Wir89 N. Wirth. Designing a System from Scratch. *Structured Programming*, 13:1, 10–18. Jan. 1989.

WMP*69 A. van Wijngaarden [ed.], B. J. Mailloux, J. E. L. Peck, C. H. A. Koster. Report on the Algorithmic Language ALGOL 68. *Numerische Mathematik*, 14, 79–218. 1969. {3.7.3}

WS92 J. Wilkes, B. Sears. A Comparison of Protection Lookaside Buffers and the PA-RISC Protection Architecture. *Technical Report*, HPL–92–55. Hewlett-Packard Laboratories, Palo Alto, CA. Mar. 1992. {4.4.2}

Lebenslauf:

Clemens Alden Szyperski

19.10.1962	Geboren in Philadelphia, PA, U.S.A.; Doppelstaatsbürgerschaft Deutsch und U.S.–Amerikanisch Eltern: Norbert und Edith Szyperski
1968–72	Katholische Grundschule der Gemeinde Rösrath, Rösrath, Deutschland
1972–81	Freiherr-vom-Stein Gymnasium, Rösrath, Deutschland
1981	Allgemeine Hochschulreife (Abitur)
1981–87	Elektrotechnik, Spezialisierung Technische Informatik Rheinisch–Westfälische Technische Hochschule, Aachen, Deutschland
1987	Diplom (Dipl.–Ing.)
1987–92	Assistent, Institut für Computersysteme Eidgenössische Technische Hochschule Zürich, Schweiz