Design and Implementation of a Component Architecture for Oberon

Diss. ETH No 11697

Design and Implementation of a Component Architecture for Oberon

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Johannes Leon Marais
M.Sc Computer Science, RAU
born July 21, 1967
citizen of Cape Town, South Africa

accepted on the recommendation of
Prof. Dr. J. Gutknecht, examiner
Prof. Dr. N. Wirth, co-examiner

1996

For my dear parents

Contents

List of Figures

Acknowledgments

Oberon System 3 with Gadgets is the product of so many people that a complete list of contributors is difficult to compile. Since the design and implementation of the original Oberon system by N. Wirth and J. Gutknecht, numerous ETH assistants, students, researchers and fans have contributed in extending, perfecting and propagating Oberon. Nevertheless, I would like to acknowledge the following people who made important contributions to the subject matter of this thesis:

- Jürg Gutknecht, the visionary and architect of Oberon System 3, for countless discussions about the design and implementation of Oberon System 3 and Gadgets. Invariably these design discussions lead to elegant solutions to difficult problems. Gadgets would not be the success it is today without his extensive support.

- The early System 3 team, Karl Rege and Ralph Sommerer. Karl introduced me to the intricacies of Oberon and provided valuable feedback as an alpha tester, and Ralph contributed to the text sub-system and also provided feedback as one of the early users.

- Emil Zeller, who built the most gadgets-based applications. Emil's software makes the most advanced use of the Gadgets document model, and includes the Web browser (Figures 3.12 and 3.13), mail client, news reader, FTP tool, tutorial system (Figure 3.11), and archiver. Without his contribution the system would not be as useful as it is today.

- Josef Templ for his contribution in porting Oberon to UNIX, locating a few design and portability problems, and also many bugs. His criticisms were an important and welcome feedback.

- André Fischer for his effort in documenting the system in electronic tutorials, and so discovering several inconsistencies in the process. Few would be able to use the system without this substantial contribution.

- Thomas Kistler, who as porter of Oberon System 3 to the Mac, fixed many problems related to portability and also made important improvements. Thomas implemented the Oberon systems based on OMI, which is in turn based on the thesis work of Michael Franz and Régis Crelier.

- All the students that completed projects under my supervision. Many of the projects contributed to the design and implementation of Gadgets. Noteworthy contributions that are now part of the Oberon distribution include Watson from Patrick Saladin and Rembrandt (Figure 3.7) from Daniel Ponti.

- Daniel Bleichenbacher for his advice on cryptographical algorithms and their performance.

- The people that contributed in one way or another to my Spirit of Oberon distribution for Windows. Important contributions came from Matthias Hausner for building the original Windows interfaces on which the port is based, Niklaus Mannhart for the OP2 compiler port, Emil Zeller for porting the network sub-system, and Patrick Saladin and Markus Dätwyler for the required set of computer games. I would like to thank Josef Templ for the media propaganda and his publication of the *Oberon System 3 and Gadgets* CD-ROM in cooperation with Addison-Wesley. I would also like to thank all the users who sent me a postcard.

- The porters of Oberon System 3 and Gadgets to other architectures: Pieter Muller for Native Oberon, Andreas Disteli for DOS-Oberon, Andreas Würtz for MacOberon, Markus Dätwyler and Marc Sperisen for Linux Oberon, and Michael Franz and Thomas Kistler for their OMI implementation for MacOS. Invariably a new port was also an opportunity for improving a part of the system.

- Erich Oswald, for putting up with the many visitors to our office, and also for interesting discussions about computer graphics and life in general.

- My colleagues at the Institute für Computersysteme for an interesting work environment and many stimulating discussions over lunch.

- Dominique for proof-reading the manuscript and screening me from the real world while I was writing my thesis. I would also like to thank

Robert Griesemer for helping with the German translation of the abstract, and Cynthia Hibbard for helping me improve my writing.

Abstract

The traditional view of computer software is that of off-the-shelf programs with little or no possibility of customization. Engineers create this type of software by combining software components in such a way that the delivered product does not show how it was constructed. Unfortunately, this construction technique does not allow the buyer to adapt components, add new components to or remove components from the program. So, buyers unhappy with the functionality of a product have little choice but to wait until the vendor releases an enhanced version of the program. Consequently, software vendors slowly incorporate new features in successive versions of a program, features that are not necessarily useful to all users. The result is that programs tend to get bigger with each new version, with no significant improvement in functionality.

The obvious solution to this problem is to give the knowledgeable buyer the flexibility of composing custom software from prefabricated parts. This approach has the advantage that only those parts that are actually required need to be bought, and with a variety of parts readily available, the buyer has much more freedom in constructing software. The most end-user accessible software that fall in this category of configurable systems are *document-based* user interfaces. A document-based user interface regards applications and their user interfaces as components that can be composed and configured interactively as if editing a document. The distinction between a document-based user interface and a GUI builder is that a document-based user interface allows the users to modify applications even after they are installed. In the simplest example, this would even allow a user to modify a dialog box in an installed application.

The principal technical difficulty in building a document-based interface is to provide the communication infra-structure that allow independently developed components to be combined interactively. Without a mutual communication protocol, components from different vendors cannot "plug" into each other. Without a dynamic connection mechanism, components would not be able to be added to

or removed from a system while it is running.

This thesis describes the design and implementation of a component architecture and document-based user interface for the Oberon system. The *Gadgets system* uses first-class components called *gadgets* to build documents and connect the documents with functionality. Gadgets ranging from typical user interface elements to high-level building blocks are composed interactively in active documents. An important feature is that independently developed components can be integrated with existing components without special effort. This gives the *composer* the ability to compose applications from a large palette of reusable components, and thus speeds up the development process.

This thesis addresses several issues, including how components communicate, inter-component communication protocols, how components are made persistent, how components can be combined into more complicated components, how custom behavior can be attached to component instances, and how documents constructed in this way are consistently transported. The thesis also discusses issues related to system structuring, dynamic extensibility, component authentication and system security.

Zusammenfassung

Traditionellerweise wird Computer-Software mit dem Begriff Programm-Paket (“off-the-shelf” program) verbunden, wobei typischerweise wenig oder keine Möglichkeit zur kundenspezifischen Anpassung der Software besteht. Solche Programm-Pakete sind in einer Weise konstruiert, die es nicht offensichtlich macht, aus welchen Komponenten sie zusammengesetzt wurden. Unglücklicherweise erlaubt diese Konstruktionsweise dem Käufer nicht, einzelne Komponenten anzupassen, neue Komponenten hinzuzufügen oder Komponenten zu entfernen. Ist ein Kunde unglücklich mit der Funktionalität eines solchen Produktes, so bleibt ihm oder ihr nicht viel anderes übrig als auf eine verbesserte Version des Herstellers zu warten. Konsequenterweise fügen Software-Hersteller langsam immer mehr Funktionen in neuere Versionen eines Programms hinzu, Funktionen die nicht notwendigerweise nützlich für alle Benutzer sind. In der Folge tendieren Programme dazu, mit jeder Version grösser und grösser zu werden, ohne dass dabei die Funktionalität signifikant erhöht wird.

Gibt man dem informierten Kunden die Flexibilität, ein System aus vorfabrizierten Teilen zusammenzusetzen, erhält man eine offensichtliche Lösung zu diesem Problem. Dieser Ansatz hat zusätzlich den Vorteil, dass man nur diejenigen Teile zu kaufen braucht, die man wirklich benötigt. Auch hat der Kunde mehr Freiheit, Software aus einer Vielzahl vorfabrizierter Komponenten zusammenzustellen. Aus der Kategorie solcher konfigurierbarer Systeme sind dokumentenbasierte Benutzeroberflächen am ehesten zugänglich für den Endbenutzer. Eine dokumentenbasierte Benutzeröberfläche betrachtet Applikationen und ihre Benutzer-Schnittstelle als individuelle Komponenten die wie Dokumente frei zusammengesetzt und konfiguriert werden können. Die Möglichkeit die Benutzeroberfläche sogar nach der Installation zu verändern unterscheidet eine dokumenten-basierte Benutzeröberfläche von einem “GUI builder”. Im einfachsten Fall erlaubt so eine Benutzeroberfläche sogar das Verändern einer Dialog-Box in einer installierten Anwendung.

Die Kommunikationsinfrastruktur die es ermöglicht, dass unabhängig voneinander entwickelte Komponenten frei kombiniert werden können, stellt die prinzipielle Schwierigkeit in der Konstruktion einer dokumentenbasierten Benutzeroberfläche dar. Komponenten verschiedener Hersteller können nicht "zusammengesteckt" werden ohne ein beidseitig eingehaltenes Kommunikations-Protokoll. Komponenten können auch nicht zur Laufzeit hinzugefügt oder entfernt werden ohne einen dynamischen Verbindungsmechanismus.

Diese Dissertation beschreibt das Design und die Implementierung einer Komponenten-Architektur und einer dokumentenbasierten Benutzeroberfläche für das Oberon-System. Das Gadget-System benutzt sogenannte "Gadgets"-Komponenten um Dokumente zu bilden und sie mit Funktionalität zu versehen. Gadgets reichen von einfachen Benutzeröberflächen-Elementen bis hin zu komplizierten Bausteinen und können interaktiv in aktiven Dokumenten zusammengesetzt werden. Eine wichtige Eigenschaft ist, dass unabhängig voneinander entwickelte Bausteine mit existierenden Komponenten ohne grossen Aufwand zusammengesetzt werden können. Dies ermöglicht dem Konstrukteur, schnell Applikationen aus einer Vielzahl wiederverwendbarer Komponenten zusammenzusetzen und dadurch den Entwicklungsprozess zu verkürzen.

Diese Arbeit beschreibt unter anderem wie Komponenten kommunizieren, wie ihre Kommunikationsprotokolle aussehen können, wie man Komponenten persistent macht, wie sie zu komplizierteren Komponenten zusammengesetzt werden können, wie man ihr Verhalten kundenspezifisch anpassen kann und wie man in solcherweise konstruierte Dokumente konsistent transportieren kann. Daneben werden auch Probleme im Zusammenhang mit System-Strukturierung, dynamischer Erweiterbarkeit, Komponenten-Authentizierung und System-Sicherheit diskutiert.

Chapter 1

Introduction

1.1 Motivation

Computer programming becomes an increasingly larger problem as computers get faster and data storage capacity increases. This is because our expectations from software grow as fast as hardware improves. The unprecedented improvement in semi-conductor technology [HH96] and our efforts to exploit these advances, motivates software getting more integrated, more complicated and larger. It is thus not surprising that with each generation, software takes longer to create, becomes more difficult to understand, more tricky to modify and more and more unreliable. This state of affairs is called the *software crisis* [NRB69, Gib94].

The root of the software crisis, the inherent complexity of software systems, cannot be eliminated. Instead, the only weapon of the software engineer is re-distributing complexity by dividing complicated things into parts—*divide et impera*—so as to regard the system on different levels of abstraction. Not only does gathering complexity in independent parts make a system easier to understand, but it is also a convenient way to divide labor between people. More time can be spent on developing a single part when the expensive labor is amortized by the product being used many times in different systems. It is generally believed that the wide-spread reuse of software parts can increase the reliability of software and decrease its time to market. This seems to be a desirable goal when we observe what the industrialization of physical goods manufacturing has achieved.

The idea of mass-produced software parts or components [McI76] was born in 1968 at the NATO conferences on software engineering. Since then we saw the development of "complexity repackaging" techniques like procedure libraries, module libraries, object-oriented class libraries, and frameworks (cf. 2.1). With so much time and money invested in priming the software industrial revolution, it is surprising that the discipline of software engineering is in such a bad state. A visionary of software components believes that the reason is that there is little incentive for manufacturing components when clients are not billed on a “pay-per-use” basis [Cox90]. Another reason might be the long time it takes for new technology to be adopted in commerce.

What has crystallized only recently though is that earlier software technologies concentrated on *building* components instead of addressing the more important problem of *composing* independently developed components. Composing requires a communication infra-structure for components to co-operate. As all communication is based on a common language and mutual understanding, the essence of composing involves defining the scope of the latter. If we take into account the rapidly changing requirements of systems today, systems must be flexible enough to allow independent improvement by exchanging components.

1.2 Contributions

With this introduction as a statement of the problem, the intended message of this thesis is *component composition* and the *communication protocols* that are required to do it. The discussion is supported by a software artifact developed at the Institute for Computer Systems, ETH Zurich.

The most visible result of this thesis is a document-based component architecture for the Oberon system called *Gadgets*. The idea of the Gadgets system is to simplify the construction of Oberon user interfaces and applications by composing prefabricated software components, with the goal of making this type of software construction accessible to a larger audience (possibly including end-users).

An innovation of the Gadgets system is to treat documents and application user interfaces in a uniform manner. Both consist of software components that can be composed at run-time, which makes the step from document composing to user interface construction a simple one.

A further contribution is *portable documents* that contain, in addition to document data, the component code that the document requires. This allows the transparent transportation of documents between Oberon implementations even if the destination machine does not have all the necessary components pre-installed.

Another contribution is a multi-level architecture for software construction, with levels ranging from interactive composing, composing by programming, and programming new components. Central to this theme is that application modules and application user interfaces are separated and can be developed independently. By interactively attaching prefabricated or own code to user interfaces, the documents are more than just attractive mock-ups but fully functional *active documents*.

A further contribution is a system architecture based on a set of hierarchies. The hierarchies separate different concerns and involve issues like code reuse, component classification, message protocols, run-time organization and persistency. One important hierarchy is a flexible and extensible communication infrastructure defining a compact set of high-level messages that allow components to be reused everywhere they are required. The use of *open interfaces* allows the extension and exchange of components. To increase the openness and extensibility of the system, policies are not centralized but distributed throughout the system into components that manage other components. This reduces the risk that a system extension should fail by introducing different sub-systems within the system itself, each of which adopt local policies.

1.3 Overview

The thesis is organized as follows:

- Chapter 2 introduces commonly used terminology and development techniques. The chapter includes short discussions of popular component systems.
- Chapter 3 gives an overview of the Gadget system, illustrating the interactive composition of components.
- Chapter 4 presents the design of the Gadget system, concentrating on how concerns are separated in hierarchies.
- Chapter 5 introduces component principles and their realization.
- Chapter 6 discusses aspects surrounding the programming of visual gadgets. It includes a discussion of message protocols and the Gadgets imaging model.
- Chapter 7 presents the Gadgets document model and discusses security aspects surrounding the transportation of portable documents.

- Chapter 8 presents conclusions and reflections.

In distilling the essence of the system for this thesis, many interesting aspects of the system have been left out, for example a description of the standard components of the system and the implementation details of the more interesting ones. The interested reader is referred to the Gadgets user and programmer documentation for details.

Chapter 2

Component Software

This chapter introduces common software construction techniques and defines the terminology used in the remainder of the thesis. The chapter includes a short discussion of related work.

2.1 Terminology

The following definitions introduce both the vocabulary of software construction techniques and their approximate historical development.

Closed and open systems. A closed software system solves a fixed problem defined by its manufacturer. It can only be used as it is. The flexibility of a closed system is increased by making it *configurable* so that a client can tune it within its design parameters. An *open system* allows a client to increase the product functionality above which was delivered by its manufacturer. Extending the product requires a knowledge of its internals and an ability to "plug in" custom code. A system open on all levels of abstraction that can be extended while running is called an *extensible software system*.

Procedure libraries. Procedure libraries—the earliest attempt at reuse—consist of a set of procedures and functions linked to application code. Procedure libraries that form an additional layer between the operating system and the application are sometimes referred to as *toolboxes*. Communication is mostly one-way from application to toolbox, although *callbacks* from the toolbox to the application sometimes happen. Procedure libraries often consists of a "flat" interface of hundreds of procedures that have to be called in the correct sequence. Consequently, they require a large learning effort. Early procedure libraries were weak on information hiding and encapsulation as they relied on global data for storing results between procedure invocations. With these, modification of data formats have unpredictable effects on clients and the library itself.

Module libraries. Modules decompose large systems into more manageable separately compiled entities. An *abstract data structure* can be hidden inside a module with access to it restricted by the module's public interface. The advantage is that changes to the implementation do not affect clients, an important requirement of independently developed software. An *abstract data type (ADT)* combines the ability to generate many instances of a data structure (which is passed explicitly as parameter to the module) with a means to make the implementation opaque. Popularized by Modula-2 [Wir83] and Ada [Ich83], module libraries insulate clients from implementation changes, thus allowing independent component development.

Class libraries. Class libraries consist of prototypical templates for generating components or objects. An example class library is the set of classes implementing all sorts of *collections* in the Smalltalk system [GR93]. Classes allow the possibility of sharing similar interfaces and code patterns through *inheritance* of features by one class from another. When a programming language combines intrinsic support for objects, classes and inheritance, it is said to be *object-oriented* [Weg90]. *Polymorphism* allows different objects to be generically manipulated. The most commonly occurring type of polymorphism is called *inclusion polymorphism*, where objects that belong to related classes are type conformant with each other [CW85]. The type compatibility of classes is determined by a *class hierarchy* that organizes classes according to shared features. *Late binding* allows objects to behave differently when the same operation is applied to them. The common vocabulary, in the case of class-driven dispatch, is that of *sending* a *message* to an object which interprets the event using code called a *method*.

As inheritance distinguishes object-oriented programming from class-based and object-based programming, the object-oriented technique primarily supports *component construction*. This makes it possible for the component constructor to derive a new component (an extension or derived class) from a similar one (the base class) by sharing most of the implementation with the latter. This is called programming by *specifying differences* or *specialization*. The component constructor uses a so-called *super-abstraction* interface to "patch" an existing class to create a derived class. In contrast, as the component implementation (including the use of inheritance) is hidden from a client, the latter sees nothing more than a collection of ADTs[1]. The latter is called the *client* interface.

Making use of the super-abstraction interface for code inheritance is a technique full of dangers and misunderstandings. Independent developers, one working on the base and the other on the extension class, often work against each other. Changing the base class can invalidate the extension because inheritance breaks the encapsulation of the base class. In addition, the interaction between base and derived class is often difficult to follow and explain [TGP89].

Scripting languages. Scripting languages differ from conventional programming languages primarily in their objective. They are often interpretive in nature, of a very high level, and "glue" together large-grained components like applications. The interface between a scripting language and an application can be as primitive as reading and writing character streams. Such a simple interface makes scripting applicable for controlling applications regardless of how they were implemented, and is thus ideally suited for orchestrating independent closed systems. Scripting systems include UNIX shell scripts [Bac86] and Tcl/Tk [Ous94].

Scripting is a flexible component composition technique popular for quickly constructing applications. On the other hand, the two-class nature of scripting languages, namely script and controlled components, limits the integration and efficiency that can be obtained—as component composition language scripts may be sufficient, but as component construction language they are lacking.

Frameworks. A framework is a set of abstract classes that provide basic (and incomplete) functionality for building a system [JF88, CP95]. A framework can be likened to the wooden frame of a house that needs to be completed by the owner. The electrical wiring is already installed but the home owner still has to plug in the appliances and paint the walls. You build something *on top of* a class library, whereas you build something *inside* a framework. This involves making concrete classes from the abstract classes provided. The task of the framework is to take control of the communication between components in the system, by

[1]Languages like CLOS [Kee89] and Dylan [Com94] go so far as to hide sending messages behind a *generic function* interface that looks like a conventional procedure call. This allows the library designer to defer the decision of exploiting late binding or not, or even change his decision at a later time.

calling the component when it needs to know or do something. This is in contrast with clients of a class library that invoke operations on objects under own régie. In a way, a component framework provides parts of the communication related to *component composition*.

An *application framework* provides the basic functionality of a typical application, like a user interface window, a menu-bar, a print button, and the like. Well-known application frameworks include MacApp [Com90], Smalltalk [GR93], Unidraw [VL89] and ET++ [WG94]. A *component framework* provides the basic functionality when building certain classes of software components. Examples include Smalltalk with its Model-Viewer-Controller framework [KP88] and CommonPoint [CP95].

Frameworks inherit the problems of class libraries and may add a few more in the form of fixed user interface guidelines. Frameworks like [CP95, WG94] require the same features from applications to make users feel "at home", like for example a universal undo mechanism and a standard print dialog box. Although this is a positive feature, a drawback is that unexpected requirements not foreseen by the framework designers might not fit well into the framework. Generally it is difficult to predict how a framework will be used in future, which turns into a problem as the communications infra-structure is wired into the abstract classes and cannot be adapted.

Compound Documents. The general dissatisfaction with and failure of integrated systems that combine functionality like word processing, spreadsheet, database and whatever else users are supposed to have in document processing, is related to the fact that end-users have diverse needs that a single package cannot fulfill. Instead of indiscriminately adding features to an integrated package, *compound documents* allow the end-user to compose selected components from different vendors into a single document. The ubiquitous example is that of a spreadsheet component embedded inside a text document. This approach is called *data centric* as end-users stop thinking about applications that manipulate data and start thinking of editing, copying and pasting data components (that act like "mini applications" or applets). In this thesis we will often use the term *document-based* for systems that adopt compound documents as the primary means of interacting with the computer.

In comparison with class libraries and frameworks, compound documents emphasize document composition at run-time, and thus require a flexible glue to keep components together, something similar to a *software bus*. This bus can be compared to a hardware bus on a failure tolerant computer where adapter cards can be plugged in and taken out while everything keeps on working. Most compound document architectures inherit features from component frameworks, requiring the overriding of methods to handle events, to display, to print, etc.

There are only a few compound document-based systems in use today. An early pioneer was the Andrew system based on X-Windows [PHK+88]. The best-known commercial system is called Object Linking and Embedding (OLE) from Microsoft [Bro93]. At the time of writing, a first version of OpenDoc [Tea93, Côt96] has been announced.

Component Architectures. Component architectures or *Componentware* attempt to address component composition instead of component construction. The goal is to simplify and unify the communication structure between components so that prefabricated components from different vendors can easily be combined. The ultimate goal is easy component interfacing. In this regard, a component architecture is of a larger scope than that of compound documents, which concentrate on the layout of documents only. Component architectures require the standardization of communication protocols between objects.

Recently there has been a lot of competition between different component architectures, with practically all the letters of the alphabet occupied in acronyms (see [Ude94] for an overview). We distinguish between document architectures, that correspond to compound documents; object architectures, that define how objects communicate; and communication architectures, that support communications across the network. For example, the Common Object Request Broker Architecture (CORBA) is concerned with inter-object communication on a network.

2.2 Examples

As examples we will discuss three document architectures: Andrew, OpenDoc and OLE-2. In addition, ET++ is used as an example of a prototypical component framework, and Visual Basic is introduced as a minimal component system that has had notable success.

2.2.1 The Andrew Toolkit

The Andrew Toolkit [PHK+88] from Carnegie Mellon University is an independently extensible object-oriented framework that provides the foundation for constructing document-based user interfaces on UNIX systems. The toolkit provides the usual set of simple components, such as menus and scrollbars, and a number of higher-level editable components, like formatted text, tables, spreadsheets, drawings etc. Components can be embedded in each other at run-time by an end-user. A generic editor called EZ can edit any component by loading the appropriate code when needed.

The Andrew Toolkit (ATK) is written in a C-based language similar to C++. The Andrew Class System (Class) permits the definition of object methods and class procedures. Class is based on a preprocessor design. The main reason for using Class was to support dynamic loading and linking, an unsupported feature of UNIX systems at the time of the system design.

Components in the ATK are called *insets*. An inset typically consists of two parts, namely a view object and a data object, corresponding to the view and model components of the MVC framework [KP88]. Only data objects may contain persistent data, from which the view objects can be reconstructed on the fly. To work around this restriction, a third observer object is connected between the view object and the data object. It conceptually contains persistent data associated with the view object. The external representation of data objects is a text file with embedded data enclosed with text mark-up. Data objects may be nested in each other in the text file. The ability to transport text files easily is exploited by the Andrew Messaging System, an application that allows the mailing of documents containing multi-media insets.

View objects are nested in a tree structure which plays an important role in distributing events. Events, like from the mouse and keyboard, are passed from the root of the view tree towards its leaves. Parental authority allows a parent to control the events passed to its children. This important system concept is very similar to the parental control property of the Gadgets system (cf. 3.2).

There are two ways to attach additional functionality to insets. The Andrew Development Environment Workbench(ADEW) provides a C code generator for templates called *controllers* to access insets in a document [Neu91]. *Ness* is a dynamic programming language embedded in source form inside text documents [Han90]. Ness specifies the actions to be completed when insets are activated by the user. The security problems of executing a Ness script is only partially addressed by a static source code scanner that flags “dangerous” Ness statements, assuming that the reader has the ability to verify received scripts.

2.2.2 OpenDoc

OpenDoc [Tea93, Côt96] is a vendor-neutral standard for cross-platform compound documents, that was released just at the time of writing. It was developed by Component Integration Labs, backed by Apple, IBM, Novell, and several other companies. Components in OpenDoc are called *parts*. Parts can be embedded in each other and contain their own editors. A layout system negotiates screen layout and a dispatcher passes events to parts. An arbitrator controls access to shared resources. Parts are presented in a shared window in the form of *frames*. Frames divide the window into separate regions for each part, and is used by the dispatcher to determine which part editor should get an event. New event types can be added to the dispatcher at run-time, and the arbitrator can be extended with new resources. In addition, a part editor can monitor the dispatcher and watch for specific events that pass through it. Parts are divided into part viewers and part editors. The part viewers are intended to be freely distributable and the part editors sold at an appropriate price. The object model is based on IBM's System Object Model (SOM).

Interestingly, the stated goal of OpenDoc is not to make the construction of compound documents as simple as possible, but simply to make it possible. OpenDoc is thus not an object-oriented framework in the traditional sense of the word. OpenDoc restricts itself to layout handling, event handling, storage, and scripting. The storage sub-system is based on Bento [HR93], which supports multiple formats, versioning, and the ability to insert or delete data in the middle of a data stream. Bento has a meta-level architecture that, for example, includes the references between parts, so that cloning of data portions can be made without actually loading the parts. The scripting mechanism is called the Open Scripting Architecture, and is based on the high-level Apple Events.

Although the imaging model is sophisticated and contains numerous classes for windows, layout, frames, canvasses, shapes and transformations, OpenDoc handles only *geometry* and not actual imaging. On each hosted platform, parts use the drawing primitives provided by that platform. A solution that addresses this portability problem, called the OpenDoc Development Framework (ODF) is planned in which the GUI controls are called *gadgets*. The current OpenDoc implementation sits on top of the hosted operating system. Apple and IBM have however announced integrating this technology into their operating systems to take advantage of plug-in components at every level.

2.2.3 OLE-2

Object Linking and Embedding, or OLE-2, is the component and compound document architecture from Microsoft [Bro93, Lin96]. The user interface components of OLE, called OLE controls, are based on the Component Object Model (COM), also from Microsoft. COM objects have a very simple structure, consisting of a variable number of communication *interfaces*. Interfaces are collections of method pointers that an object exports and that can be enumerated at run-time. COM does not support code or interface inheritance and instance variables. Components are extended by wrapping them inside other components—a process Microsoft calls *aggregation*. Microsoft is working on a further development called Network OLE, that will allow objects to communicate across the network based on remote procedure calls [HS96].

OLE controls are the components of compound documents in OLE-2. An OLE container component may contain OLE controls. The controls receive instructions from the container by a built-in receptor called a *sink*. A control can communicate with a container after a hand-shaking process that dynamically sets up a new sink in the container. The programming of OLE controls is simplified by the Microsoft Foundation Classes, by managing many events in a default way. Methods provide the control with basic behavior, and properties control its appearance.

Even though OLE-2 is difficult to understand and difficult to build applications with, the wide support for Windows motivates many companies to use it. This is why OpenDoc also accepts OLE-2 controls as parts. Unfortunately, without a redesign of the many mammoth Windows applications, embedding one of them in the other with OLE-2 makes little sense due to exorbitant memory requirements.

2.2.4 ET++

ET++ [WG94] is a class library and application framework integrating user interface building blocks and basic data structures. Implemented in C++, it is portable between several operating systems and windowing platforms and consists of about 300 classes rooted mostly in a class *Object*. The classes are organized in layers, for example to abstract the host system, toolkits for data structures and user interface, frameworks for a desktop environment, browsers and debuggers. The system supports dynamic loading and linking of objects at run-time, although the feature is not extensively used. Run-time type information about the names and types of instance variables support run-time exploration of the system. No meta-level architecture is used; most of the information is extracted using macros from the source code. The persistency mechanism is based on methods *PrintOn* and *ReadFrom* associated with objects to write and read data to and from streams. Streams can either be files or memory buffers, the latter being used to make deep copies of arbitrary complex polymorphic data structures.

The view sub-system includes a base class of visual objects—called *VObject*—that can draw itself, handle input events and manage their size and position. *VObjects* are very lightweight, having no built-in coordinate transformation and establishing no clipping boundary. They react to input events by overriding methods in the base class. *VObjects* are combined in collections by a class *CompVObject*. *VObjects* pass events that they cannot handle to their container, thus inverting communication in comparison with the Gadgets system (cf. 3.2). The *Clipper* class defines an independent coordinate system and clips the graphical output of a *VObject* to a rectangular area. *Clippers* can be nested in each other. Display update is based on *invalidation*; all invalidated areas are redrawn when the system is idle. Change notification is based on dependency lists as in MVC [KP88].

The ET++ framework contains high-level building blocks like a rich text building block, a grid view for tabular data, and a tree view for hierarchical data. The application framework provides support for multiple documents per application, data conversion between different formats, and undo-able commands.

A disadvantage of ET++ (as in other class libraries and component frameworks) is that object composition must be explicitly programmed out. The connection cost of connecting two classes with different interfaces can be high, often requiring a third class to make the needed conversions. This is especially problematic when classes are developed independently. To solve the problem of high connection costs and make prefabricated objects easier to connect with each other, ET++ has recently been extended with a component architecture consisting of about 60 classes [Ble95]. This development is loosely based on concepts from the Gadgets system. In this extension, components are connected to each other with a component bus. The component bus corresponds approximately to the message infra-structure that connects children of a container in the Gadgets system. Two component buses can be connected with each other only over a router component. Components communicate with each other using synchronous messages, either in a broadcast or uni-cast manner. A message contains a sender, destination, message name and message arguments that can be of any type and number, and are self-describing with meta-level description. The ability to overload operators in

C++ is used to simplify the construction of messages. To allow components to understand each other, some messages are standardized. One family of messages allows the exchange of simple data types, and corresponds to the attribute message in the Oberon system. In contrast to Gadgets, programming is still required to attach code to dialogue elements. The current implementation is a prototype and supports only a single component bus and a few components. Also dialogue components cannot be nested in each other.

2.2.5 Visual Basic

Visual Basic [Cor93] from Microsoft consists of a Basic interpreter, an interactive GUI editor, and a programming environment for the development of stand-alone programs. It includes a large set of prefabricated visual components called *controls* for static user interfaces. Controls have editable properties and user activated events that execute Basic procedures. An *Inspector* is used to modify properties interactively, and the Basic interpreter allows direct access to properties in the form of control instance variables. Direct procedural communication between controls is not possible. This has resulted in a dubious programming style where changing properties have additional side-effects that approximately correspond to calling a procedure in a conventional system [Gla95]. Visual Basic does not support the MVC model [KP88], does not allow the nesting of controls, and is not object-oriented. The latest version of Visual Basic allows the construction of controls using Microsoft's component architecture called OLE-2 [Bro93], which can be reused in other environments too. Visual Basic and a similar product Visual C++ are popular due to the large number of off-the-shelf controls.

2.3 Summary

This chapter summarized the software construction techniques of procedure libraries, module libraries, class libraries, scripting languages, component frameworks, compound documents and component architectures. Examples from the domain of document and component architectures were presented.

Chapter 3

The Gadgets User Interface

3.1 Introduction

The Oberon system is a single-threaded, single-user, co-operative multi-tasking operating system [Wir88a, WG92] that runs on bare hardware or on top of a hosted operating system as a single-window application [Fra93, BCFT92]. The latest version of Oberon, called Oberon System 3, is an extended Oberon version that has intrinsic support for persistent objects and for building graphical user interfaces. The major client of these new features is the Gadgets system, a new compound document-based user interface [Mar91, Mar94a, Gut94b, Gut94a, Mar94b]. This chapter introduces the Gadgets system from the perspective of the end-user. It describes the conceptual model that an end-user has about the system.

The Gadgets system relies on compound documents as the user interface for both documents and applications (cf. 2.1). To aid in the construction of full applications, the Gadgets system extends the compound document architecture into the application domain by adding non-visual components. Correspondingly, application user interfaces are regarded as documents with attached functionality, and any document is a potential application user interface.

The idea of unifying user interfaces and documents in this way is not completely new. For example, a similar idea was the basis of a system called EmbeddedButtons [Bie91] developed at Xerox PARC for the Cedar system. The latter system was limited to end-users creating *active documents* consisting of arbitrarily shaped *buttons* with attached functionality written in a scripting language. In comparison, the Gadget system uses a larger and more diverse set of components and is a full fledged system.

The components of the Gadget system are objects called *gadgets*. Gadgets range from visual dialog elements like buttons, text editors and documents, to non-visual components like model gadgets that manage application data.

Before continuing with a more detailed description of the user interface and the role of gadgets, it is worthwhile to discuss the system's design goals as a reference when reading the remainder of the chapter. Succinctly, the design goals of the system include multiple programming levels, tangibility and composability.

Multiple programming levels. Gadgets emphasizes ease of use, understanding and implementation. The inherent complexity of the system is abstracted with an incremental approach to using and programming the system.

- *Level-0 programming* of Gadgets involves the interactive composition of documents and user interfaces, and the dynamic connection of functionality to user interface elements. This step involves interactive layout and composition of components, modification of component properties, connecting data models to the user interface, and the specifications of actions invoked when a user activates components. An elementary scripting facility allows combining components in more complicated ways. This level of programming is accessible to the largest audience, including end-users.

- *Level-1 programming* involves Oberon programming to create *glue* that binds components together. This level hides the Oberon messaging scheme (cf. 4.2.1) from the novice by providing more convenient procedural interfaces. To separate the application code from the user interface, Gadgets uses the Model-Viewer-Controller framework [KP88] to create an intermediate layer of model (or data) components between the user interface and the application. This insulates the application code from changes in the user interface. At a lower level of abstraction, the message passing techniques are available to the programmer for programming more specialized features.

- *Level-2 programming* involves the programming of new components. Different skill levels depending on the classification of the programmed component as model, elementary, container, view or document gadget are required (cf. 3.3). For example, model gadgets are easier to program than elementary visual gadgets, which is again easier to program than a visual gadget that contains other gadgets.

Tangibility. By tangibility is meant giving components a certain amount of "realness" to the user. Just as a geometric figure in a drawing can be edited in place, components like buttons, checkboxes, sliders, text editors and other user interface elements are activatible and editable in-place. We want a user not only to click on a button to complete an action, but also to grab the button and move it around, resize it, or delete it, just as intuitively as manipulating a geometric figure in a drawing editor. We also imagine grabbing parts of a user interface, dropping them into an e-mail and mailing them to a friend. This requires that components can be dragged and dropped into *containers*, that components can be interactively connected to each other and to application code, that existing user interfaces can be taken apart, and that the properties and actions of components can be inspected and modified on-the-fly.

Composability. Composing involves reliable ways of combining components with each other. Composed components communicate by sending messages over connections that are connected and disconnected at run-time. One way of composing components is to nest visual gadgets inside each other: a *complete integration principle* requires that any visual gadget can be inserted in a container gadget. This ensures that components break application boundaries and can migrate from one domain to another. Another way to compose is to link components with communication channels. These channels are used, for example, to connect model components with view components in the MVC framework [KP88].

3.2 User Interface Vocabulary

Before continuing with a classification of gadgets and with examples of their interactive composition, we introduce definitions and conventions that will simplify reading further chapters. Note that the Gadgets system adopts many features of the Oberon system, even though it is intended as a complete user interface replacement. For completeness we also review some of these features.

The Display. Figure 3.1 shows a typical display organization, vertically divided into the *user track* on the left and the *system track* on the right. The user track contains documents undergoing editing and the *system track* contains *tools* that operate in a remote fashion on documents. Each track is further tiled into viewers, each with a menu bar and content. The menu bar shows the document name and

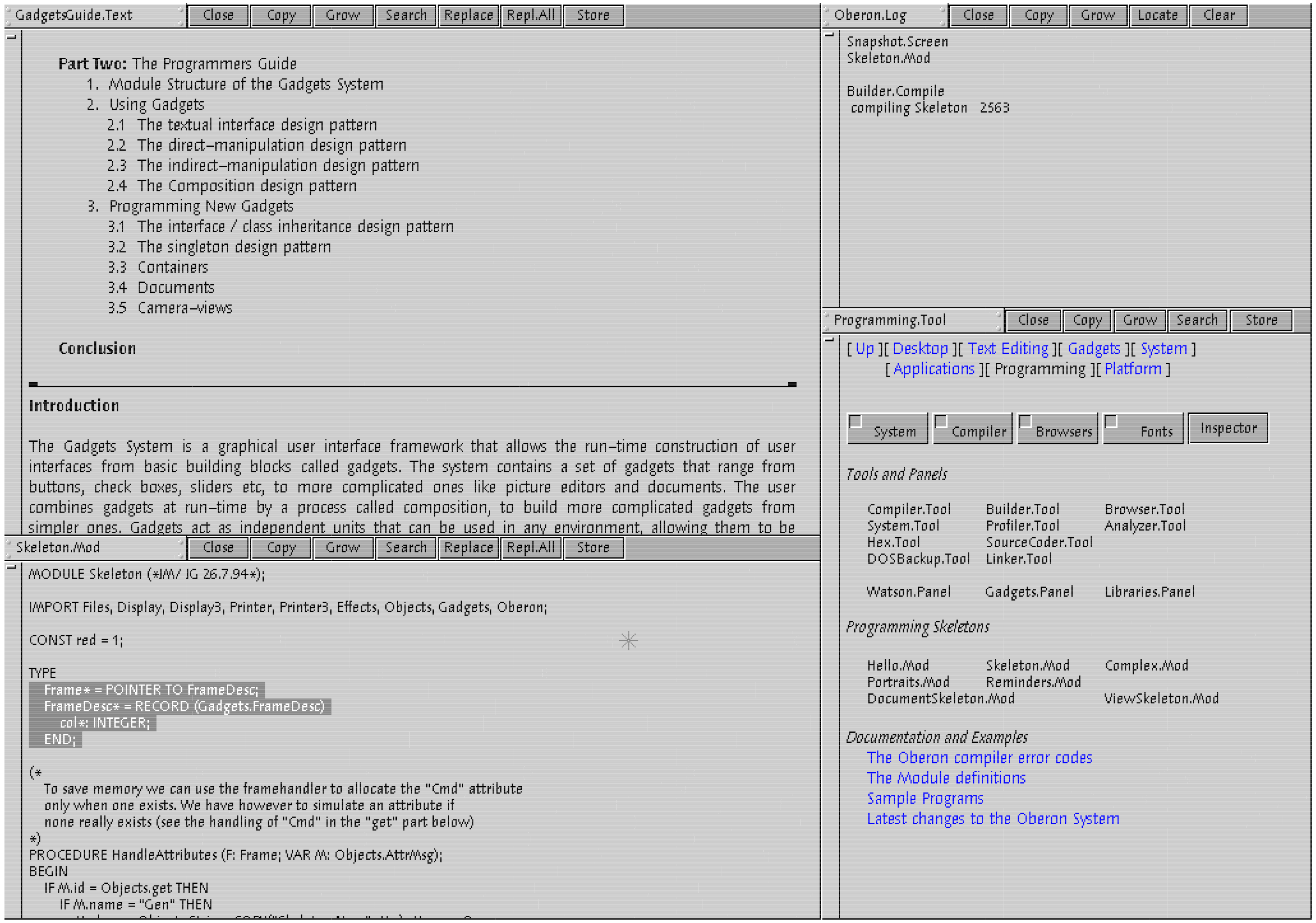

Figure 3.1: The Oberon screen organization

buttons that operate on the document. In this snapshot, all viewers contain text—both program and prose text are displayed and edited in the same manner. The programmer tool on the right contains gadgets embedded inside the text. Note that in this, and in all further screen snapshots we will see, all user interface objects on the screen are gadgets.

Commands. The user controls the Oberon system with *commands*. A command is either a mouse event, a keyboard event, or the execution of a procedure. Mouse and keyboard events are interpreted by the gadget located at the mouse pointer or having the keyboard focus. A word in the form *M.P*, where *M* is a module name and *P* is the name of an exported and parameterless procedure, can be invoked directly by clicking on it in a text document. To speed up working with the system, text documents called *textual tools* are prepared with collections of related commands.

Commands obtain parameters from the system state, do some processing, and change the system state. An example is compiling an Oberon module with the compile command, with the compiler reporting success or failure to the log located in the upper right corner of the display. Another example is clicking on a directory command to show files that match a specified pattern in a new text document (which is immediately ready for further editing). A common way to pass parameters to commands is with a text string that is written immediately following the command. Often one command can deliver results to the following command. For example, a compile command can be followed by a list of module names obtained from the directory command. Because of the use of text to pass parameters and deliver results, the Oberon system and this sub-system of the Gadgets system, is said to have a *textual user interface* or *TUI*.

Gadget principles. Gadgets distinguish themselves from other objects in the Oberon system by their conformance to a set of shared rules as defined by the Gadgets architecture. These rules include the principles of *complete integration*, *parental control*, and *small world*.

The principle of *complete integration* ensures that a component can be used wherever it is required without regard to application boundaries. This is also referred to as gadget components being *general*, or being *first-class citizens*. This principle has the pleasing property that any component, even one written for a specific application, can potentially be reused in another application. To emphasize this point, we can imagine taking a figure gadget like an ellipse from a drawing application and use it to annotate our own user interfaces.

The principle of *parental control* delineates non-atomic components like nested part-of hierarchies into boundaries that reflect responsibility. A container component takes complete responsibility for all of its substituent parts. In return, a part has to accept the authority of its container. The hierarchy of responsibility so obtained not only reflects the way components are made persistent (cf. 5.4) but also in the way inter-component communication takes place. A container component has the right to monitor the communication with and between its parts, suppress it or even modify it, thus ensuring that the part components behave in the way the container requires them to. In return, the container has to provide an acceptable level of service to its parts. In practice, a "friendly" co-operation between container and its parts is the norm, where the container takes the initiative in asking a part for more information about its requirements.

The *small-world principle* delineates non-atomic components like part-of hierarchies into information hiding boundaries. The internal structure of a component is hidden, and communication with a sub-component is done indirectly through the container's message interface. Shared message protocols combined with the parental control principle, allow the internal structure of a component to vary. Let's say for example that an engineer wants to create a new bitmap component where each pixel of the bitmap is an object itself. The resulting memory efficiency problem that results when each pixel becomes an object most probably rules out using the standard gadget types. The small world principle however makes it possible to optimize the internal data structure for bitmap storage and *pretend* to the outside world that all messages sent to the bitmap are handled correctly. This, in effect, hides the internals of a container object like a bitmap from the outside world and still allows—with appropriate programming—for existing components to masquerade as a pixel. As long as the bitmap provides a suitable "container like" interface to other components, none of those embedded components are the wiser.

The principles sketched above have several consequences for programming components and the system itself. For example, as each component is completely responsible for its embedded components there is no way of making global statements about the system. The lack of any global assumptions makes the system more extensible and open to different directions of extension. As a disadvantage, it can complicate the way in which components are programmed, as they have to assume much more responsibility.

Gadget classes. Each gadget instance belongs to a class that is determined by a *generator*. A generator is a procedure that generates an instance of that gadget. After generation, the class of a gadget is determined by its type and behavior (as determined by a *message handler*). (More details about handlers are presented in section 4.2.1.) Two gadget instances of the same type with different message handlers belong to a different class. Because of the possibility of exchanging the message handler in Oberon, the class can change at run-time.

In a side remark, note that our terminology for type and class deviates from the literature [Weg90, Lun89, Nel91b]. Although many object-oriented programming languages do not make a distinction between type and class, the terminology is that types define conformance (what can be assigned to what) and classes define implementation. A type can have many implementations (classes) and a class can implement many types. The correspondence between message handlers and classes is discussed by Szyperski [Szy92b]. Rather than introducing another term for types that have the same implementation, *class* is used here in the spirit of languages that do not distinguish between the two concepts.

Attributes. Attributes are properties of gadget instances that define their state, representation and behavior. Each attribute has a name and a typed value. Allowed types are string (or ARRAY OF CHAR), LONGINT, REAL, LONGREAL, CHAR and BOOLEAN, a subset of the Oberon language elementary types. Attributes are distinguished from the instance variables of a gadget by being visible and modifiable at run-time by the end-user using an attribute *Inspector* (Figure 3.2). The Inspector uses a universal message protocol to enumerate, inspect, and modify attributes of any gadget instance (cf. 5.3).

Two important attributes of a gadget instance include the specification of its generator and its user-defined name. The generator attribute is used during externalization and internalization to keep track of the class (and is read-only). The name attribute is used to search for gadgets at run-time. Names need not be unique—a scoping mechanism based on the nesting of containers is used to resolve ambiguities (cf. 5.3.4). Each gadget class typically defines further attributes.

Note that the accessibility of attributes by the end-user places a responsibility on the programmer of the class to "export" only those internal details of a gadget that can easily be understood and that can be of possible use to the Oberon user. Attributes thus tend to be limited in number—typically less than half-a-dozen—and tend to be of a simple nature.

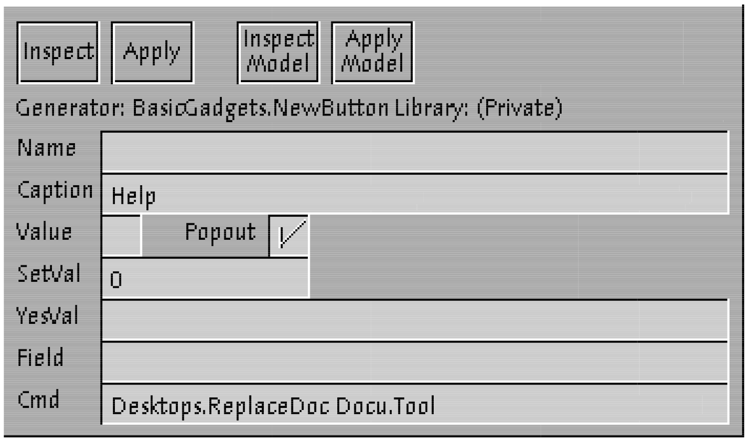

Figure 3.2: The Inspector

Links. A link is a named uni-directional reference from one gadget to another. A gadget instance might have several links to other gadgets. The links are typically used to remember an “acquaintance” of a gadget. One use for links is to couple a model and view gadget according to the MVC framework [KP88]. Just like attributes, links are intended to be accessible by the end-user for constructing more complicated gadget data structures at run-time. It is important to note that links are not the only references between gadgets—they are only the ones that are visible to the user.

The Inspector supports following “Model” links from view to model gadget. A more advanced tool called Columbus [Sal96] allows the interactive navigating and construction of links. Both tools use a universal message protocol to enumerate, retrieve, and set the links between gadgets (cf. 5.3.2).

Documents and libraries. Gadget collections called *documents* can be saved to a file and later reloaded from the file in exactly the same state. The persistency technique is based on *libraries* (cf. 5.4). A library is a collection of persistent objects, each object being addressable by a name or a number. By inserting gadgets into a *public library*, they become accessible to the system as a whole. Public libraries play an important role in organizing reusable components. An example library is a collection of shared gadgets like icons, menu bars and the like. Public libraries are loaded on demand when an object of the library is referenced by an application. Most applications either use direct references to the components in a

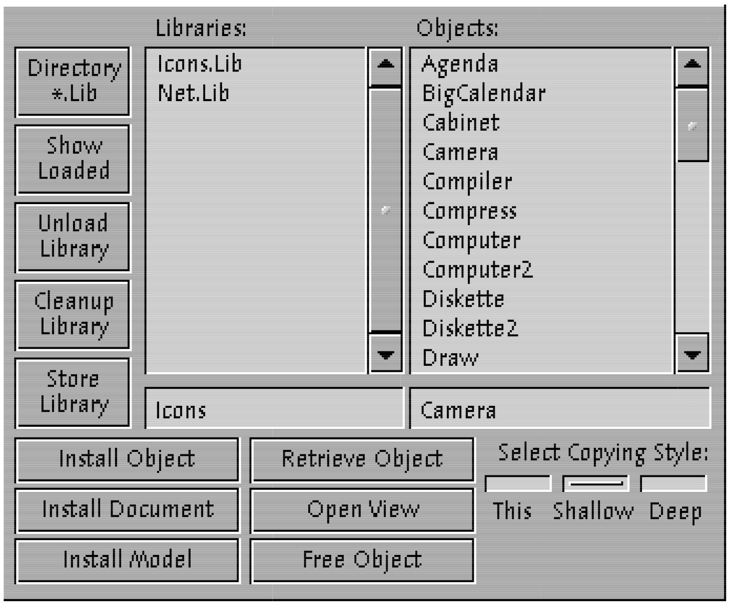

Figure 3.3: The library management tool

public library or make copies of its contents as required. An application for the end-user management of public libraries is shown in Figure 3.3.

Both public libraries and the components they contain are named. This enables the user (and programmer) to refer to a public object in the form of a string "L.O", where *L* is the library name and *O* the object name. This syntax is similar to commands in the form *M.P* (where *M* is the module name and *P* is the procedure name). Such similarity has caused the adoption of another concept from modules, namely that of *import*. A component data structure is said to import a public library should it contain at least one reference to a component of that library. As nothing prevents a component that imports a public library from belonging to a library itself, it is possible to build a library hierarchy similar to that of a module hierarchy. We accordingly say that one library imports another[1].

Cloning. It is often simpler to copy existing gadget configurations instead of creating new ones. We imagine the user or programmer making copies of a shared

[1]The similarity is illustrated in [Tem94] where Oberon modules are recast into a public library-like concept similar to that of Oberon System 3, unifying the module and library hierarchies.

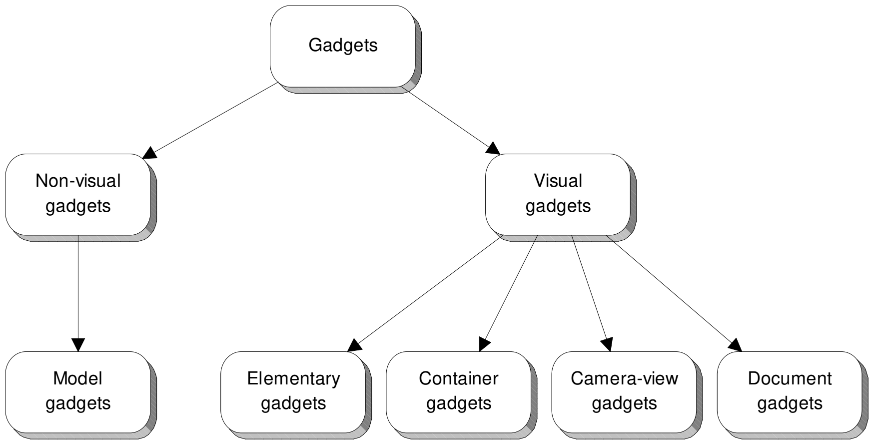

Figure 3.4: The logical gadget classification hierarchy

"template" component (a gadget in a public library) whenever they are required. The Gadgets framework does this by sending a *clone request* (cf. 5.3.3) to a component, expecting the component to deliver a copy of itself. The references between gadgets, for example those in a whole-part data structure, bring up the question what exactly is the meaning of copying a component. In our implementation, a *shallow copy* returns a copy of the receiver component containing the same parts as the original. A *deep copy* recursively forwards the copying request to the parts of the receiver to make a complete copy of a structure from the receiver onwards. Special precautions are made to make structurally identical copies when components are connected in a network (cf. 5.3.3).

At first glance, it seems that cloning is an aspect that falls in the domain of the programmer instead of the end-user. Although largely so, there are situations that the end-user should be aware of as they affect the way a user interface behaves. One such situation is related to the user copying visual gadget in the MVC relationship with model gadgets (cf. 3.3.1).

3.3 A Gadget Classification

A classification hierarchy organizes gadgets classes according to their purpose (Figure 3.4) and also reflects the current scope of the Gadgets system.

Two orthogonal properties are implicitly present in the classification: *atomic-*

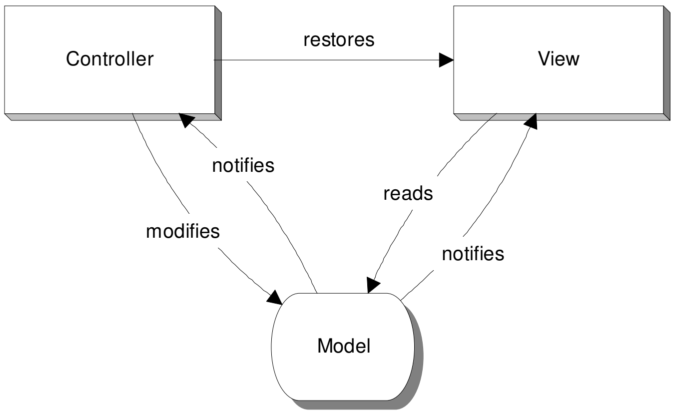

Figure 3.5: The model-view framework

ity and *concreteness*. A component is atomic if it does not consist of any further components. A component is *compound* when it has component parts. A component is *concrete* if it can be seen on the display; that is, it is of a visual nature. A component is *abstract* if it cannot be seen; that is, if it is of a non-visual nature. Concrete and abstract components can for example be view and model gadgets respectively. Abstract components are often called non-visual gadgets, and concrete components visual gadgets.

3.3.1 Model Gadgets

Model gadgets are abstract components that enter into the model-view relationship as models. They contain application-dependent data and form the interface between the user interface and the application code. A model gadget is represented on the Oberon screen by a visual gadget—called a *view*—that knows how to represent the model's state.

The Model-Viewer-Controller (MVC) framework. The original MVC scheme (Figure 3.5) is slightly modified in the Gadgets system by merging the controller and view components. The MVC scheme works as follows. A communication protocol between model and view keeps the state and its representation synchronized. Should the state of the model change, the view updates its visual representation accordingly. Interaction between the user and the model state is through the controller. A mouse or keyboard event is interpreted by the controller in such a way to change the state of the model, which in turn causes a *change notification* to be sent to the view, which then updates itself accordingly. Application modules can also change the state of the model directly, for example to show the result of a computation.

The popularity of the MVC framework is related to two facts.

First, many views—perhaps of a different representation—can be linked to the same model. Change notification then involves sending a message to all views displaying that model. For example we can link a textfield gadget—used for entering short strings from the keyboard—and a slider—with its value adjustable by sliding a knob—to a model gadget representing a number. With such a configuration the user is able to pick how a number is input, either by typing or with the mouse.

Second, if application modules restrict themselves to only manipulating the state of model gadgets, they are relatively immune to changes in the visual representation (both in class and in their number). This gives the user interface designer more freedom in changing the user interface without having to adapt the application code.

Pluggable Models. More possibilities for component reuse are obtained when general-purpose models like numbers, strings, and text—not coupled directly with any specific application—are used. The Gadgets system provides a small set of model components like *Integers*, *Reals*, *Strings* and *Booleans*. To allow "plugging-in" these components into arbitrary views, a communications protocol for the exchange (and necessary type conversion) of elementary data types between model and view is used. Some views, like text entry fields, restrict the input possibilities according to the data type of their model.

The MVC composition interface is realized in such a way that a view can register interest in a specific (and named) *aspect* of a model. This is useful when the model gadget is a mediator between the view and the model's own parts. This allows, for example, a text entry field to represent a specific attribute of a model, as in the case of a model gadget representing a complex number, either the real or imaginary parts. Another variation is to guarantee consistent radio-buttons by grouping them together with a model gadget. Each radio-button activates or deactivates itself according to the value of the model. This requires that the radio-buttons register interest in a specific model aspect and also watch out for a specific value occurring.

3.3.2 Elementary Gadgets

The most numerous gadgets are the visual and atomic *elementary gadgets* (a selection is shown in Figure 3.6). Several elementary gadgets can be linked to model gadgets, including buttons, checkboxes, textfields, sliders, etc., while many others do not require a model. Attributes configure an elementary gadget's representation and behavior. An example is a *push button* gadget. Its attributes include one specifying the caption and another an Oberon command to be executed when the button is pushed. Whereas a command is explicitly visible in a text document, the elementary gadgets "hide" commands in *command attributes*.

The adoption of commands as the primary way to invoke actions from user interface components, has several advantages. Once users have grasped the concept of a command, they can immediately build graphical user interfaces equivalent to those of a textual user interface. In fact, many older Oberon TUI applications have been transparently extended with a GUI in this manner. This flexibility illustrates the loose coupling of a user interface and the application code. As all computation in Oberon is initiated by commands, the user can fall back on a large collection of commands. Also, using level-2 programming, a programmer can create new commands. It is imagined that a suitable factoring of commands, such as, the shell commands of UNIX [Bac86, Tan87], can create greater possibilities for constructing applications.

3.3.3 Container Gadgets

Containers are visual compound gadgets. The prototypical container is a *panel*, a rectangular surface containing other visual gadgets (Figure 3.7). We use the usual anthropomorphism when describing containers. The gadgets contained in a container are called *children*. The container is called the *father* of the children. A child has only one father. The *parental control* principle (cf. 3.2) makes fathers responsible for their children. Containers can be nested in each other. We refer to the parent, its parent, and so on, as *ancestors*, and the children and all their children as *descendants*.

The children are organized in a priority order that determines which will overlap others. They are clipped visually to each other and to the boundary of the container. The understanding between the container and the child is that the container will try to accommodate the changes in size and location that the child makes to the container. Depending on the container class it might not allow a child to behave in certain ways. Containers accept visual gadgets that are dragged

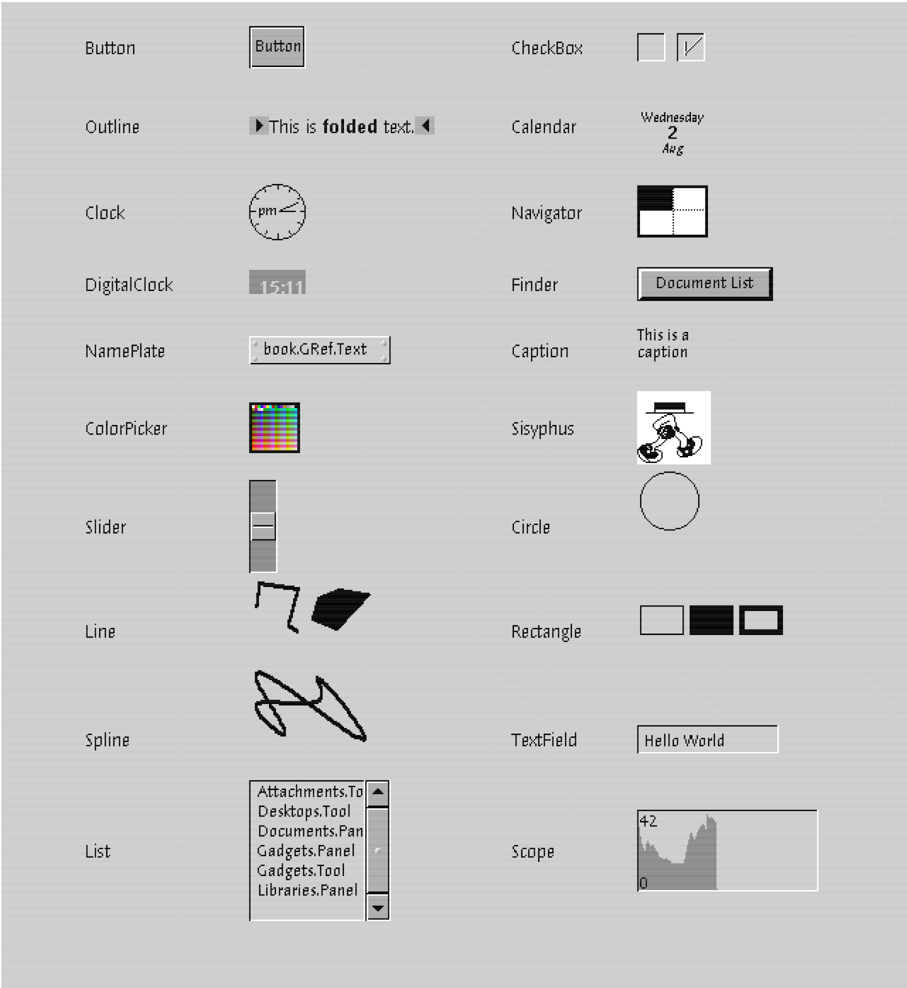

Figure 3.6: Examples of elementary gadgets

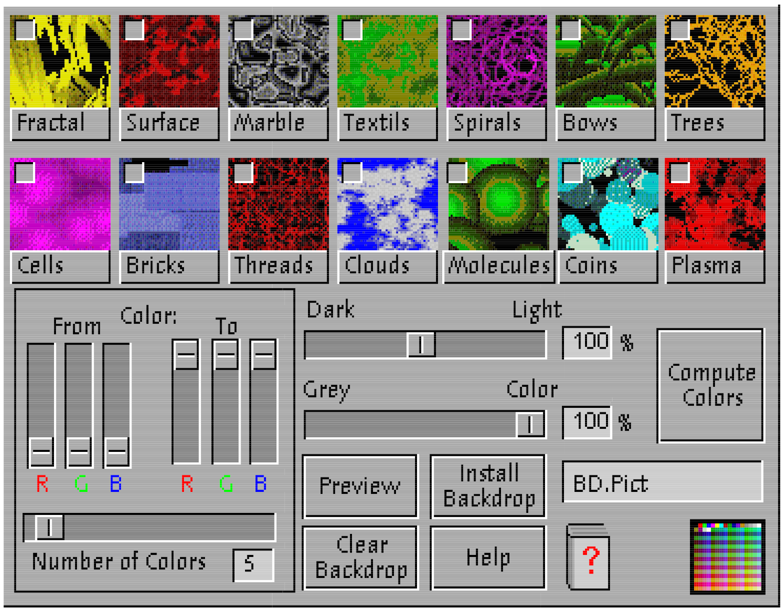

Figure 3.7: A panel example

and dropped into them and allow children to migrate from one father to another.

The event of dropping a visual gadget into another is called a *consume event*. In the case of a panel container, the consume event is interpreted as removing the gadget from the previous container and insert it into the destination container.

Component assembly. Containers like *icons*, *iconizers*, *notebooks* and *organizers* construct more complicated components by *assembly*. In a way similar to the popular Lego blocks, components are combined in different ways. For example, a notebook gadget logically orders its children one behind the other and gives you the chance to page through them—no assumptions are made about the class of its children. Iconizers are used to construct popup menus by adopting two gadgets, one used as the representation for the menu, the other as the surface that pops up (Figure 3.18). Organizers solve a simple constraint system for arranging children automatically. (The constraint algorithm is based on [Car86].)

Some other variants on the assembly idea have also been experimented with. In [Brä93] a client generator gadget of a discrete event simulation system is parameterized with a client arrival distribution by dropping an *event distribution*

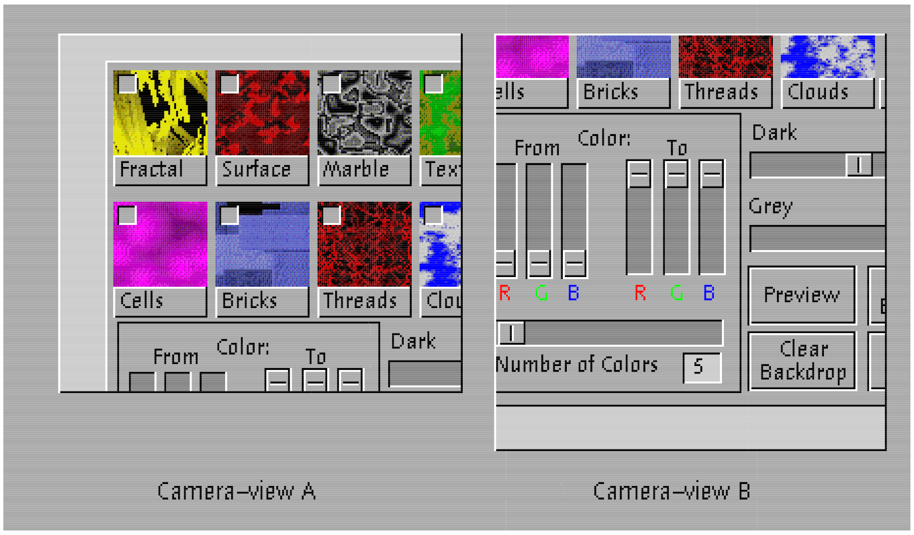

Figure 3.8: Example of camera-views

gadget into it. In [Wal92] simple electronic circuits can be built by gadgets and then simulated by throwing switches and re-wiring chip gadgets on the fly.

3.3.4 Camera-view Gadgets

Camera-view gadgets show another visual gadget from an adjustable perspective. As an example, Figure 3.8 shows two views of the panel in Figure 3.7. Many camera-view gadgets can show the same visual gadget, each with a different perspective. Camera-views are thus similar to views, except that they have a visual gadget instead of an abstract gadget as "model".

There are two important uses for camera-views. First, they are useful when multiple views of large visual gadgets are needed. This allows copying components from one section to another section further away than the size of the viewer. A second use is to share one and the same object in different containers. For example, the library mechanism allows the sharing of icon bitmaps between user interfaces by creating multiple camera-views of the same visual gadget. The effect of camera-view gadget is twofold. First, the same visual object can be visible—through camera-views—at different positions on the display. Second, the tree-like data structure of container gadgets on the display is modified to a directed a-cyclic graph (DAG). These effects influence the imaging model and the Oberon messag-

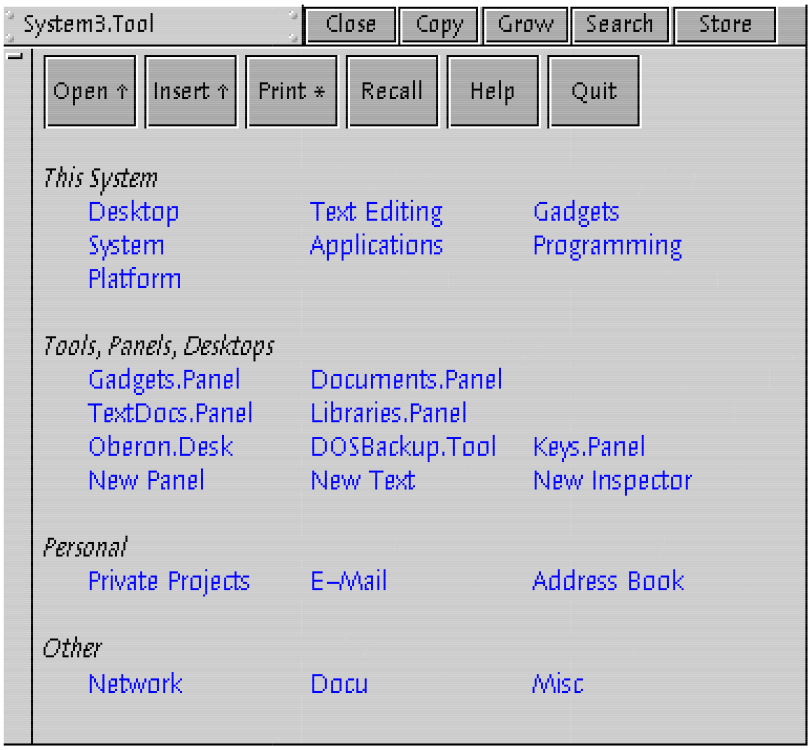

Figure 3.9: A document in a viewer

ing mechanism (cf. Chapter 6).

3.3.5 Document Gadgets

Document gadgets—the visual representation of documents—are compound visual gadgets. Documents are either displayed directly in the viewer system (Figure 3.9) or embedded into each other. Figure 3.10 shows a panel document embedded in a text document. When in the viewer system, documents are attached to a menu bar. Embedded documents do not have a menu bar.

The classic Oberon system uses a tiling viewer system (as previously shown in Figure 3.1). The Gadgets framework extends the latter with an overlapping viewer system integrated into a large container called a *desktop* that covers a whole track (Figure 3.11) or a viewer. This allows the use of one or the other windowing model, or both of them concurrently.

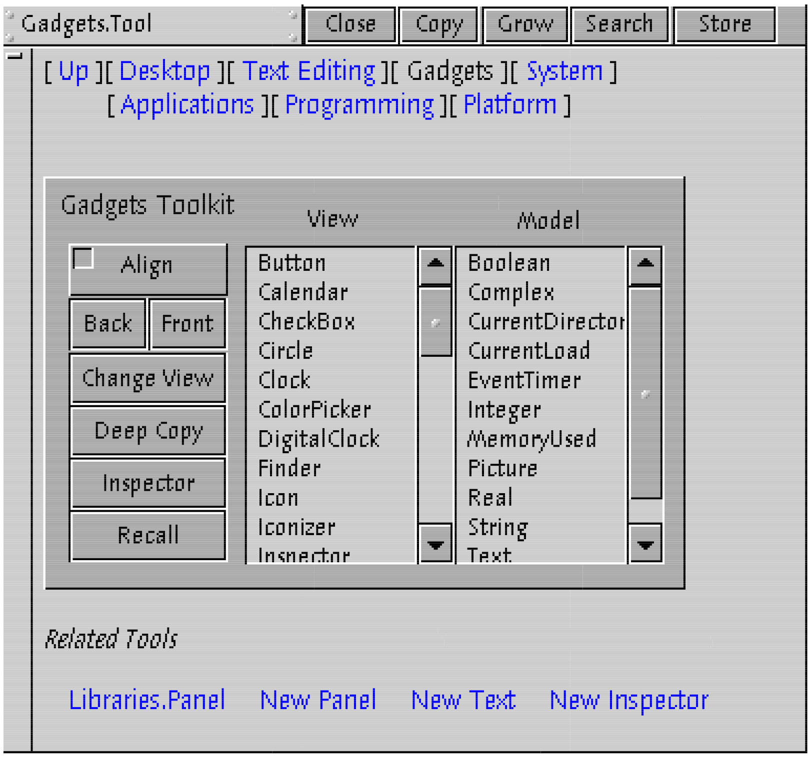

Figure 3.10: A document embedded in a document

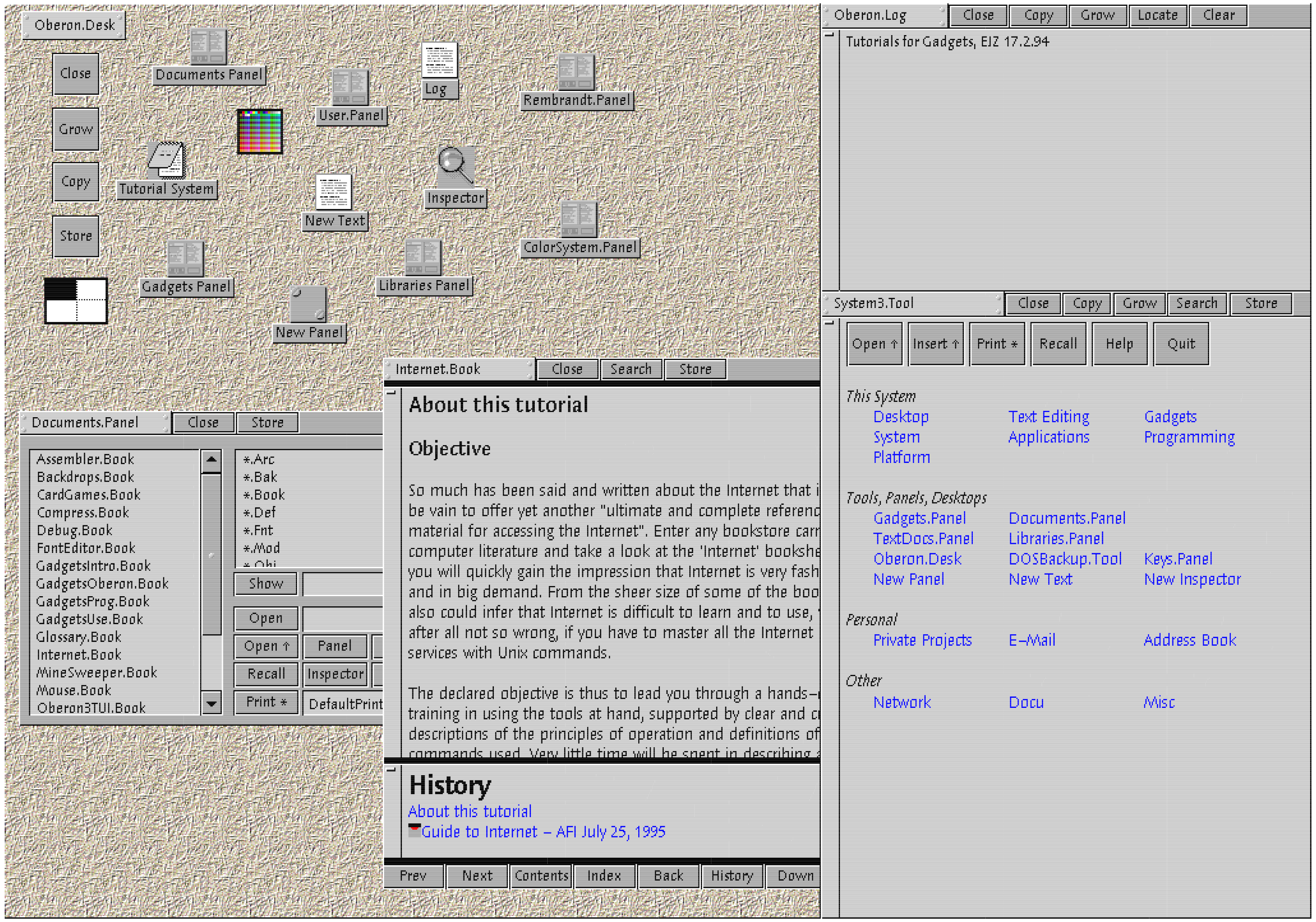

Figure 3.11: A desktop example

Desktops are documents too—the documents placed inside the desktop are embedded in the desktop. A desktop enables you to save the current layout to a file. It is typical to have several desktop documents for different purposes and different display resolutions.

The most popular documents are the panel documents (that contain a panel) and the text documents (that contain a text gadget). The document and its contents, for example the panel document and its panel should not be confused with each other—they are separate components with different purposes. The relationship between viewers, documents, and their contents is explained in Chapter 7.

A document has a name shown in the *nameplate* gadget in the left corner of the menu bar. The document name identifies the document content (and implicitly the document class). When provided with a document name, a central document manager first tries to determine the class of the document involved. Using this information, an empty document gadget instance is created, followed by a request to the document gadget to *fill in* or *load* its contents. Such a design has two useful properties. First, all documents are treated uniformly: there is only one command to open all documents, one command to print a document, and so on. Second, because the initial empty document is requested to load its contents by message, it is free to interpret this event in any way it pleases. The standard document classes delivered with the framework typically interpret the document name as a filename that contains the document contents. Advanced documents can generate document contents and user interfaces on the fly as they are opened.

For example, Figure 3.12 shows a World-Wide Web (WWW) hypertext markup language (HTML) document fetched over the Internet. To open this document, the uniform resource locator (URL) of the HTML page is used as document name. Many internet services like FTP, News, Gopher, and Finger can be accessed directly by specifying an URL as a document name.

There are two possibilities of opening a document. The first way is to have the opened document appear in a new viewer either in the tiling or overlapping viewer system. A second way is to replace the document from where the open command has been activated. Thus depending in which viewer the open command is executed, that viewer contents is replaced by the newly opened document. This feature is used to implement a typical WWW browser-like functionality in the HTML and FTP documents.

The same feature is used to organize the documents of the Oberon system itself. For example, when Gadgets starts, the system track contains a text document with hyper-links showing an overview of the installed system components

Figure 3.12: An HTML document

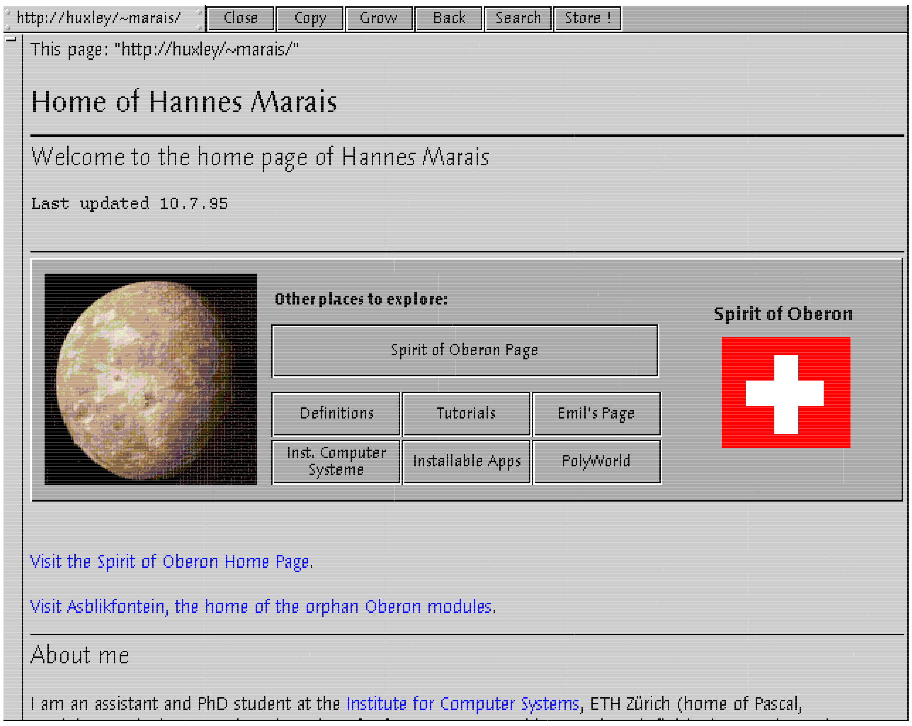

Figure 3.13: A panel embedded in an HTML page

and applications (Figure 3.9). In a similar way as a WWW browser, the user can browse locally through the documents of the system. As documents are addressed in a uniform manner locally and on the Internet, the transition between local and remote browsing is transparent and not noticed by the Oberon user (except for the delay of fetching a document from the Internet).

Some further interesting possibilities are exploited by combining the Oberon HTML browser with the Oberon document model. The HTML browser can also embed documents as inline objects into a standard HTML page. These pages are only fully usable by Oberon clients on the Internet. Figure 3.13 shows the author's WWW home page that contains an embedded panel (which is only visible to Oberon-based WWW browsers). The panel contains buttons to page to other WWW pages—as expected, the panel can be edited and used in place.

3.4 Interactive Composition

In comparison with static user interfaces, user interfaces based on gadgets are both used and edited in place. This section gives an overview of this aspect of a gadget's behavior.

3.4.1 Gadget Interaction

During interaction we distinguish between *using* and *editing* a gadget. A user *uses* a gadget when he or she invokes its primary function. This is for example to click on a button to activate some action, to adjust the knob of a slider, or to enter text into a text entry field. The exact behavior is different for each gadget class. To *edit* a gadget means to change its layout (position and size), embedded container, attributes and links. As mentioned before, the editing of attributes and links is done with standard tools. Layout editing is interactive and involves grabbing the gadget and changing an aspect *directly in place*. For example, to change the position of the gadget, you pick it up and move it to its new location. In a similar manner to drag and drop, the gadget is moved from one container to another (and also between different user interfaces). To change a gadget in size it is grabbed and resized (with feedback). After the change has been made, the display immediately reflects the new situation.

The use–edit distinction. The integration of *use* and *edit* reduces the programmer-user distinction and gives users a smooth path enabling them to incrementally grow toward programming activities. This issue was first presented by Randall Smith [SUC91] after his experience with the *Alternate Reality Kit* [Smi87], a system that attempts to construct familiar real-world metaphors for users. He discusses what he calls the use–mention distinction and different approaches to its implementation.

The possibility of both being able to use and edit a gadget in place at any given moment has the problem of figuring out what an input event from the mouse or keyboard means, i.e., is *use* or *edit* required. At any given moment both possibilities are valid. Systems like NeXTSTEP [GM93] have a switch that allows the user interface designer to switch between the use and editing modes. This is called the *modal approach*. The mouse and keyboard events are interpreted differently depending on the mode selected. In use mode, the user interface can be tested. In edit mode, the dialogue elements can be moved and resized. In delivered software

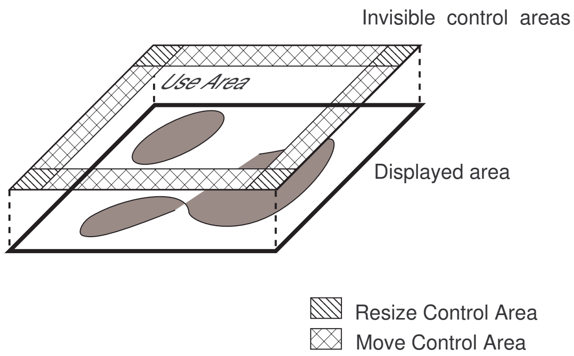

Figure 3.14: The gadget control areas

the edit possibility is not possible, effectively *freezing* the user interface against further modification.

To circumvent introducing a mode, user interface designers often sacrifice screen space for mouse control areas. Windows have a thin border in which the user can resize or move them, mixing using and editing. Depending on where in a control area a mouse event is initiated, a different reaction results; typically, borders are used to move a window around and corners to increase or decrease the size. Graphic editors use control points inside geometric figures instead. The term *affordance* refers to the association of a permanent knob, handle, or other part with an appropriate function [SUC91].

A third possibility to reduce the use–edit distinction is the *tools approach* [SUC91]. Just as tools can be applied in the real-world to change objects, software tools can modify components. The tool approach is popular in Oberon with tool texts and panels operating on documents and objects that are located inside other viewers.

In Gadgets, the way how a gadget is edited depends on the gadget itself (i.e. do not assume the one or the other possibility). This means having the gadgets *edit themselves*. Gadgets have their editors built in, instead of having a special edit window that knows how to edit all possible gadgets. This is correct when we want the system to be extensible and not to force a specific way of doing things to a component. This approach also eliminates much of the editing knowledge embedded in containers by transferring editing to the child itself.

To prevent component programmers from deviating too much from a proposed user interface style and to simply programming, a default edit behavior is provided for gadgets. This involves control areas around the boundary of a gadget for moving and resizing the gadget (Figure 3.14). Control areas are invisible, thus saving screen estate. A drawback is that it takes novices a few minutes to get used to them. The decision to leave the editing behavior to the gadgets themselves turned out valuable for implementing geometric figures based on control points. In this case, the figures override the default gadget edit behavior and replace it with mouse sensitive control points located inside the figure.

To make it possible to edit very small gadgets, a function varies the width of the editing areas depending on the width or height of the gadget. A minimum width of 1 screen pixel and a variable maximum width is used.

Locking. In-place editing has the disadvantage that careless users can destroy a document with only a few mouse clicks. To prevent such mishaps from happening, user interfaces can be *locked*. This is by setting the *Locked* attribute of the container to disable the control areas of its children. A container can be unlocked at any later time with the Inspector.

Division of Responsibility. With gadgets having editors "built-in", the *edit* aspect of user interface interaction is distributed through the system. The implementation can be imagined as follows. As soon as the mouse enters into the area a gadget occupies on the screen, it starts to receive mouse events. It is completely up to the gadget to do whatever it pleases with these events. A decision might be made to change itself in size based on the mouse key pressed and the position of the mouse pointer. Once the decision is made, the gadget itself takes control of the mouse by dragging a rubber-band rectangle indicating the selected size. As soon as the mouse button is released, the gadget sends a request to its parent to update it according to its new size.

A special mechanism allows the parent to determine if a gadget responded to mouse events. If not, the parent can take control of the mouse instead. Also, the parent might prevent the mouse events from arriving at a child at all. An important question arises out of this, namely who is responsible for handling certain events. In effect, a division of responsibility is required; an example illustrates why this is necessary.

It might happen that a gadget does not have enough circumstantial knowledge to edit itself. For example when the user selects several gadgets in a container and wants to move them around as a whole. A single child does not know about the

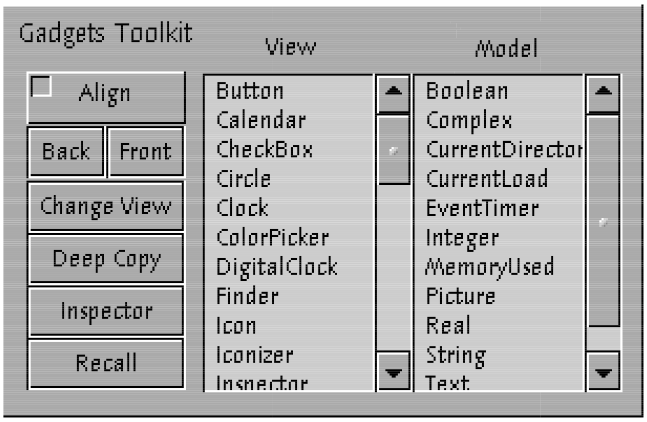

Figure 3.15: The *Gadgets.Panel* document

other selected gadgets due to the encapsulation boundary. In this case, it is clear that group editing operations are the responsibility of the container and not of the children. A first approximation would be for the container not to let the selected child obtain mouse events and directly take control of editing. A more refined way is to have the child defer mouse operations under certain circumstances to the parent. This level of co-operation between parent and child gives the child some additional possibilities for controlling the interaction.

3.4.2 Examples

This section is intended to give a feel for interactive composition—the reader is encouraged to explore further possibilities using the interactive tutorials contained in the Oberon distribution.

The primary tool for composition is the *Gadgets.Panel* (Figure 3.15). The two lists presented side by side enumerate the standard visual and model components. The controls on the left let you to align gadgets in different ways, change overlapping priorities, make deep copies, open the inspector and recall the last deleted gadgets. After opening a fresh instance of a container gadget like a panel, you proceed to insert gadgets into the container by clicking on the gadget you want in the *View* list. The insertion point is set with the caret using the left mouse button in the container. Figure 3.16 shows a partially completed panel containing a caret (a small black cross). You arrange the gadgets freely using the middle mouse button. Text captions are entered by setting the caret and typing on the keyboard.

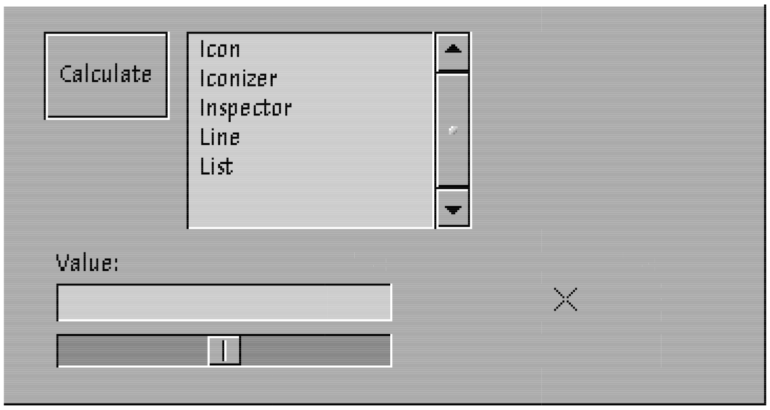

Figure 3.16: Constructing a panel

The attributes of a gadget are shown in the Inspector by *selecting* the gadget with the right mouse key (making it appear in a selected pattern), and then hitting the *Inspector* button. The result—for a button gadget— is shown in Figure 3.17.

By changing the attributes the caption and behavior of the Button can be changed. The figure shows a button with a caption "Open" having the command "Desktops.OpenDoc Rembrandt.Panel". The command "Desktops.OpenDoc" is the universal command for opening documents. In this case the document manager is requested to open a document called "Rembrandt.Panel"—the name of the control panel of the bitmap editor. The document name could just as well have been the URL of an HTML page. Section 3.4.3 shows how to parameterize commands further.

As a further example, Figure 3.18 shows the steps to create a popup menu from a panel and a text gadget. The panel and the text gadget are picked up and dropped into an *iconizer* container that shows the two as the sides of a "flip card". The card is flipped from one child to another by pressing the middle mouse button on the small button located at the top left corner of the iconizer. To move a gadget from one container to another (in this case from the panel into the iconizer), the user picks up the panel with the middle mouse button and drops it with a left interclick into the iconizer. The iconizer then adjusts its size according to that of the panel. The contents of the iconizer can be further edited in place. Figure 3.19 shows the alignment popup menu of the *Gadgets.Panel* that was constructed in exactly this way.

There are numerous examples of composition and assembly in the Gadgets

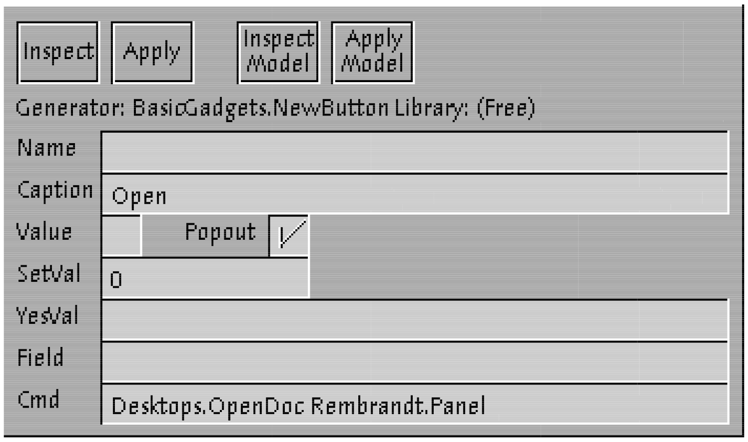

Figure 3.17: Inspector inspecting a button

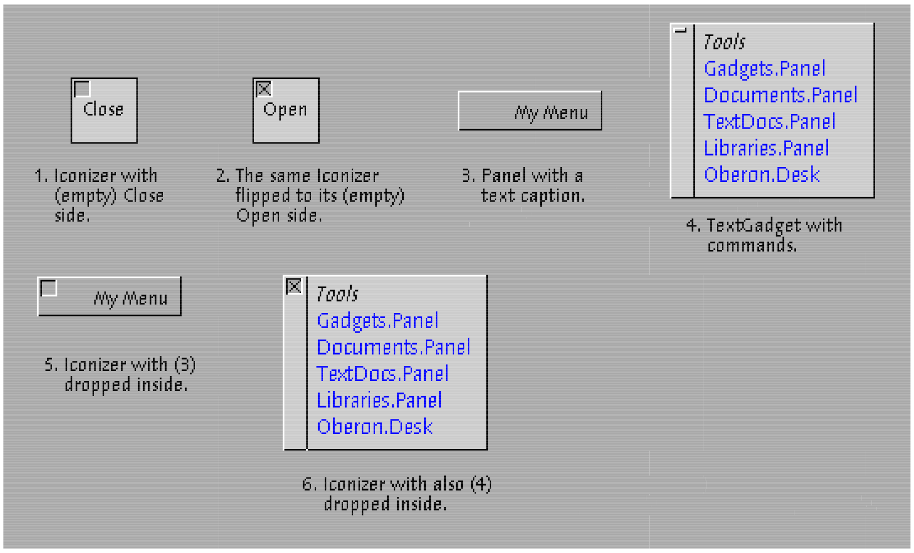

Figure 3.18: Assembling a menu

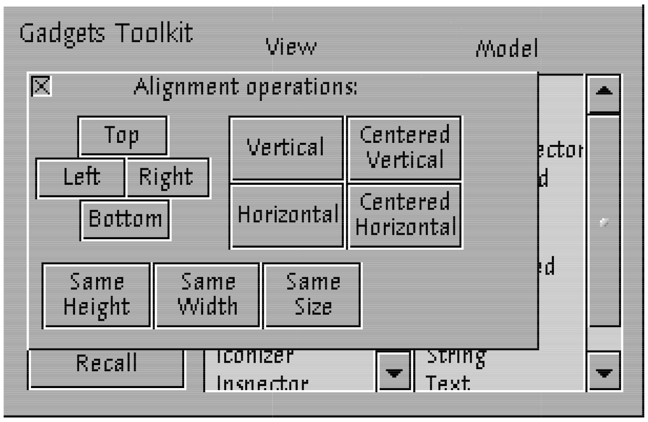

Figure 3.19: The alignment popup menu

system. One further example—which will not be sketched in detail—is to cut out a part of a picture and dropping it into a button to provide it with an iconic representation. Needless to say, any gadget can be used instead of the picture gadget.

Adding models. After inserting gadgets into a container, doing the layout, adjusting attributes and connecting commands, a further possibility is to link model gadgets to the user interface. This is done by selecting all the gadgets that should have the same model and then clicking on a model name in the *Model* list of the *Gadgets.Panel*. If model and view share a common communication protocol, all selected gadgets will update their state according to the current state of the model. The Inspector is used to follow the link from model to view as to edit its attributes.

So far all construction steps have been interactive. In preparation of the steps that will follow and for level-1 programming, it is necessary to give names to the gadgets that will be referred to by macros or modules. This is done by filling out the name attribute of a gadget using the Inspector. As a last step, the constructed user interface is locked and saved in a document.

3.4.3 Commands and Macros

In the Oberon textual user interface (TUI), commands of the form *M.P* read their input from the text immediately written following the command. When a gadget "constructs" similar text parameters as the user manually does in the TUI, the command implementation would not know if it was invoked from the TUI or from the GUI, making these two ways of activating commands interchangeable. This idea is the basis for separating the application code and the user interface.

The proposed implementation is based on macro symbols for the construction of command strings. The idea is that parameters originate either from the user interface where the command was invoked, or from the global system state. The command attribute can contain macro letters—followed by macro parameters—that are substituted with other text pieces before execution. The command attribute—with all macro symbols expanded—is executed in the conventional TUI fashion.

The predefined macro symbols are as follows:

Attribute substitution. The macro *&O.A* is replaced by the value of the attribute *A* of the named gadget *O* in the *context* of the gadget executing the command. We define the context of a gadget as the set of siblings and their descendants (this essentially this means that a sibling or one of its descendants named *O* is picked according to some search strategy). Our search strategy proceeds in a breadth-first manner until the named candidate is found. All other gadgets with the same name are ignored. A shortcut macro # is also provided for the case when the gadget executing the command corresponds to *A*.

Text selection. The $\uparrow$ symbol is substituted with the current text selection.

Gadget selection. The $\uparrow$A macro is substituted with the concatenation of the attribute named A of the set of selected gadgets. The attribute values are separated with a space character.

Initiator substitution. The $!A$ macro is replaced by the attribute A of the *initiator* gadget in a consume operation. The initiator is defined as the gadget that is dropped into the container. We call the container the *recipient*. The initiator macro is used in commands related to drag and drop operations.

The most popular macro symbol is attribute substitution (&). For example, we can now extend the opening example of the previous section with the substitution macro. Figure 3.20 shows a modified panel where the document name is written inside a textfield. The *Value* attribute of the textfield reflects the text seen in the textfield. The button parameterization is achieved by naming the textfield *Input* and setting the *Cmd* attribute of the button to “Desktops.OpenDoc &Input.Value”.

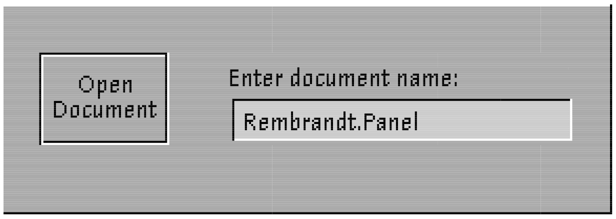

Figure 3.20: A document opening tool

On pressing the button, “&Input.Value” is expanded to whatever is written inside the textfield. The expanded text is then executed in the same manner as when clicked on by the user in a text editor.

The straight-forward mapping of gadget actions into commands might seem simplistic and limited at first glance. In fact, its usefulness is directly related to the set of useful commands and not to the mechanism itself. It does not for example rule out the use of more complicated actions as the invoked command could potentially implement a complete scripting language. The current opinion of the designers is that Oberon is the programming language of choice for programming more complicated gadget behaviors (as commands). Several commands related to setting and retrieving attributes, enumerating directories, and managing public libraries are part of the Gadgets distribution, and each application developed with Gadgets adds to the list. In the author’s opinion, the biggest problem with this approach is that the authors of these applications tend to make commands too application specific so that they often cannot be reused.

3.4.4 Summary

With the machinery sketched so far in this chapter it is possible to interactively build graphical user interfaces in restricted domains. The mileage you get vary on the gadget classes and prefabricated commands available in that application domain. To be really useful it for example requires a certain level of factorization of components and commands so that they can be mixed and matched. In this regard, the Gadgets system only contains a limited set of commands that are applicable to gadgets in general. Also, the container classes are of a general nature that can be applied to numerous problems. Consequently, the system is only an enabling technology for componentware in more specific domains.

I believe that even if the number of component classes and commands are dramatically increased, it is not certain that completely interactive application composition is possible. A certain amount of "glue" programming often remains necessary. This type of programming is currently mainly done in the Oberon language, which is a too low-level abstraction for end-users. It seems that at user interface level, efficiency and typing plays a lesser role, thus opening up the possibility for using scripting languages. The current design of the Gadgets framework does not rule out the use of scripting languages for user interface programming—it certainly is an interesting area for further system development.

This chapter has mainly concentrated on the interactive aspects of the system. In the following chapter we start exploring the structure of the system, followed in the remaining chapters by technical implementation details.

Chapter 4

Overview of Design Concepts

According to Booch, the canonical form of a complex system is a hierarchy [Boo94]. Most interesting systems embody many hierarchies—Booch identifies two of them, namely the *part of* or *object* hierarchy and the *is a* or *class* hierarchy—the understanding of which can vastly simplify a complex system. As the Oberon language [Wir88b], in which the Gadgets system is implemented, is modular, we identify at least three hierarchies, namely the module hierarchy, the type hierarchy, and the object hierarchy. With messages as first-class citizens in Oberon, a further decomposition of the type hierarchy into the object type and message type hierarchies is possible. Also, the component model of Oberon divides the object hierarchy into two further hierarchies called the *display hierarchy* or *display space* and the *library hierarchy* or *persistence hierarchy*. The goal of this chapter is to introduce these hierarchies as used in the Oberon and Gadgets systems.

4.1 The Module Hierarchy

The *module hierarchy* is the structuring mechanism for code reuse in the Oberon system. Modules *export* a public interface to their clients and *import* other modules to make use of their features. The *explicit* dependency relationship between modules creates a hierarchy in the form of a directed a-cyclic graph. *Implicit* dependencies between modules exist when modules assume a common feature—for example the value of a constant or units in a coordinate system—that is not explicitly defined by an interface. Modules are used to implement procedure libraries, abstract data types, and component implementations. In the case of the Gadgets, a module typically implements one or more gadgets.

Independently extensible software is modular out of necessity; separate compilation of modules where adding a new compilation unit does not require recompilation of other compilation units is a must. A way to dynamically integrate extension modules at run-time is also required, as for example used in *dynamic link libraries* or *DLL's*. Separate compilation combined with late-binding enables previously loaded modules to invoke code that is loaded later.

The module as vehicle for code reuse is not commonly accepted. In languages like C++ [Str87] and Smalltalk [GR93] the concept of a class is merged with the concept of a module so that object-orientation comes to play even if no intention is present to exploit its advantages. The importance of a modular structure has recently been emphasized with the development of name-spaces for C++. Even the language Beta in which everything is expressed as a *pattern* has a higher abstraction mechanism called a *fragment* that corresponds to a compilation unit [MMPN93].

4.1.1 Module Interfaces

As the interface of a module (or a component) presents a *contract* to its clients, a change made to an interface can invalidate clients of that module. The larger the number of direct and indirect clients, the greater the disruption. This *ripple effect* has serious repercussions in open systems where the system is independently extended. The simplest solution is to freeze the interface of a module as soon as it is made available to a larger audience and to restrict further changes to the implementation only. It is a common approach that has proved its value with commercial DLLs. There are however techniques that leverage additional flexibility even in the face of interface changes, and so allow the system to evolve. These techniques were especially helpful during the development of the Gadgets system.

Symbol file technology. A clever module loader and finer-grained finger printing of module interfaces enable interface *extensions* to be made without invalidating clients. This is based on the observation that a pure addition cannot be used by any clients yet. The *object model* of symbol files increases the capability for evolution and has been built into some Oberon implementations [Cre94]. It also allows exported elements to be removed from a module at the risk of invalidating those clients that depended on them. This is a dangerous operation as eventual run-time consistency cannot be checked statically without having access to all components.

Narrow Interfaces. The number of possible client dependencies can be decreased by reducing the size of a module interface—such an interface is called narrow. An *open message interface* is an example of a narrow interface (see section 4.2.1 for more details). It is not visible from the interface what messages a component understands and so objects can be upgraded with additional functionality without the clients noticing. Although it is a practical feature, it is not recommended as implicit dependencies are often introduced, for example, in the form of assuming that a component understands a particular message. So without appropriate documentation or browsing tools that examine implementations, it is not known if an object understands a message or not.

Bottleneck modules. It is common to factor out mutual dependencies between two or more modules into a separate bottleneck module. As long as the bottleneck's interface is not changed, the clients that depend on it can be modified without invalidating each other. Oberon uses this technique by defining message protocols in bottleneck modules. In ETHOS [Szy92b] bottleneck modules are used to define *directory objects* that generate instances of often used objects.

4.1.2 User Interface and Application Coupling

Decoupling the application code from the user interface separates concerns and makes independent modification of application and user interface possible. Its successful application is dependent on the modular decomposition of the application.

The connection between user interface components and the application is a bidirectional channel. During direct manipulation data is transferred to or actions invoked in the application. In the opposite direction, the results of a computation are transferred from the application to the user interface elements for presentation. There are several means to obtain the connection between interface and application, which in turn affect their coupling.

First we list common approaches found in literature for connecting user interfaces and code. *Callbacks* involve the registration of procedures with objects, specifying under which conditions the object should call the procedure. The MVC paradigm [KP88] allows the registration of view update procedures at the model. The X Toolkit registers callbacks at the widgets (user interface objects) which are called when a specific event occurs [NO90]. *Active values* registers functions with variables which are called when the value of the variable changes. *Constraint systems* can be used to ensure that variables maintain certain relations

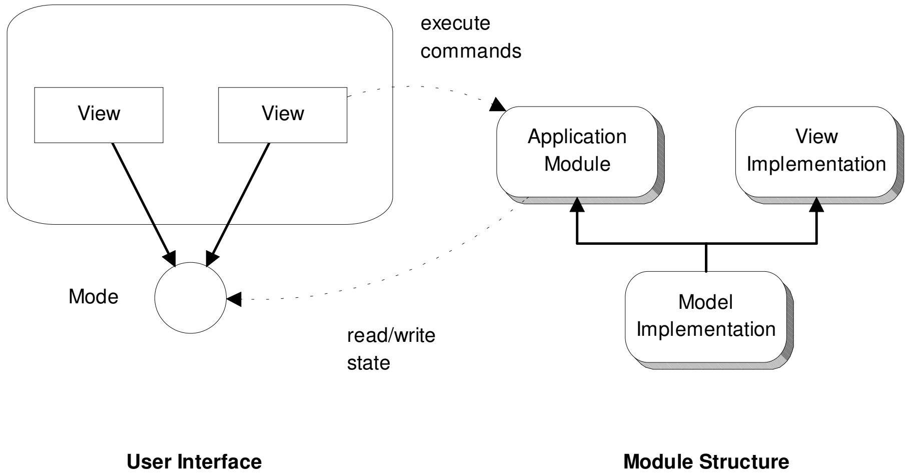

Figure 4.1: User interface and application coupling

[MGD+90, Hil92]. All these approaches need a certain level of support from application objects and create UI and application dependencies.

In the Gadget system, the connection between the user interface and the application are the model gadgets (cf. 3.3.1) and Oberon commands. Actions invoked from the user interface are realized as commands (cf. 3.4.2). To encourage reuse, views and models in the Gadgets framework are realized as separate modules and the application dependent code is located third module, as shown in Figure 4.1.

The issue of user interface and application coupling involves the question how the user interface (both model and view) knows about the application module, and how the link from the application module to the user interface is maintained. The idea is as follows. The user interface component that executes a command sets a global variable to itself. This variable, called the *executor*, keeps track which part of the user interface the command originated from. The application retrieves references to UI components in the latter context by requesting a location service to locate components by name (cf. 5.3.4). Once located, the state of UI components (models or views) are *read*, the calculation performed, and the results *written* back to the UI component. If a model's value has changed, a change notification mechanism is initiated.

In the remainder of this section we will review how different programming styles can affect component coupling.

User interface dependencies. There are several possibilities for exchanging data between the application and the UI, each of which illustrating a different level of application module and UI coupling.

A *component dependency* is present when the application-specific code is merged with the user interface component. An example is to make a subclass of an existing component—like a button—and hard-code its behavior to something application-specific.

A *view dependency* is present when the application module makes assumptions about the type of a view component in the UI. It is a strong coupling that leads to failure when either the view is exchanged with a different type, or when the view interface changes.

A *model dependency* is present when the application module makes assumptions about the type of a model component in the UI. It is a medium-strong coupling that leads to failure when either the model is exchanged with a different type (for example an INTEGER with a REAL), or when the model interface changes. In contrast with a view dependency, a model dependency leaves freedom in selecting different views.

A *name dependency* is present when the application module assumes that a component with a specific name is present in the user interface, thus leading to failure when the component is deleted or renamed. A dilemma arises when different commands have to share access to the same component but make different assumptions about its name. A solution is to pass the names of referred components in the command parameters. The application then parses the parameters to find out the names of components before locating them in the user interface. Name dependency only becomes a problem when commands are heavily factored.

A *protocol dependency* is present when the application module makes an assumption about the message protocol a user interface component understands (in contrast to the model and view dependencies that make assumptions about a type). A protocol dependency allows the exchange of a component with another as long as it understands the same protocol. A protocol dependency is preferred when components are manipulated directly.

A *parameter dependency* is present when the application module makes assumptions about the syntax of (text) parameters passed to it. Most Oberon commands exhibit this dependency. Typically, a parameter syntax error is detected and reported. This type of dependency is to be strived for as the application makes no assumptions about the user interface at all. The drawback is that it reduces the communications channel between UI and application to a one way channel—

results can not be presented directly by UI components. There is however scope for improvement in this area: it is easy to imagine a piping mechanism as in Unix [Bac86], where the result of a command (a text) is piped into another component. This would require a more fine-grained decomposition of many of the existing TUI commands.

The location service involves a searching mechanism that can be a performance bottleneck when many components must be repeatedly located. A *single instance dependency* is present when the application optimizes the search operation by remembering component references across command invocations in global variables. This style of programming is not compatible with the Oberon design. The wrong assumption is that each application user interface has a set of separate modules. This assumption is invalidated when user interfaces—a simple collection of gadgets—are cloned on the display. The analogue would be to start the same application more than once on a process-based system. As modules can be loaded once only, the same application modules have to share many copies of the same user interface. It is thus imperative that an application search for components *each time* a command is executed. It is possible to reduce the search operation to a single one by creating a named *directory component* that has references to all often used components in a user interface instance (a model dependency).

The dependencies introduced in the previous paragraphs are guidelines for decoupling user interface and application code. A clean decoupling might cost more work initially, be less efficient but it allows for easier maintenance, modification and reuse, whereas a strong coupling does the opposite. We leave it to the designer which one or mixture of these dependencies are preferred.

4.1.3 Model-View Coupling

A particularly difficult coupling to weaken is that between model and view gadgets. It is for example tempting to simplify interfacing by having views "know" about the model they represent. This design pattern is extensible in the dimension of the views only. When new models are linked to existing views, the latter have to be updated also to "know" about the new model class. This situation appears surprisingly often as models are good candidates for extension. This is referred to as the *cartesian product problem* of composing components, which hints at the fact that all possible combinations of models and views must conceptually be supported.

In principle, the composition problem is solved by connecting models and views through a data-exchange protocol. The definition of an extra protocol mod-

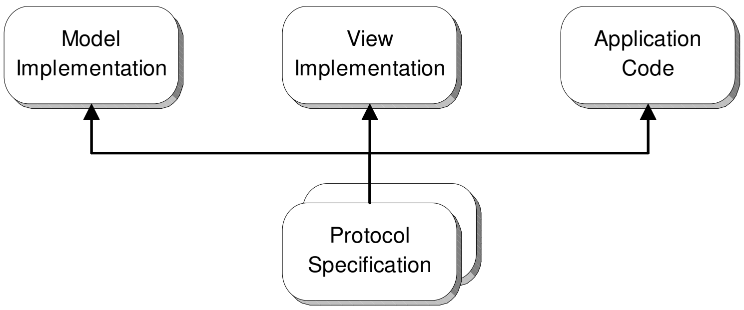

Figure 4.2: The extensible MVC module decomposition

ule leads to a decomposition that is extensible (Figure 4.2).

The remaining dependency is on the protocol itself, which should preferably be extensible too. First, as sketched in section 4.2.1, the protocol messages can be extended, and the models and views updated to react accordingly on the new types. Second, new protocols are definable at any time, and also incorporated into components without invalidating any existing clients.

The result of the MVC decomposition is that several orthogonal possibilities for extension in the dimension of the model, view, protocol, and application exist. The rationale for implementing a separate protocol module is the largest when the protocol is of general use, but it is surprisingly difficult to find and implement such general protocols. Currently, the attribute message (cf. 5.3.1) forms such a protocol for the exchange of simple data types.

4.1.4 Examples

The Oberon module hierarchy is split into two parts, namely modules of the Oberon TUI and the Gadgets GUI. The TUI is a minimal Oberon system with a compact set of modules geared towards working with text as a user interface medium. We shall refer to this part as the *Oberon base system*. The Gadgets GUI builds a component architecture on top of this basis.

The base system. The module hierarchy of the base system (Figure 4.3) is divided into the inner core, the outer core, and the applications.

The inner core is responsible for memory management and garbage collection (*Kernel*), file directories and files management (*FileDir* and *Files*), and the

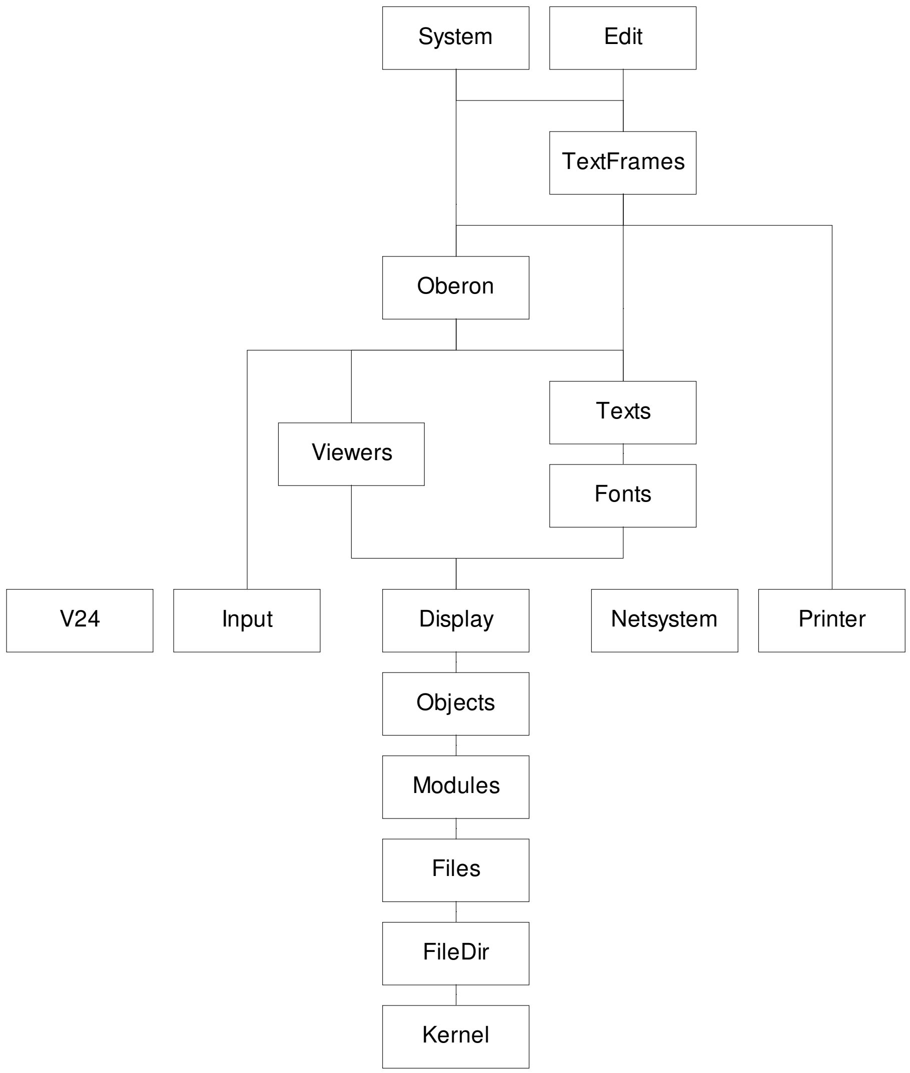

Figure 4.3: The base module hierarchy

dynamic loading of modules (*Modules*).

On top of the inner core we find the outer core. Its most prominent module *Objects* defines the root of the object types in the system and provides a library manager (cf. 5.4). As part of the Objects module we also find the *object message protocol*, which is extensively discussed in Chapter 5. Above the Objects module comes the driver layer. It contains drivers for attached devices like the mouse and keyboard (*Input*), display (*Display*), printer (*Printer*), network (*NetSystem*) and serial ports (*V24*). As explained in section 3.2, the Oberon display is covered with *viewers*; the module *Viewers* provides their basic functionality. The reliance on text in the basic system results in a small subsystem for managing fonts (*Fonts*) and defining an abstract data type for texts (*Texts*). The module *Oberon* brings all the modules below it together as the distributor of mouse and keyboard events to the viewers on the display, the passing of textual parameters to commands, and the handling of the mouse cursor. In the outer core, modules Objects, Display Texts, and Oberon play important roles as bottleneck modules that define the important message interfaces of components.

The outer core is essentially completed with the module *System* that manages system-wide tasks like copying and deleting files, setting the time, handling exceptions, etc. It makes use of a module *TextFrames* (part of the applications) that provides a rudimentary text editor for editing modules, writing texts, and showing results of commands. A module called *Edit* provides features for storing and loading text files, setting fonts, and other editing operations. Not shown in the module hierarchy are applications like the compiler for compiling modules and a set of mostly programmer-oriented tool modules like browsers and so on.

Note the base system module hierarchy of System 3 is approximately that of the original Ceres Oberon implementation [WG92] except for module *Objects* that ties up many of the types defined in the system in a type *Objects.Object*. This module is the innovation in Oberon System 3 in comparison with earlier versions that did not have the concept of first class objects. A detailed description of this module is found in Chapter 5.

The Gadgets system. The Gadgets system module hierarchy is divided into the Gadgets core, the gadget component collection, and the document collection (Figure 4.4). The Gadgets core extends the base system with additional functionality as required by the gadgets. The gadget component collection is a loose collection of modules that implement the gadget components. The module structure of the gadget component collection is rather flat as gadgets seldom import each

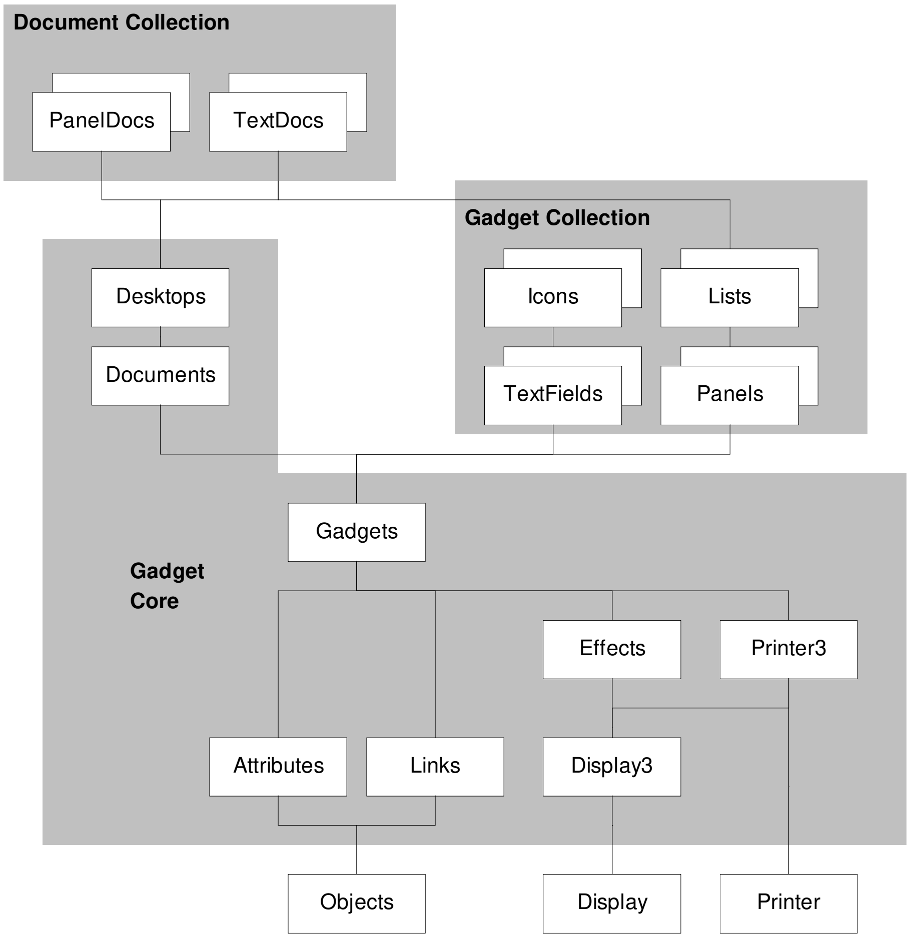

Figure 4.4: The Gadgets module hierarchy

other directly but rather communicate using message protocols defined lower in the module hierarchy. The document collection provides "application like" functionality that is associated with the different document classes.

The Gadgets core consists of modules that provide arbitrary clipping regions for the display and printer driver (*Display3* and *Printer3*), handling of attributes (*Attributes*) and of links (*Links*), special effects and non-portable code (*Effects*), and definition of gadget types, common message handlers and other functionality (*Gadgets*). The gadget component collection is a large set of modules implementing about 40 or so gadgets ranging from simple models like *integers* and *strings* to complicated containers like *panels* and *text gadgets*. To prevent a large module inflation several gadget classes are sometimes collected in a single module. The module *BasicGadgets* for example contains the implementations of the commonly used model gadgets, buttons, sliders and check boxes. Module *TextFields* contains the implementation of string entry fields and text captions used for decoration.

The document collection contains implementations of the document classes. If a component class, say X, is to be elevated to the status of a document, a module called "XDocs" provides the basic functionality for that document. This includes a definition of a document class, specification of the document iconic representation, the look of the document menu bar, how the document stores itself to disk, and common commands that can be applied to a document of that type.

4.2 The Type Hierarchies

The type hierarchy is the secondary structuring mechanism after the module hierarchy in the Oberon system. It is formed by a language mechanism called *type-extension*, which is the foundation of object-oriented programming in Oberon.

4.2.1 Type Extension

The following program fragment shows how type extension is typically applied:

```
TYPE
  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD
    ... object fields ...
  END;

  SpecialObject = POINTER TO SpecialObjectDesc;
  SpecialObjectDesc = RECORD (ObjectDesc)
    ... additional fields ...
  END;
```

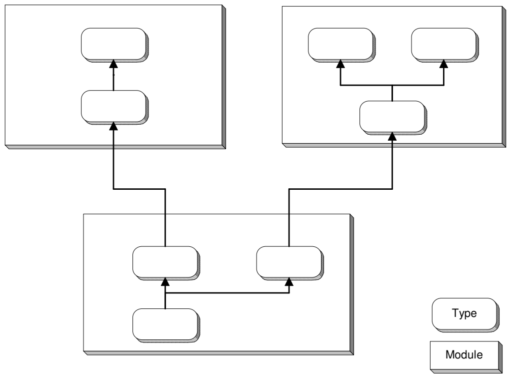

Figure 4.5: The type hierarchy embedded in modules

In this example, the type *SpecialObjectDesc* is a type extension of the type *ObjectDesc*. An extended type (or derived type) inherits all the record fields of the type it extends (the base type) and can add its own fields. Variables of an extended type are assignment compatible with variables of a base type (i.e. inclusion polymorphism [CW85]). A type can only be derived from a single base type. There is no distinguished root type in the Oberon system or language from which all RECORDs must be derived from, hence, the system consists of disjunct type hierarchies. By defining a pointer type based on a RECORD type, it is possible to allocate RECORDs of that type dynamically on a heap using the NEW operator.

Types are associated with modules and can be extended across module boundaries (Figure 4.5). An important feature is that a derived type can be defined without having the source code of the module implementing the base type.

Late binding. Late binding is obtained by declaring procedure variables in a RECORD type and assigning them after instantiation of the RECORD to actual procedures. Sending a message corresponds to calling an assigned procedure. The message contents is the actual parameters of the procedure, one of which is often a *self* parameter. As RECORD instances of the same type can have different assigned procedures, this is an instance *centered-approach* [US87, Lie86] as opposed to a *class-centered* approach. An example of a typical type definition is shown below.

```
TYPE
  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD
```

```
    ... object fields
    method0: PROCEDURE (self: Object; ... parameters );
    method1: PROCEDURE (self: Object; ... parameters );
  END;
```

Notice that *delegation* [Lie86, US87] can be simulated by passing a *self* parameter different from RECORD where the procedure variable is invoked from. When using procedure variables to implemented late binding, code inheritance must be programmed out by hand—there is no direct support for it in the Oberon language. Code-inheritance is simulated by calling procedures—typically attached to a base type—from an assigned procedure. When many RECORD instances share the same set of assigned procedures, a shared *method block* of procedure variables can be attached to each RECORD instance to save memory (cf. 6.5.3).

Open message interfaces. An extensible object interface is obtained when instead of normal parameters, a RECORD is passed as message to an object. This makes messages explicit and also separates the message protocol from the object itself. It also has the interesting feature of allowing both object types and message types to be extended in independent dimensions. As an example, the following type definitions can form the roots of an object and message type hierarchy:

```
TYPE
  Msg = RECORD END;

  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD
    ... object fields
    handle: PROCEDURE (self: Object; VAR M: Msg);
  END;
```

An example of a concrete message is a request for an object to copy itself. For the purpose of this illustration, we can define the *CopyMsg* as a type extension of the type *Msg*, and implement a *message handler* that can be assigned to *handle*:

```
TYPE
  CopyMsg = RECORD (Msg) copy: Object END

PROCEDURE MessageHandler(self: Object; VAR M: Msg);
BEGIN
  IF M IS CopyMsg THEN
    WITH M: CopyMsg DO (* open up access to fields *)
      ... make copy of self
      ... assign copy to M.copy
```

```
    END
  ELSE
    ... look for other messages
  END
END MessageHandler;
```

The message handler exploits the Oberon type system to discriminate on messages. A run-time type test of the form M IS T, where M is a RECORD and T a type, distinguishes between message types passed to an assigned procedure. The technique of passing messages as RECORDs is called *open message interfaces*, and is extensively used in Oberon because of its important advantages. Messages can be manipulated like any data structure, and as the message handling is explicitly programmed out, there are few restrictions in the message dispatching structure. This allows the implementation of broadcasting, message forwarding and delegation, and value driven message dispatch. The Oberon system makes extensive use of message extension, message broadcasting and message forwarding.

A useful aspect of open message interfaces is that objects can be extended with additional behavior without changing their types. The change needed for an object to process additional messages is localized in the message handler of an object and is not visible in the interface of the object. This gives greater freedom in changing component implementations without the need to recompile clients. A drawback is that the knowledge of what message protocols an object understands is not made explicit in its interface. This requires the programmer to check for message protocol compatibility at run-time instead of having the compiler do the checks statically. In principle this must be done to ensure that no implicit dependencies on functionality are introduced between caller and callee.

Handler variations. The task of message handlers is to discriminate messages received and execute the appropriate action (or method). In Oberon the discrimination process is explicitly stated, while in languages like Oberon-2 [Mös93], C++ [Str87], and Objective-C [CN91] it is implemented behind the scenes by the compiler and the run-time system. Of these languages, Objective-C has the most dynamic implementation of message passing, while in the others static lookup method tables are used. Objective-C generates a unique identification for each method signature in the system—similar to a type tag in Oberon—which is used to find the appropriate method in a hidden table of (Signature, PROCEDURE) entries. The searching mechanism is shared by all classes. At first, it seems that a similar mechanism can be used to hide the messaging mechanism in Oberon and optimize message discrimination at the same time. There are however two features of the messaging mechanism that cannot be satisfactorily solved in this way.

First, the message type hierarchy is used to optimize message discrimination. The following code fragment shows how a handler can first discriminate on the level of the protocol family followed by discrimination on a finer level. It assumes that type *Family1Msg* and *Family2Msg* are the base types of further message family variants.

```
PROCEDURE Handler(obj: Object; VAR M: Msg);
BEGIN
   IF M IS Family1Msg THEN
      IF M IS Family1Variant1Msg THEN ...
      ELSIF M IS Family1Variant2Msg THEN ...
      END
   ELSIF M IS Family2Msg THEN
      IF M IS Family2Variant1Msg THEN ...
      ELSIF M IS Family2Variant2Msg THEN ...
      END
   END
END Handler;
```

Second, while languages like Oberon-2 and Objective-C select methods on the receiver and message type, Oberon can discriminate in *a generic way* on the contents of messages (i.e., value discrimination). This is for example possible by generically handling messages of a certain protocol. In the following section on the *display space*, message discrimination is influenced by the message contents. The following example illustrates the process.

```
PROCEDURE Handler(obj: Object; VAR M: Msg);
BEGIN
   IF M.field = X THEN
      IF M IS Type1Msg THEN ...
      ELSIF M IS Type2Msg THEN ...
      END
   ELSE
      ... do something else
   END
END Handler;
```

More specialized techniques can be devised. A *switch* can forward messages to one or the other object depending on its setting. A *replicator* can broadcast all messages received—irrespective of message type—to a collection of objects.

4.2.2 Type Safety

Systems that are extended by independent parties tend to suffer in robustness as each additional component is another potential source of error. Major issues are how errors can be localized and eliminated, and how to prevent components interfering with each other.

The Oberon approach is to do as much checking for problems as is possible at compile time. The language is designed so that the type system cannot be circumvented without explicitly indicating the need to do so. Unsafe and non-portable implementations are obtained when low-level operations from module SYSTEM are used. At run-time appropriate type checks and checks for exceptional conditions are also completed. Should an exception occur at run-time, it is detected immediately and does not have lingering effects. The combination of static and dynamic checking is called *type safety*. In the author's opinion, type safety is one of the most important requirements for ensuring that an extensible system scales well.

As an added benefit of type safety and having more type information available at run-time system, we can use a garbage collector and gain the possibility to do metaprogramming [Tem94]. In fact, due to the lack of global knowledge in an extensible system, a garbage collector is the only instance that can reliably free unreferenced heap objects (and is thus an integral part of the Oberon system).

4.2.3 Type Definitions

The Oberon type hierarchy is divided into two disjunct parts: the type hierarchy of dynamically allocated objects and the type hierarchy of the stack allocated messages.

Although the earlier Oberon systems had a hierarchy of message types rooted in a single type [WG92], these systems were limited by the lack of a shared object base type to integrate different sub-systems with each other. The *Write text editor* for Oberon [Szy92a] partially addressed the problem by introducing a common root for user interface *elements* that are embeddable only in text streams. ETHOS [Szy92b], another operating system based on ideas of the early Oberon system, and Oberon System 3 tackled the problem at its root by defining a root type for *objects*.

Just as common features of objects are factorized in a type hierarchy, the common features of messages are factorized in the message hierarchy rooted in type *ObjMsg*. Different protocol families are extended from this base type. As an ex-

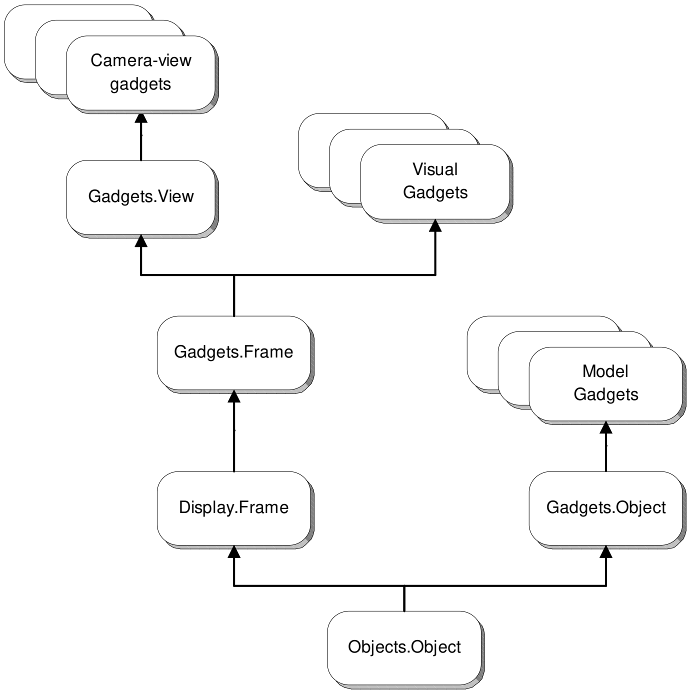

Figure 4.6: The object type hierarchy

ample, the messages sent to visual gadgets have common fields that are collected in a message type called *FrameMsg*. Extensions of the latter type form a family called the *frame messages*. Also, the message type *UpdateMsg*, based on type *FrameMsg*, is the base of the change notification messages (cf. 6.3.8).

The ability to group messages in families and the ability for objects to process messages from different families is similar in concept to that of interfaces in Java [Mic95] and Objective-C [NeX92]. An interface is a collection of methods that an object class has to implement if it admits to implement that interface. Objects can implement several interfaces, making it similar in concept to multiple inheritance and mixins. Not to overload the concept of an interface in Oberon, these interfaces are called *message protocols*.

Now we are in a position to combine the module and two type hierarchies to illustrate the static system structure. The Oberon object type hierarchy (Figure 4.6) is rooted by a base type *Objects.Object*. The derived type *Display.Frame* is the basis of the visual components. The types *Objects.Object* and *Display.Frame*

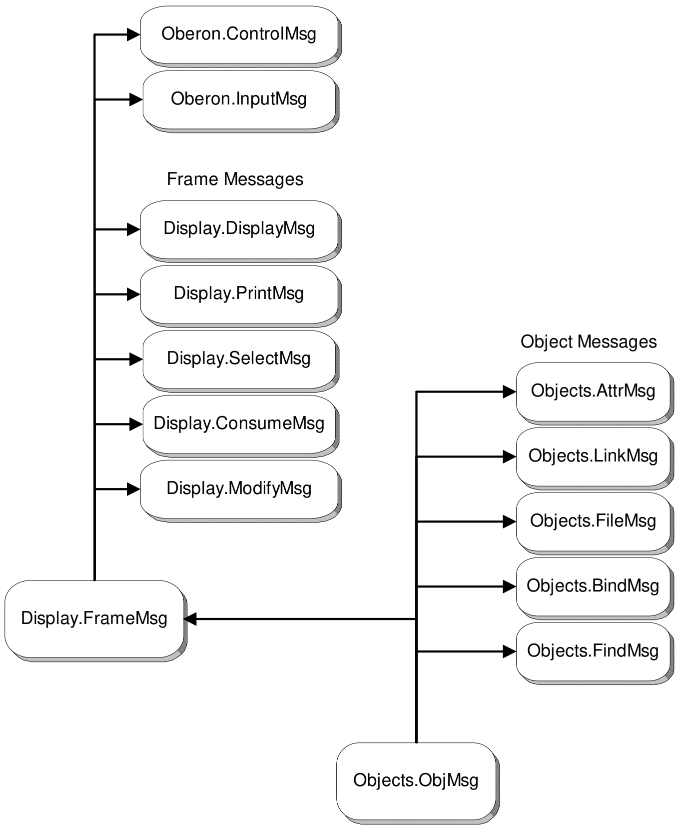

Figure 4.7: The message type hierarchy

belong to the base system and provide rudimentary support for programming components. The Gadgets framework extends these types with *Gadgets.Object* and *Gadgets.Frame* to provide a base type for models and visual gadgets respectively. In contrast to the types *Objects.Object* and *Display.Frame* that do not have standard implementations, these types provide *default handlers* to implement common behavior. The type *Gadgets.View* forms the base type of the camera-view gadget class.

The message type hierarchy is rooted in the type *Objects.ObjMsg* (Figure 4.7). A family of messages called the *object messages* is derived from the latter. The object messages are primarily concerned with features like attributes, links, and persistency (cf. 5.3). *Display.FrameMsg* is the base type of messages that are related to frames and their visual aspects. We will be discussing the object and frame messages in more detail in chapters 5 and 6.

4.3 The Display Hierarchy

4.3.1 Structure

The run-time structure of objects in a system is referred to as the *object hierarchy* [Boo94]. Amongst the object hierarchies of the Oberon system, the *display hierarchy* or *display space* plays a special role as it is visible to the user on the computer display. It consists of visual objects nested in each other, anchored by a top-most container called the *display root* that covers the whole display. The display root contains the system and user track, which contain the viewers, which contain the menus and documents, and so on.

The presence of a visual object in the display space is not a-priori; visual objects that are not located in the display space are said to be *off-screen*. Once inserted into the display space, visual objects are connected to other visual objects by one or more uni-directional *references*.

References are classified according to the role they play in organizing the display space. *Part-of references* are those that collect the children of a container together. *Camera-view references* connect camera-view gadgets with the visual model they display. *Model references* connect view gadgets with their model gadget. *Special purpose references* connect gadgets across containment boundaries. A subset of these references are accessible to the user in the form of named *links*.

Note that model and camera-view references "tie" together parts of the tree-like display space at models and visual gadgets, turning the display space structure into a *graph* (Figure 4.8). An unrestricted graph structure might however lead to a container having itself as descendant, or a camera-view taking a look at itself, situations that lead to infinite recursion trying to display them. Consequently, the part-of and camera-view references must be restricted to an a-cyclic graph (a DAG). In the following section we will see how we can guarantee this invariant.

4.3.2 Messages

In this section we investigate how messages and the display space interact. The largest part of the section involves a discussion how messages travel through the display space.

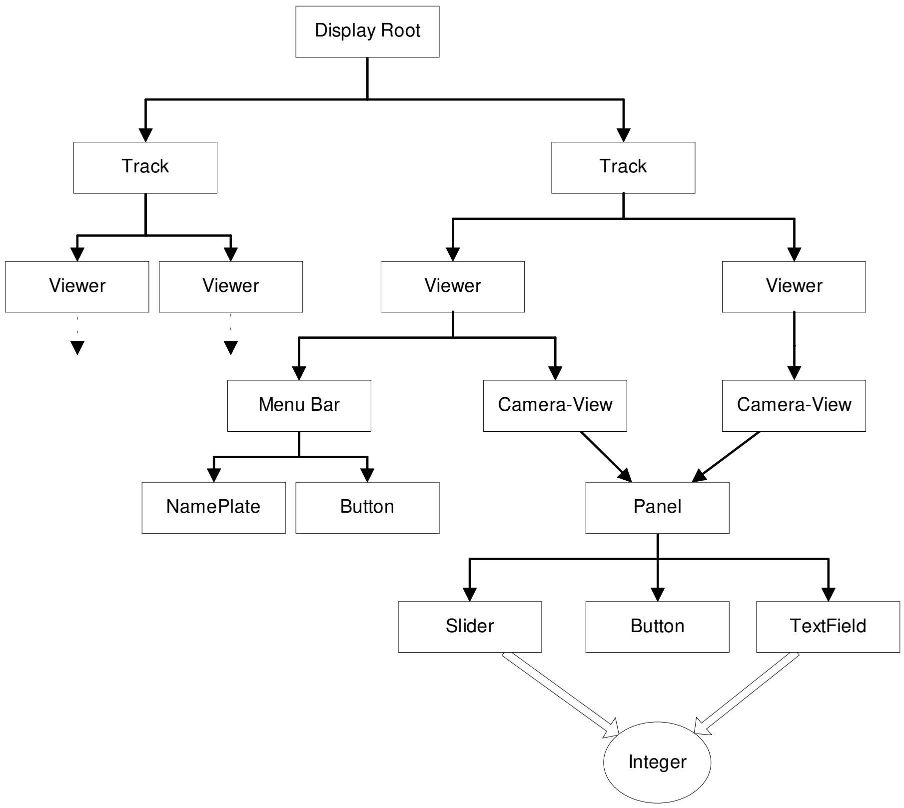

Figure 4.8: The display space structure

Message broadcasting. The small world principle enforces the encapsulation of containers by hiding their children from the outside world and the parental control principle makes the container responsible for their children (cf. 3.2). We can only enforce these principles when communication with a child (and its descendants) is only through its parent. As each visual object in the display space has a parent (except for the display root), we conclude that all communication must take place indirectly through the display root. The display root passes the message to its children, which in turn have to pass it along to their children and so on. The process of passing messages from one gadget to another is called *forwarding*. Forwarding makes messages seem to "trickle down" through gadgets in the display space. During forwarding we have to make sure that messages only flow along the part-of and camera-view references that form the DAG, otherwise we might encounter cases of infinite forwarding cycles (and thus eventually run out of stack space).

Each container in the display space receiving a forwarded message has to make a decision as to what direction the message is to be forwarded. The forwarding can either be directed to a specific child or to all children. A forwarding strategy that visits all the descendants of a container is called a *broadcast*. An example of a broadcast is the *model update* change notification that reaches all visual gadgets in the display space.

The principle advantage of using broadcasts for change notification in the MVC framework [KP88] is that no explicit dependency lists [Kno89] of views need to be managed. Dependency lists are sets of view objects that need to be notified when the state of a model has changed. These lists are typically problematic to maintain and have the unpleasant property of making models "know" something about their views. Using broadcasting there is a separation between model and the view, the view knowing about the model (as it should be), but the model not knowing about its views. Although a simplification is made by the elimination of dependency lists, the cost of change notification is increased. Even though broadcasts are relatively cheap in Oberon due to efficient forwarding mechanisms, they are still not for free (see the following paragraphs).

In addition to broadcasting, it is often necessary to inform a specific gadget of an event. We run the danger of braking our stated principle of parental control should we send the event notification directly to that gadget. We solve the problem by broadcasting these event notices into the display space and including in the event the *destination* object for which a broadcasted message is intended. We thus distinguish between a *directed broadcast* (to a specific component in the display space) and a *true broadcast* (that reaches everything in the display space).

Note that according to the small world principle it is not possible for a container to know where to send a message if it cannot find the destination amongst its children. It might be that one of its children is a container that contains the destination gadget. In such a case, a container forwards the message to all of its children in the assumption that one of them might be able to handle it. If no special handling is done by containers, a directed broadcast has a similar forwarding pattern as a true broadcast except for the last hop from the parent to the destination. This broadcasting is also the default behavior if a container does not understand the message itself (the forwarding of these are necessary to allow the definition of new message types).

Improving communications. According to the broadcasting rules specified so far, all messages to visual objects must be sent indirectly through the display root. As the cost of a broadcast increases at least linearly with the number of receivers (camera-views increasing the cost even more by “flooding” sub-sections of the display space many times), it is well worth to try and tune the process a little without invalidating our stated principles.

For example, according to the principle of parental control, a container may suppress forwarding if the descendants are not interested in the message. The detection of what is interesting to an object is in general difficult to specify as gadgets typically know little about their descendants. An early version experimented with the idea of visual gadgets showing interest in certain events by setting *flags* that can be inspected by parents. The idea was later completely dropped when the rules for setting the flags became too complicated over multiple container layers.

It is also possible to question if broadcasting is suitable for messages that are of little or no interest to a container. It is for example doubtful if the requesting of attributes should be under parental control too. In the current implementation, all *frame messages* must be broadcasted and *object messages* are sent directly to components. The *attribute message* (cf. 5.3.1)—used to read and write gadget attributes—is an example of an object message that need not be monitored by a parent. This is so because attributes are properties unrelated to the fact if a gadget has a parent or not. Some gadgets—like models—do not have parents anyway. This interpretation requires that only those messages that are related to the visual aspects of a components are broadcasted.

A further reduction in number of broadcasts is obtained if the small world principle is applied consequently. A special case of parental control applies when a container itself is the initiator of a message sent to one of its children. As a child can only belong to one parent, and a parent always knows best, there is no need to broadcast the message—it can simply be forwarded directly to the child. This interpretation is often exploited in container gadgets like panels.

One case that is not sufficiently solved in the current implementation, is that, due to the lack of explicit dependency lists, a change notification of a model is broadcast even when the programmer is sure that no views are interested (when for example manipulating a model off-screen)[1]. The gadgets framework separates changes from change notification, which means when the programmer forgets to broadcast a change notification after modifying a model, an inconsistent model and view is the result.

In general, it can be said that broadcasting is a concept that enforces our state principles of parental control and small worlds. No major efficiency problems have surfaced in our implementation even though little emphasis was made on reducing its cost. This is mainly due to the fact that the size of the display space (in number of objects) tends to be quite modest. Even though visualization systems requiring real-time display of data are not well-addressed in this framework.

Multiple receives. Messages trickle down the display space following the part-of and camera-view references. At a specific point in a depth-first traversal of the display space, a message followed a certain path from the display root to a specific gadget in the display space. Because the camera-views tie up separate parts of the display space with a common visual gadget, the same message might arrive a few times by different routes at the same gadget. For example in Figure 4.9 we identify four paths from gadget A to gadget G namely (B, C, E, and F are camera-views):

ABDEG
ABDFG
ACDEG
ACDFG

The number of paths through which a message can reach a gadget is exactly the number of times that the gadget is visible on the display. This makes us observe that should a broadcast be addressed to only G, instructing itself to display itself, it would receive the message once for each position where it is visible on

[1] Oberon System 3 does not allow the exchange of the notifier with an "empty" one as in earlier Oberon versions [WG92]

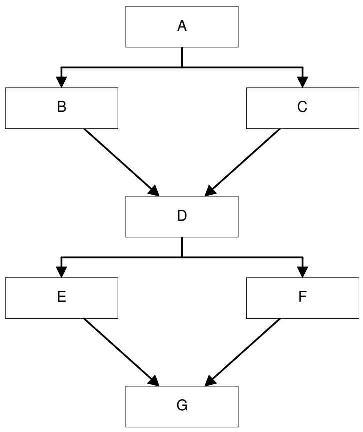

Figure 4.9: Message paths

the display. This is called *multiple receive* and is the basis of the *multiple view* framework used in the gadgets imaging model (cf. 6.4).

It turns out that one of the problems with multiple receives is that a gadget cannot detect when it happens. This is because the gadget obtains control from the outside and that a certain amount of time might have passed before a following receive occurs. A typical requirement for knowing about the "receive count" is when a gadget has to update a data structure on the first receive, but only has to use the data structure (without modification) on the remainder of the receives. This problem is addressed by time stamping each message broadcast. The time stamp is simply a number incremented for each broadcast. By keeping track of the time stamp of the last message received, a gadget can "count" how many times it received a message during a broadcast.

Important to note is that a gadget should not assume that messages arrive in time stamp sequence. It might happen that a gadget triggers another broadcast while handling a first broadcast and so delays the first broadcast's completion. The effect is that of virtual time running backwards.

Message paths. The actual message path during a broadcast has several interesting uses. It sometimes happens that a gadget needs to determine its *context*, i.e.

its location in the display space; for example, it might be interested to know something about its ancestors. A typical use could for example be to find out in which viewer or in which document a gadget is embedded. The current implementation keeps track of the path a message followed to reach a gadget in a *message thread*. The thread traces out a route through the display space and can be used by a gadget to find out more about its environment. Note that a simple static back-pointer is not sufficient to keep track of the parent of a visual gadget as the display space is a DAG.

One interesting possibility is to have a gadget change its behavior according to ancestors in the message thread. Exploitation of this possibility leads to dynamic inheritance where a gadget can inherit features and behavior from its ancestors. Another use of the message thread is to calculate the visibility of a gadget (cf. 6.4).

Message invalidation. A way to *terminate* a broadcast early is by *invalidating* the broadcasted message. This involves setting a flag in a message to indicate that the message is "invalid" and should be ignored. The flag has a second use in detecting if a gadget could handle a message or not, with the idea that the receiver invalidates a message if it could be handled successfully.

Message forwarding cycles. Earlier it was mentioned that message broadcasting can fall prone to forwarding messages in never ending circles. This can for example happen when a cycle of camera-view gadgets is created and a message broadcast takes place. This problem is difficult to solve as no global knowledge about the display space structure is the privy of a central instance that could check for the condition. A distributed solution is based on the observation that if child gadget C is inserted into container P, a cycle is present when during broadcasting to C the message arrives at P. This situation can be detected by broadcasting a message (essentially any one) to C and checking to see if it arrives at P. This of course only works if P is clever enough to notice that the test is being performed and does not forward the message to C by mistake. This can be guaranteed by only inserting C in P after the message test has been passed.

We can ensure that the display space is cycle-free by doing this test each time a containment or camera-view reference is inserted. Note possible failure of the algorithm is possible if C or an ancestor decides not to forward the test message. An implementation of this cycle test is given in section 6.2.

4.3.3 Examples

A procedure *Display.Broadcast* initiates a directed or true broadcast of a frame message into the display space (cf. 6.2). The messages that travel through the display space fall into several categories.

The most obvious use for true broadcasting is that of change notification: messages that notify that a model has changed travel through the complete display space. Coded inside the notify message we have an indication of which model has changed, including sometimes an indication what *aspect* of the model. Each view compares the model in the message with the model it displays for equality, and if necessary, updates its own representation accordingly.

Keyboard and mouse events are broadcast into the display space. A "character typed" event travels through the display space until it arrives at a frame that has the *keyboard focus*. A convention of only a single frame having the focus at any specific time ensures that the character is not consumed by two or more components. The message is invalidated to indicate that the typed character has been accepted. Mouse events are forwarded in such a way that they only arrive at frames that are physically located at the mouse position.

Further broadcasts include a request to remove a frame from the display space, to redisplay a frame, to insert a new frame at a specific position in a container, to adjust the position of a child in a container, to determine what frame is at a specific position on the display, and so on. The set of messages is discussed in more detail in chapter 6.

4.4 The Persistence Hierarchy

The persistence hierarchy is the "memory" of the Oberon system from one session to another. It collects sub-sets of the run-time configuration of components and make them persistent. An example is storing a "branch" of the display space on secondary storage. In a following session we can reconstruct an exact copy of that part and "paste" it back into the display space. A similar use, not related to the display space, is to collect template or configuration components that are shared by Oberon applications.

4.4.1 Libraries

A persistent set of components is called a *library* (cf. 5.4). A component belongs to only one library at any moment and components that do not belong to a library are said to be *free*. A free component is *inserted* in a library by *binding* it to that library. With restrictions spelled out later, components can migrate from one library to another, that is, they can be rebound. By convention, a clone of a bound object is free, and binding a composite component causes all of its constituent parts to be bound to the same library.

When two objects, belonging to different libraries, are connected by reference, we say that one object is using or *importing* the library of the other object. It is this import relationship between libraries that form the Oberon *persistence hierarchy* (Figure 4.10). With an exception sketched later, libraries can mutually import each other.

Libraries are divided into *public* and *anonymous* libraries. Public libraries have names whereas anonymous libraries have none. A system-wide cache of private libraries is maintained in module *Objects* which forms the repository for shared components in Oberon. A lookup service in this module provides a mapping from a public library name to library. If not already internalized, the lookup service will *load* a public library from disk (the library name corresponds to the library filename). This means that when an application user interface imports a component from a public library, the act of opening that document will automatically cause the public library to be loaded (and all libraries that it imports). The garbage collector frees a library from memory as soon as it detects that the library has no more clients.

Anonymous libraries are lighter weight than public libraries. They are used internally in applications to store components and are not cached. Consequently the same anonymous library can be loaded many times from disk, and also cannot be imported by other libraries (due to them having no name).

Public objects. Components bound to public libraries are called *public components* or *public objects*, as they are shared by Oberon applications. For easy retrieval, some public objects are named. A *dictionary* associates names with objects in a public library (the public names of objects should not be confused with the object's intrinsic name determined by the *Name* attribute—the public name might differ from the latter). By combining the library name L with the object name O we can refer to a public object as $L.O$. A lookup service allows us to resolve $L.O$ into a component reference.

The *Libraries.Panel* manages the contents of public libraries (Figure 3.3). It enables the user to inspect, insert into and delete components from public libraries. An especially useful feature allows the user to link a public model to a view in a

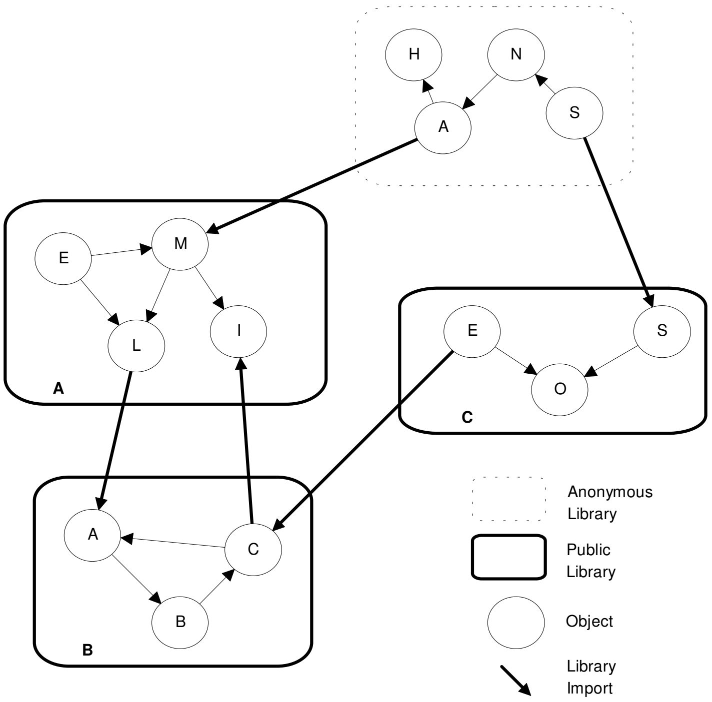

Figure 4.10: The persistence hierarchy

document. This causes the import link from the document to the public library to be automatically established when the document is opened.

Anonymous libraries. Most documents are made persistent with anonymous libraries. This involves binding all (non-public) components in a document to the same anonymous library. If different anonymous libraries were used for the same document, as for example could be imagined for implementing local storage of children in a container, references across container boundaries would not be possible (remember that we cannot import anonymous libraries).

Rather than attempting to keep the document contents consistent with its library during user editing, the binding process is completed just before storage with an empty fresh anonymous library. Also, just after loading or storing, the anonymous library is discarded. The short life of anonymous libraries restricts their role in the persistence hierarchy—they are second class to public libraries that have a longer lifetime.

Note that we have to disallow re-binding of public objects as it would cause objects that might be in use to suddenly disappear. It does however have the disadvantage that a periodic garbage collection of public libraries need to take place. This is done by inserting deep copies of all the named objects in a public library into a new public library, discarding the old library, and renaming the new library.

4.4.2 Examples

Except for older document classes that store their contents in own formats, most documents use anonymous libraries to store their contents. In some cases, like for example text, a component might use an additional anonymous library to store its constituent parts. Libraries are also used to serialize complex data structures to send them across the network. As the library mechanism requires positionable output streams to function (cf. 5.4), this involves a rather inefficient detour to a file buffer before transmission.

Public libraries are used for shared components like bitmaps and configuration data. The document menu bars are also stored in public libraries; this allows end-users to customize their work environment. The library hierarchy is often quite flat, with anonymous libraries importing public libraries, and public libraries seldom importing each other.

The library mechanism is extensible with new classes of libraries. One of these extensions is font libraries, consisting of light-weight character glyph objects. In the same theme, the TrueType font libraries generate character patterns on demand as they are retrieved from the font. An experimental object store has also been realized as a library [Mei93]. This library distinguishes itself from the standard component libraries in that loading and storing of the library contents is dynamic and incremental. Only the subset of components that are required at a specific point in time is cached in memory. Seldom used components are swapped back to disk after a period of inactivity to make space for other components in memory.

4.5 Summary

This chapter has given a broad overview over the design concepts of Oberon System 3 and Gadgets by presenting the module hierarchy, the object and message type hierarchy, the display hierarchy, and the library or persistence hierarchy. In the following two chapters we will give concrete details about the implementations of these hierarchies.

Chapter 5

Objects and Gadgets as System Components

The purpose of this chapter is to introduce the Oberon component model. This model defines the operations common to all components in Oberon System 3 and in the Gadgets system, like for example the generation, storage, cloning, linking, configuration, and location of *objects*.

5.1 The Principal Types

We start off the object and message type hierarchies by defining *Objects.Object* as the root type of component objects and *Objects.ObjMsg* as the root of the messages types (cf. 4.2):

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
   stamp: LONGINT;
   dlink, slink: Object;
   lib: Library; ref: INTEGER;
   handle: Handler
END;

ObjMsg = RECORD
   stamp: LONGINT;
   dlink: Object
END;

Handler = PROCEDURE (obj: Object; VAR M: ObjMsg);
```

The most important feature of the *Object* definition is the message handler that interprets the messages sent to an object instance. This is the *open message interface* as sketched in the previous chapter. The fields *lib* and *ref* indicate to which library the object is bound (cf. 5.4). The remainder of the fields are mainly influenced by the design of the Gadgets system. The *dlink* and *slink* fields are used to connect objects with each other in lists.

The *slink* field builds a list of objects that is passed around in the system. One use of the *slink* list is to specify the gadgets that are inserted into a container. Note that the drawback of not using an auxiliary structure to collect objects is that an object can be in one list only. Accordingly we define the lifetime of the *slink* list to be at most a single message broadcast.

The *dlink* field builds the message thread (cf. 4.3.2). The thread is extended a gadget at a time during message forwarding by prepending the objects that the message pass through. The head of the thread is the *dlink* field of the received message and the thread runs "backward" from receiver through the *dlink* field of type *Object*.

The *stamp* field of the message indicates the time at which the message was created and broadcasted. The stamp remains the same during the lifetime of the message and is not modified during message forwarding. The *stamp* field of the *Object* type is used to keep track of the time-stamp of the last message an object received. The time-stamp is used to detect multiple receives (cf. 4.3.2).

5.2 The Canonical Component Module

The *canonical module structure* speeds up component development and ensures that components can be extended. A component consists of a type definition, message definitions, procedures to copy and initialize an object, and the ubiquitous message handler. Here is an outline of the typical module structure:

```
MODULE Example;
  IMPORT Objects;

  TYPE
    MyObject* = POINTER TO MyObjectDesc;
    MyObjectDesc* = RECORD (Objects.ObjDesc)
      (* extended fields *)
    END;

    MyMsg* = RECORD (Objects.ObjMsg)
      (* message arguments *)
    END;
```

```
    PROCEDURE Copy*(VAR M: Objects.CopyMsg; from, to: MyObject);
      (* Copy fields of from to to *)

    PROCEDURE Handle*(obj: Objects.Object; VAR M: Objects.ObjMsg);
      (* Message handler *)

    PROCEDURE Init*(obj: MyObject);
    BEGIN
      obj.handle := Handle; (* install message handler *)
      (* initialize own fields *)
    END Init;

    PROCEDURE New*; (* Component generator *)
    VAR obj: MyObject;
    BEGIN NEW(obj); Init(obj); Objects.NewObj := obj
    END New;

END Example.
```

In the Oberon language, the items marked "*" are *exported* from the module and are visible to clients that import this module. Due to a lack of expressiveness in the Oberon language it is necessary to export practically everything from the module even though the user of a component and the extender of a component are interested in different aspects of the module, namely the external and the internal view.

External view. The component client is interested in the component type definition, the messages that it understands, the procedures that can be applied to it, and how to generate one. Of these external aspects, generating a new component has some interesting problems.

Calling procedure *New* generates a new instance of the component and assigns it to the global variable *Objects.NewObj*. The caller is picks up the object from there. This round-about way of generating objects is necessary because instantiation of objects from secondary storage and by an end-user request involves an *up-call* to a module that might not have been loaded yet. Executing an up-call like this can only be done by executing a *command* procedure, which is restricted to be parameterless and result-less by the run-time system. (The module *Modules* provide an interface to the dynamic module loader for exactly this operation.)

There have been attempts to create a more elegant component generation technique for Oberon. One approach involves using a module called *Types* that can allocate an object dynamically by specifying its type name [fC]. Unfortunately no component code is executed and hence we cannot initialize the object correctly.

In Oberon-2 type-bound procedures are bound automatically [Mös93], but constructors (as in C++ [Str87]) were left out. An approach that extends Oberon with metaprogramming facilities [Tem94] has promise in that procedure activation frames can be build and executed at run-time, and thus lifting the restrictions on formal and return parameters.

The approaches with module *Types* and *type-bound procedures* have the disadvantage that a dependency on a specific component type is created. This does not allow the type of a component to vary over its lifetime. This occurs for example when we would like to replace out-dated components with new components of a different type. The generator procedures have the advantage that they create a level of indirection between the generator name and the actual type, allowing us to vary the object type independent from its generator. In fact, this feature has been used numerous times over the lifetime of the system to update components transparently with more modern versions.

The approach using meta-programming was ruled out as these features are not standard in all Oberon distributions, which leaves us with the current implementation using a global variable. Eventually when Oberon is extended with multi-threaded ability, this decision would have to be revised.

A further important aspect is how clients know how to make use of a component, as a component is only as good as its documentation. In this regard, the elimination of definition modules from the Oberon language is unfortunate. In Oberon System 3, programmers are expected to add *exported comments* to their modules, which are then extracted and formatted into hyper-text by a tool called Watson [Sal95] to form the component documentation. By keeping the documentation in the module itself, we have a larger possibility of it being consistent with the actual implementation. A similar approach is followed in Java [Mic95], where the position of exported comments is part of the language syntax.

Internal view. Two internal views of objects are possible, namely that of the implementer of the component and of the person that wants to extend the module. The primary task of the implementer is the implementation of the message handler. Depending on the class of object, different message types are to be handled. The simplest objects only respond to the *object messages* whereas the visual objects respond to both the *object messages* and the *frame messages*. The typical handling of object messages is sketched in the following section.

To extend a component, at least the object type, message handler, *Copy*, and *Init* procedure need to be exported. The object extension must watch out that call-

Type	*Message variants*
Objects.AttrMsg	get, set, enum
Objects.LinkMsg	get, set, enum
Objects.CopyMsg	deep, shallow
Objects.BindMsg	
Objects.FileMsg	load, store
Objects.FindMsg	

Table 5.1: Object message summary

ing the *Init* procedure of the base object is done before initializing the extension and that the correct message handler is written over that of the base object class. This "hidden" knowledge is not always available to the extender who typically does not have the source code of the base object. The situation is aggravated when subtle sequence dependencies between base class and subclass are introduced by objects sending messages to themselves [TGP89].

5.3 The Object Messages

The *object messages* are concerned with object attributes, links between objects, copying objects, persistency, and locating objects in the display space. The object messages are all defined in module Objects. A summary of the object messages and their variants is shown in Table 5.1. Message *variants* determine the exact interpretation, like an operation code, of a specific message type. Note that by convention, all message types names end in "Msg".

5.3.1 The Attribute Message.

The attribute message retrieves, sets or enumerates the attributes of an object. An important client of the attribute message is the attribute Inspector which uses it to inspect and modify a gadget. Single attribute are named and have typed values. Only few basic types, called *attribute classes*, are supported.

```
CONST
  (* Attribute class *)
  Inval = 0; String = 2; Int = 3; Real = 4; LongReal = 5; Char = 6; Bool = 7;

  (* Operation type *)
  enum = 0; get = 1; set = 2;

TYPE
  AttrMsg = RECORD (Objects.ObjMsg)
    id: INTEGER; (* get, set or enum *)
    Enum: PROCEDURE (name: ARRAY OF CHAR);
    name: ARRAY 32 OF CHAR;
    res: INTEGER;
    class: INTEGER; (* Inval, String, Int, Real, LongReal, Char or Bool *)
    i: LONGINT;
    x: REAL;
    y: LONGREAL;
    c: CHAR;
    b: BOOLEAN;
    s: ARRAY 64 OF CHAR
  END;
```

The *id* field indicates the operation performed by the object, namely to *set* or *get* an attribute called *name*, or to *enumerate* all attribute names. Note that a message that contains an *id* field is said to come in *variants*, with value of the field specifying which variant. The *res* field returns a code to indicate if the operation was successful. A zero or positive value indicates that an attribute was successfully set or retrieved. A negative value indicates a failure.

Attributes types are identified by numbered constants. The *class* field is used as an in-parameter to indicate what the attribute type is during a set operation. A get operation uses *class* as an out-parameter to indicate the type of the attribute returned. Depending on the value of *class* the attribute value set or retrieved is located in the corresponding field *i*, *x*, *y*, *c*, *b* or *s*. Enumerating the attributes names of an object involves passing an *Enum* procedure that is called by the receiver for each attribute name belonging to the object.

As the use of messages with variants and *res* fields is common, we will present a first detailed example of how the message is applied by clients:

```
VAR
  obj: Objects.Object;
  M: Objects.AttrMsg;

(* Set the "Value" attribute of obj to 42. *)
  M.id := Objects.set; M.name := "Value";
  M.class := Objects.Int; M.i := 42; M.res := -1;
  obj.handle(obj, M);
  IF M.res < 0 THEN (* failure, wrong attribute type? *) END
```

```
(* Get the "Value" attribute of obj. *)
  M.id := Objects.get; M.name := "Value";
  M.class := Objects.Inval; M.res := -1;
  obj.handle(obj, M);
  IF M.res >= 0 THEN
    (* M.class indicates type and where to locate the value *)
  ELSE (* failure, attribute does not exist? *)
  END
```

Enumerating attributes is implemented using the following code skeleton:

```
PROCEDURE Enum(name: ARRAY OF CHAR);
BEGIN (* called for each attribute of obj *)
END Enum;

VAR
  obj: Objects.Object;
  M: Objects.AttrMsg;

M.id := Objects.enum; M.Enum := Enum;
obj.handle(obj, M);
```

We now present an example of how the attribute message is handled by an object by showing an extract of the message handler. This example assumes that the object has an integer attribute called *Value*. Internally, this attribute is stored in a RECORD field called *val*. The only special situation we have to take care of is the attribute *Gen*, which must return the object generator (cf. 3.2). Note that the attribute handling mechanism is quite regular and is often copied as standard "code pattern".

```
PROCEDURE HandleAttributes(obj: Objects.Object; VAR M: Objects.AttrMsg);
BEGIN
  IF M.id = Objects.get THEN
    IF M.name = "Gen" THEN
      M.class := Objects.String; COPY("Example.New", M.s);
      M.res := 0
    ELSIF M.name = "Value" THEN
      M.class := Objects.Int; M.i := obj.val;
      M.res := 0
    END
  ELSIF M.id = Objects.set THEN
    IF M.name = "Value" THEN
      IF M.class = Objects.Int THEN
        obj.val := SHORT(M.i);
        M.res := 0
      ELSE (* wrong type ! *)
        M.res := -1
      END
    END
```

```
    ELSIF M.id = Objects.enum THEN
      M.Enum("Value");
    END
  END HandleAttributes;

  PROCEDURE Handle(obj: Objects.Object; VAR M: Objects.ObjMsg);
  BEGIN
    WITH obj: MyObject DO
      IF M IS Objects.AttrMsg THEN
        WITH M: Objects.AttrMsg DO HandleAttributes(obj, M) END
      ELSE (* message not handled *)
      END
    END
  END Handle;
```

The careful reader will notice that attribute *Gen* is not enumerated—this is a historical oversight.

5.3.2 The Link Message

The link message is used to set, retrieve, and enumerate links between objects. The message has a similar structure and use to *AttrMsg* except that it has only one class, namely of type *Objects.Object*. The object to be set or retrieved as a link is passed in the field *obj*. We will not repeat an example for using the link message due to its similarity with the attribute message.

```
CONST
  (* Operation type *)
  enum = 0; get = 1; set = 2;

TYPE
  LinkMsg = RECORD (Objects.ObjMsg)
    id: INTEGER; (* get, set or enum. *)
    Enum: PROCEDURE (name: ARRAY OF CHAR);
    name: ARRAY 32 OF CHAR;
    res: INTEGER;
    obj: Object.Object;
  END;
```

5.3.3 The Copy Message

Copying an object involves sending a *CopyMsg* to the object to make a copy of itself. The *id* field indicates if a *deep* or *shallow* copy should be made (cf. 3.2).

```
CONST
 shallow = 0; deep = 1; (* CopyMsg id *)

TYPE
  CopyMsg = RECORD (ObjMsg)
    id: INTEGER; (* deep or shallow *)
    obj: Object (* result *)
  END;
```

If the object is a root of a data structure and a *shallow* copy is made, the receiver only needs to copy references to other objects. In the case of a deep copy, the receiver forwards the message to all referenced objects to make copies of themselves. The process continues in a recursive manner until the whole data structure has been copied. In a graph of objects where many message paths lead to the same object *X*, this naive strategy will copy object *X* exactly the number of times a forwarding path to that object exist. A *structure preserving* deep copy is obtained if an object only copies itself the first time the copy message arrives and returns that copy for each further time the message arrives. The message time stamp is used to detect when the message arrives again by assigning it to the *stamp* field in the type *Objects.Object* when the message arrives the first time. The actual copy is cached in the *dlink* field of the object.

The following code fragment shows the process for an object that contains a reference to another object.

```
TYPE
  MyObject = POINTER TO MyObjectDesc;
  MyObjectDesc = RECORD (Objects.ObjDesc)
    val: INTEGER;
    ref: Objects.Object;
  END;

PROCEDURE Copy(VAR M: Objects.CopyMsg; from, to: MyObject);
BEGIN
  to.val := from.val;

  (* copy ref *)
  IF from.ref = NIL THEN to.ref = NIL
  ELSE
    IF M.id = Objects.deep THEN
      from.ref.handle(from.ref, M); (* forward message *)
      to.ref := M.obj
    ELSE to.ref := from.ref (* just return the reference *)
    END
  END
END Copy;

PROCEDURE Handle(obj: Objects.Object; VAR M: Objects.ObjMsg);
```

```
  VAR obj1: MyObject;
BEGIN
  IF M IS Objects.CopyMsg THEN
    WITH M: Objects.CopyMsg DO
      IF M.stamp = obj.stamp THEN (* second arrive *)
        M.obj := obj.dlink
      ELSE (* first arrive *)
        NEW(obj1); obj.stamp := obj.stamp; obj.dlink := obj1;
        Copy(M, obj, obj1); M.obj := obj1
      END
    END
  ELSE
  END
END Handle;
```

There are a few points to observe regarding copying objects. The message time stamp should remain the same for the complete copy operation and should have a different value from the previous copy operation. The use of *dlink* to remember the copy—not the true purpose of this field—disallows making structure copies that depend on the context as determined by the message path (remember that the message path is built using a chain of objects connected by the *dlink* field).

5.3.4 The Find Message

The find message locates objects having a specific name in a data structure of objects. The typical use is locating objects in the display space.

```
TYPE
  FindMsg = RECORD (ObjMsg)
    name: Name;
    obj: Object (* result *)
  END;
```

On receiving this message, an object should check if its own name matches that of the message. If the same, the object should assign itself to the message *obj* field. If not, the object should forward the message for further processing to its children. Both a depth-first and breadth-first search strategies are allowed, although the breadth-first strategy seems to make the most sense as it locates objects nearest to the starting point of the search. Note that the nesting of containers automatically creates a scoping mechanism for reducing ambiguities during the search.

The *FindMsg* is often sent to the container of the visual gadget that has just executed an attribute as a command. By convention, the *executor* gadget assigns its parent (found in the message thread) to a global variable called *Gadgets.context* before executing such a command. This forms the starting point of the search with the find message. We can combine the *FindMsg* and *AttrMsg* to locate an object called "XYZ" in the context and read the value of one of its attributes.

The following example makes use of the procedural level-1 interface to messages. A standard procedure *Gadgets.FindObj* hides the searching process, and procedure *Attributes.GetInt* reads the value of the attribute.

```
MODULE Example;

IMPORT Attributes, Gadgets;

  PROCEDURE Test*;
  VAR obj: Objects.Object; x: LONGINT;
  BEGIN
    obj := Gadgets.FindObj(Gadgets.context, "XYZ");
    Attributes.GetInt(obj, "Value", x);
    (* do something with x ... *)
  END Test;

END Example.
```

5.4 The Library Mechanism

Libraries. The library persistency mechanism forms the bulk of the implementation of module *Objects*. A library is an abstract data type for managing the objects bound to the library and for the object names in the dictionary associated with a library.

```
TYPE
  Dictionary = RECORD END;

  Library = POINTER TO LibDesc;
  LibDesc = RECORD
    name: ARRAY 32 OF CHAR;
    dict: Dictionary;
    maxref: INTEGER;

    GenRef: PROCEDURE (L: Library; VAR ref: INTEGER);
    GetObj: PROCEDURE (L: Library; ref: INTEGER; VAR obj: Object);
    PutObj: PROCEDURE (L: Library; ref: INTEGER; obj: Object);
    FreeObj: PROCEDURE (L: Library; ref: INTEGER);
    Load: PROCEDURE (L: Library);
    Store: PROCEDURE (L: Library)
  END;

  EnumProc = PROCEDURE (L: Library);
```

```
PROCEDURE ThisLibrary (name: ARRAY OF CHAR): Library;
PROCEDURE FreeLibrary (name: ARRAY OF CHAR);
PROCEDURE Enumerate (P: EnumProc);
PROCEDURE OpenLibrary (L: Library);
```

The *name* field is either an empty string when the library is *anonymous* or the library name when it is *public*. The library name in the case of a public library corresponds to the disk file the library was loaded from. Libraries have methods associated to insert, retrieve, and delete objects, and to store and load the library. Module *Objects* provide standard implementations for the methods; a library is initialized by these with procedure *OpenLibrary*.

Loaded public libraries are cached internally to module *Objects* and are accessed (and, if required, loaded) by *ThisLibrary*. A public library is unloaded with *FreeLibrary* and the library cache is enumerated with *Enumerate*.

Reference numbers. Bound objects are assigned *reference numbers* in a library. A library can be imagined as a potentially infinite ARRAY of objects where the reference number is the index. A bound object's library and reference number are assigned to the variables *lib* and *ref* in type *Objects.Object*. The library method *GenRef* returns an unoccupied index in the library, *GetObj* retrieves an object from a specific index, *PutObj* inserts an object at a specific index, and *FreeObj* removes an object from a specific index. The field *maxref* indicates the highest occupied index. Internally, objects are stored in a multi-level sparse tree with index and leaf nodes of references to leaf nodes and objects respectively.

Binding an object is the process by which it obtains a reference number in a library. The *bind message* indicates the library in which an object must insert itself.

```
TYPE
  BindMsg = RECORD (ObjMsg)
    lib: Library
  END;
```

According to the rules outlined in section 4.4, an object should only bind itself if it is either free or belongs to an anonymous library different from its current library. The rules are summarized in the following code fragment.

```
PROCEDURE BindObj(obj: Objects.Object; lib: Objects.Library);
VAR ref: INTEGER;
BEGIN
  IF lib # NIL THEN
    IF (obj.lib = NIL) OR (obj.lib.name[0] = 0X) & (obj.lib # lib) THEN
      lib.GenRef(lib, ref);
      IF ref >= 0 THEN lib.PutObj(lib, ref, obj) END
    END
  END
END BindObj;
```

Dictionaries. The primary function of dictionaries is to associate names with reference numbers (and thus indirectly with objects having those reference numbers). A secondary function of dictionaries is to store often used strings. This is useful when objects have to write the same string many times in a library file—in this case writing the dictionary entry number is much shorter. We thus enter in the dictionary (*key*, *string*) pairs. Key values are split into two ranges: negative key values are used for strings, and zero or positive key values for names associated with reference numbers.

```
PROCEDURE GetKey (VAR D: Dictionary; name: ARRAY OF CHAR; VAR key: INTEGER);
PROCEDURE GetName (VAR D: Dictionary; key: INTEGER; VAR name: ARRAY OF CHAR);
PROCEDURE PutName (VAR D: Dictionary; key: INTEGER; name: ARRAY OF CHAR);
```

Procedures *PutName* and *GetName* retrieve and set the name corresponding to a key. Procedure *GetKey* associates a key ($key < 0$) with a name—if not done so already—and returns its value. A second call to *GetKey* with the same name returns the same key.

Storing and loading libraries. The following procedures of module *Objects* load and store a library from and to a file. The library is positioned at *pos* in file *f* and has a length *len*. The bound objects and dictionary are packaged inside the extent [pos, pos+len) of the file.

```
PROCEDURE LoadLibrary (L: Library; f: Files.File; pos: LONGINT; VAR len: LONGINT);
PROCEDURE StoreLibrary (L: Library; f: Files.File; pos: LONGINT; VAR len: LONGINT);
```

During a store operation, the procedure *StoreLibrary* will request each of the bound objects to write their contents to the file. Correspondingly, during a load operation, the procedure *LoadLibrary* will request each object to read its contents from the file. The two requests are combined in the file message with variants *load* and *store*.

```
CONST
  load = 0; store = 1; (* FileMsg id *)

TYPE
  FileMsg = RECORD (ObjMsg)
    id: INTEGER; (* load or store *)
    R: Files.Rider
  END;
```

External library format. The library image is divided into four sections on disk. The *header section* identifies the library with a special tag and contains the offsets to the other three sections in the file. The *generator section* comprises a sequence of generator strings (attribute "Gen"), one for each bound object. The *data section* contains the contents of objects written to the file. The *dictionary section* holds the contents of the libraries' dictionary.

The following pseudo-code illustrates the *StoreLibrary* implementation.

```
PROCEDURE StoreLibrary (L: Library; f: Files.File; pos: LONGINT; VAR len: LONGINT);

  Write the library tag

  (* Generator section *)
  FOR each bound object DO
    Retrieve generator attribute
    Write generator as string to file
  END

  (* Data section *)
  FOR each bound object DO
    Request object to write its data to file
  END

  (* Dictionary section *)
  FOR each dictionary entry DO
    Write reference number to file
    Write object name to file
  END
```

The generator section is written separate from the data section as all objects need to be instantiated during loading, before requesting any object to load its contents. Also, the dictionary must be written last as objects extend the dictionary with additional entries when storing themselves. Both requirements are related to the way how references are made persistent (see the paragraph titled "Persistent references" for details).

The following pseudo-code illustrates the load operation. To save disk space, we assign numbers to generator strings instead of unnecessarily writing the same generator multiple times to the file. Also consecutively used reference numbers are collected in *runs* in a form or run-length encoding.

```
PROCEDURE LoadLibrary (L: Library; f: Files.File; pos: LONGINT; VAR len: LONGINT);

  Read the library tag

  Position to dictionary section
  FOR each dictionary entry DO
    Read reference number from file
    Read object name from file
    Insert entry in dictionary
  END

  Position to generator section
  FOR each object DO
    Read generator string from file
    Execute generator string as command
    Assign generated object to current index
  END

  Position to data section
  FOR each object DO
    Request object to read data from file
  END
```

Persistent references. Objects are instantiated before the object contents are loaded. On receiving a load message, an object can access—with *GetObj*—an object that follows *physically after* it in the file. This makes reference numbers a natural mechanism for persistent references.

The binding process allocates a reference number for each object. Conversion between object reference and reference number is called *swizzling*. To swizzle a reference to a reference number we only need to retrieve the *ref* field of the referenced object. To swizzle a reference number to an object reference, we use the *GetObj* method of a library. When requested to store its contents, an object swizzles references to other objects and writes the reference numbers in the library file. The opposite process reconstructs data structures when loading an object.

Swizzling is only slightly more complicated when we have references between different libraries. A reference to an object in another library requires that we use both the reference number and imported library name as an object identification. This shows why anonymous libraries cannot be imported—they have no names and cannot be located with *Objects.ThisLibrary*.

The following two procedures from module *Gadgets* hide most of the complication of swizzling from the programmer. Procedure *WriteRef* writes a pointer value and procedure *ReadRef* reads it back. Note that the programmer has to pass in *lib* the library of the object that is writing or reading the reference—this is so that the dictionary can be extended when necessary.

```
PROCEDURE WriteRef(VAR r: Files.Rider; lib: Objects.Library; obj: Objects.Object);
VAR ref: INTEGER;
BEGIN
  IF obj = NIL THEN Files.WriteInt(r, -1)
  ELSE
    IF obj.lib # NIL THEN
      IF obj.lib # lib THEN
        IF obj.lib.name = "" THEN (* private library *)
          (* error: object belonging to private library is referenced *)
        ELSE
          Files.WriteInt(r, obj.ref);
          Objects.GetKey(lib.dict, obj.lib.name, ref);
          Files.WriteInt(r, ref);
        END;
      ELSE (* belongs to the same library *)
        Files.WriteInt(r, obj.ref);
        Objects.GetKey(lib.dict, "", ref);
        Files.WriteInt(r, ref)
      END;
    ELSE (* error: object without library is referenced *)
    END;
  END;
END WriteRef;

PROCEDURE ReadRef(VAR r: Files.Rider; lib: Objects.Library; VAR obj: Objects.Object);
VAR ref, entry: INTEGER; F: Objects.Library; name: ARRAY 32 OF CHAR;
BEGIN
  Files.ReadInt(r, ref);
  IF ref = -1 THEN obj := NIL
  ELSE
    Files.ReadInt(r, entry);
    Objects.GetName(lib.dict, entry, name);
    IF name = "" THEN (* own library *)
      F := lib
    ELSE F := Objects.ThisLibrary(name)
    END;
    IF F # NIL THEN
      F.GetObj(F, ref, obj);
      IF obj = NIL THEN (* error: imported object does not exist any more *)
      END
    ELSE (* error: imported library not found *)
    END
  END
END ReadRef;
```

Several error conditions may occur during loading and storing of libraries, including missing libraries and objects, referencing freed objects or importing ones from anonymous libraries. All situations can be detected but it is not always sure what to do when they occur. The current implementation is forgiveful at the risk of losing information. This involves writing NIL pointers for incorrectly referenced objects inside *WriteRef* and returning a NIL pointer on error conditions in *ReadRef*. As error conditions often occur due to programmer negligence (for example forgetting to bind objects), we also write warning messages in the log.

Note that the call to *Objects.ThisLibrary* in procedure *ReadRef* can lead to a large number of side-effects. It might for example be that the referenced public library still has to be loaded. If that library imports another, it would have to be loaded too, and so on recursively. To prevent an infinite cycle of one library trying to load the other, which in turn tries to load the first, *Objects.ThisLibrary* has to insert a newly created public library in the library cache *before* any of its objects is requested to load itself.

Robustness. Library loading can either fail *completely* or *partially*. A complete failure occurs when a public library does not exist. A partial failure occurs when objects are not present in public libraries where they are expected or when a module implementing an object cannot be found or loaded. Such condition occurs as users might edit public libraries or even delete modules. A graceful degradation strategy for library loading implies that programmers have to reckon with failures instead of terminating the execution.

The simpler types of failure involve missing object and libraries. An object that is aware that *ReadRef* might return NIL can attempt to correct the situation or try to restrict its own functionality.

A more disruptive failure is when a module that implements an object cannot be loaded. This situation is reported to *LoadLibrary* by the module loader when executing the object generator. The problem is partially addressed in *LoadLibrary* by creating a *substitute* or *dummy* object instead of the original object. The dummy assumes responsibility for the original object data. The exact data length is known in advance when we insert a length tag in front of each object's data. At first thought, it seems that the dummy can read an image of the data into memory, which is then later stored in the same format. This is however only possible if the object data is relocatable and does not contain references. References might change without the dummy being aware of it, making the library inconsistent on storage. Without type information, the dummy cannot interpret the data to detect which references must be corrected. And by ignoring the data, a part of the data structure reference is lost (namely the outgoing references from the dummy). The current implementation does exactly the latter.

Successfully loaded objects now might suddenly find that they own a reference to a dummy (by type testing the result of *ReadRef*). It is now up to the object to decide what to do with the dummy. Most objects simply cut away the dummy by setting the reference to NIL and then reporting the situation to the log. Storing a document that reports such a failure during loading ultimately involves losing data but gaining self-consistency.

Interestingly, the non-recursive nature of the library storage mechanism has an advantage in robustness over recursive solutions. A recursive storage solution nests referenced objects inside an object on disk (see [Gri91] for implementations). When the outer object's loading fails, the nested objects are lost as well, even though they could potentially be loadable. As the library storage mechanism is non-recursive, only those objects with inconsistent or missing modules fail on loading and no others.

A further failure possibility is related to components undergoing continual refinement and modification of external data formats. Objects typically write a data format version code as a header to their contents on disk. An object always writes the latest data format and can read all previous data formats. The ability to read all older data formats is preferred to supplying converters to convert older components (experience shows that users cannot be relied on to do such conversions correctly).

The version code approach only works reliably if all independent users have exactly the same version of a component. As soon as one Oberon installation forgets to upgrade older modules, the system has to contend with versions that are in advance of what it expects. A possible solution is to ignore advanced versions and create a default component configuration. However, the distribution problem is so problematic in network-aware component-based systems that such an ignorant solution seems to be unacceptable. The component distribution problem is addressed in the Gadgets framework on a higher level of documents (Chapter 7).

5.5 Programming Support

The Oberon base system only provides the object message definitions and an implementation of the library mechanism. These provisions are sufficient but spartanic and often require a lot of programming for even simple operations. The Gadgets system adds more behavior and conveniences on top of the base system, some of which the following paragraphs introduce.

Level-1 programming support. The attribute message mechanism tends to be tedious when lots of attributes are used or when conversions between attribute types have to be made. As a concession to level 1 programmers (cf. 3.1) a more convenient procedural interface in module *Attributes* hides the messaging mechanism. Module *Links* provides a procedural interface for setting and getting links. Module *Gadgets* contains several more procedures ranging from reading/writing object names, generating objects by name, creating model-view pairs and locating named and public objects. Level-1 programmers are referred to the Gadgets programmers guide for more details.

```
PROCEDURE GetBool(obj: Objects.Object; name: ARRAY OF CHAR; VAR b: BOOLEAN);
PROCEDURE GetInt(obj: Objects.Object; name: ARRAY OF CHAR; VAR i: LONGINT);
PROCEDURE GetReal(obj: Objects.Object; name: ARRAY OF CHAR; VAR x: REAL);
PROCEDURE GetLongReal(obj: Objects.Object; name: ARRAY OF CHAR; VAR y: LONGREAL);
PROCEDURE GetString(obj: Objects.Object; name: ARRAY OF CHAR;
        VAR s: ARRAY OF CHAR);

PROCEDURE SetBool(obj: Objects.Object; name: ARRAY OF CHAR; b: BOOLEAN);
PROCEDURE SetInt(obj: Objects.Object; name: ARRAY OF CHAR; i: LONGINT);
PROCEDURE SetReal(obj: Objects.Object; name: ARRAY OF CHAR; x: REAL);
PROCEDURE SetLongReal(obj: Objects.Object; name: ARRAY OF CHAR; y: LONGREAL);
PROCEDURE SetString(obj: Objects.Object; name, s: ARRAY OF CHAR);

PROCEDURE GetLink(obj: Objects.Object; name: ARRAY OF CHAR; VAR ob1: Objects.Object);
PROCEDURE SetLink(obj: Objects.Object; name: ARRAY OF CHAR; ob1: Objects.Object);
```

Decorations. Experience shows that it is often useful to "attach" new attributes to objects that the objects did not originally define. The attached attributes live with the predefined attributes and are typically ignored by the object itself (as it would not know what to do with them). Examples of attached attributes include a reference to a tutorial that explains something about the object or constraint information that is used for visual layout. These attributes are of use to tools that *operate* on those objects and not to the objects themselves. This means of attaching information to objects is called the *decorator* design pattern [GHJV95], and the Gadgets framework explicitly supports it.

The types *Gadgets.Object* and *Gadgets.Frame*, declare an additional RECORD field *attr* that references an abstract data structure that keeps track of these attributes. We will only introduce the type *Gadgets.Object* here, as the type *Gadgets.Frame* is discussed in more detail in the following chapter.

```
TYPE
  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD (Objects.ObjDesc)
    attr: Attributes.Attr;
    link: Links.Link;
  END;
```

A similar facility as *attached attributes* is provided for links by the Gadgets framework. It enables the user or programmer to connect components to each other or to "decorate" a component with another. Just as for attributes, the component is not aware of links that have been attached to it. An additional RECORD field *link* in types *Gadgets.Object* and *Gadgets.Frame* manages the link abstract data structure.

Default Message Handlers. Even though module *Objects* defines the *AttrMsg*, *LinkMsg*, *CopyMsg*, *BindMsg*, *FileMsg* and *FindMsg*, it does not provide any support for handling these messages. In a similar manner module *Display* defines the set of frame messages but does not provide further support for handling them. Some aspects of these messages can be handled in a generic way for all objects and have consequently been factored out into *default message handlers*. Handlers for extensions of *Gadgets.Object* and *Gadgets.Frame* are provided in module *Gadgets*. The handlers are called *Gadgets.objecthandle* and *Gadgets.framehandle* respectively. The implementation of an own message handler must pass control to the default handlers in a similar manner as a super-call in other object-oriented languages. Messages that are not understood by a message handler should be passed to the default message handler for interpretation. The following paragraphs state the default handling of the object messages by the default message handlers.

AttrMsg. The default handling of the attribute messages includes the universal "Name" attribute of a gadget and the management of the attached attributes. The field *attr* of types *Gadgets.Object* and *Gadgets.Frame* refers to a data structure in module *Attributes* that contains the list of attached attributes.

LinkMsg. The default handling of the link message involves the handling of links that have been attached to a gadget. The field *link* of types *Gadgets.Object* and *Gadgets.Frame* refers to a data structure in module *Links* that contains the list of attached links. The type *Gadgets.Frame* contains a field *obj* that refers to the model of the gadget and is seen as a link called "Model" by clients.

CopyMsg. The default handling of the copy message is to copy the fields belonging to the types *Gadgets.Object* and *Gadgets.Frame*.

BindMsg. The default handling of the bind message is to bind the object and all the objects that the attached links reference.

FileMsg. The default handling of the file message is to store or load the fields of the types *Gadgets.Object* and *Gadgets.Frame*.

FindMsg. The default handling of the find message is to check if the searched for object matches self and returning itself, should this be the case.

5.6 An Example

The following module is an implementation of a component called "Complex", a model gadget having two attributes specifying the real and imaginary parts of a complex number.

```
MODULE Complex;
  IMPORT Files, Objects, Attributes, Gadgets;

  TYPE
    Complex* = POINTER TO ComplexDesc;
    ComplexDesc* = RECORD (Gadgets.ObjDesc)
      real*, imag*: REAL
    END;

  PROCEDURE HandleAttributes(obj: Complex; VAR M: Objects.AttrMsg);
  BEGIN
    IF M.id = Objects.get THEN (* retrieve attribute *)
      IF M.name = "Gen" THEN
        M.class := Objects.String; COPY("Complex.New", M.s);
        M.res := 0
      ELSIF M.name = "Real" THEN
        M.class := Objects.Real; M.x := obj.real; M.res := 0
      ELSIF M.name = "Imag" THEN
        M.class := Objects.Real; M.x := obj.imag; M.res := 0
      ELSE Gadgets.objecthandle(obj, M)
      END
    ELSIF M.id = Objects.set THEN (* set attribute *)
      IF (M.name = "Real") & (M.class = Objects.Real) THEN
        obj.real := M.x; M.res := 0
      ELSIF (M.name = "Imag") & (M.class = Objects.Real) THEN
        obj.imag := M.x; M.res := 0
      ELSE Gadgets.objecthandle(obj, M)
      END
    ELSIF M.id = Objects.enum THEN (* enumerate attributes *)
      M.Enum("Real"); M.Enum("Imag");
      Gadgets.objecthandle(obj, M)
    END
  END HandleAttributes;

  PROCEDURE Handler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
```

```
  VAR obj0: Complex;
  BEGIN
    WITH obj: Complex DO
      IF M IS Objects.AttrMsg THEN
        WITH M: Objects.AttrMsg DO
          HandleAttributes(obj, M);
        END
      ELSIF M IS Objects.CopyMsg THEN
        WITH M: Objects.CopyMsg DO
          IF M.stamp = obj.stamp THEN M.obj := obj.dlink
          ELSE (* first time copy message arrives *)
            NEW(obj0); obj.stamp := M.stamp; obj.dlink := obj0;
            (* copy object *)
            obj0.handle := obj.handle;
            obj0.real := obj.real; obj0.imag := obj.imag;
            M.obj := obj0
          END
        END
      ELSIF M IS Objects.FileMsg THEN
        WITH M: Objects.FileMsg DO
          IF M.id = Objects.store THEN
            Files.WriteReal(M.R, obj.real);
            Files.WriteReal(M.R, obj.imag);
            Gadgets.objecthandle(obj, M)
          ELSIF M.id = Objects.load THEN
            Files.ReadReal(M.R, obj.real);
            Files.ReadReal(M.R, obj.imag);
            Gadgets.objecthandle(obj, M)
          END
        END
      ELSE Gadgets.objecthandle(obj, M)
      END
    END
  END Handler;

  PROCEDURE New*;
  VAR obj: Complex;
  BEGIN
    NEW(obj); obj.handle := Handler;
    obj.real := 0.0; obj.imag := 0.0;
    Objects.NewObj := obj;
  END New;

END Complex.
```

The Complex gadget is derived from *Gadgets.Object* to make use of the default message handlers. This reduces the message handler to the handling of the *Objects.AttrMsg*, *Objects.CopyMsg* and *Objects.FileMsg*—the remainder of the object messages are handled by *Gadgets.objecthandle*.

The implementation is derived from a standard model gadget pattern. The statements and identifiers written in bold face show the changes required in this pattern. This involves declaring the gadget type, handling the two new attributes, copying the two RECORD fields, storing and loading the RECORD fields, and initializing the RECORD fields. The remainder of the code was copied.

Note that reusing code in this case involves *copying* it and not *sharing* it. This illustrates that the message handler approach often results in larger grained messages than what is typically found in object-oriented languages. A proficient programmer would factorize out the common code by introducing separate messages for initializing, reading and writing contents, copying the RECORD fields, etc. At the same time the exact behavior of an object is made more opaque by hiding the factorized code in the base class. It is thus not unexpected news that programmers who equate reuse with code sharing make extensive use of late binding, even if it has a high cost with complicated message interfaces.

5.7 Summary

This chapter introduced the Oberon component model by discussing the object and message type definitions, the set of object messages, how objects are made persistent, and how the Gadget systems extends this model for more convenient programming. In conclusion an example implementation of a model gadget was given.

Chapter 6

Visual Gadgets

Visual gadgets' most tangible properties are that they display themselves and interact with the user. Other properties include being able to print, communicate, and integrate in any container. The sum of a gadget's properties is defined by the message protocols it understands. These message protocols are defined along the lines the *end-user* expects from gadgets—an advantageous approach when the user migrates from level-0 to level-1, and ultimately level-2 programming (cf. 3.1). This strategy results in a compact set of messages, including a display message, a print message, a size adjustment message, a selection message, a drag and drop message, a control message for removing gadgets, and an input message for handling mouse and keyboard events.

The largest part of this chapter is concerned with the essence of the Gadgets system, namely the use of the common message protocols. The remainder of the chapter discusses the Gadgets imaging model.

6.1 The Principal Types

The type *Display.Frame*, derived from *Objects.Object*, is the base type of all visual components in the Oberon system. Messages related to frames are derived from the type *Display.FrameMsg* and are called *frame messages*.

```
TYPE
  Frame = POINTER TO FrameDesc;
  FrameDesc = RECORD (Objects.ObjDesc)
    next, dsc: Frame;
    X, Y, W, H: INTEGER
  END;
```

```
FrameMsg = RECORD (Objects.ObjMsg)
  F: Frame;
  x, y: INTEGER;
  res: INTEGER
END;
```

The type *Display.Frame* assumes a specific part-of and coordinate system organization. The field *dsc* contains the head of the list of child frames connected by their *next* field. These pointers are the part-of references mentioned in section 4.3.1. Their presence in the frame type has several drawbacks. First, we assume that all frames have children, which not only forces a specific child organization, but also breaks container encapsulation when programmers directly access the *next* and *dsc* fields. In addition, this restricts a child to belong to a single parent. A further problem of having the organization fixed so low in the type hierarchy is that many type guards are required in container implementations with children consisting of only derived frame type (like gadgets). On the other hand, the presence of the two fields has made it possible to remain compatible with older Oberon applications and simplifies their portation to Oberon System 3. Besides, their absence would require an additional data structure for the management of children, which would ultimately cost memory. A contrasting view is used in ETHOS [Szy92b], another development of the Oberon system, where the child organization is strictly left to a container.

The fields *X*, *Y*, *W* and *H* are the display coordinates of the frame. The exact interpretation of these fields are left open to the parent frame. The Gadget system interprets the coordinates as relative to the parent frame, whereas many applications belonging to the TUI genre interpret the coordinates as absolute screen coordinates.

The *F* field specifies the message destination; a NIL indicates a broadcast to all frames. The *res* field is negative when the message is valid, otherwise it is *invalidated*. The *x* and *y* fields of *Display.FrameMsg* specify the display origin of the message receiver, and are used to calculate the absolute display coordinates of the message receiver by adding the relative *X*, *Y* coordinates (of type *Display.Frame*) to the message origin.

The Gadgets Frame Type. The visual gadget hierarchy is anchored by the type *Gadgets.Frame*. It is derived from type *Display.Frame* and extends the latter with common fields shared by all visual gadgets. The use of this base type has the advantage of a matching default message handler *Gadgets.framehandle* that can handle many messages in default manner.

```
TYPE
  Frame = POINTER TO FrameDesc;
  FrameDesc = RECORD (Display.FrameDesc)
    attr: Attributes.Attr;
    link: Links.Link;
    state: SET;
    mask: Display3.Mask;
    obj: Objects.Object;
  END;
```

The *attr* field refers to a data structure of attached attributes and the *link* field to a data structure with attached links. The *mask* field specifies the clipping mask used for displaying a gadget (cf. 6.4.3). The *obj* field contains a reference to the model gadget linked to the frame (if one exists).

The *state* field contains four boolean *flags* with information about the gadget state. The *selected* flag indicates if the gadget is selected, the *lockedsize* flag indicates if the width and height of the gadget is constant, the *lockedcontents* flag locks all the children of the gadget against further editing, and the *transparent* flag indicates if the gadget has a transparent background. The *selection* flag is controlled by the user by clicking with the right mouse button on a visual gadget. Once selected, the gadget shows itself in a white semi-translucent selection pattern. The *lockedcontents* flag is set with the "Locked" attribute of a container. The *lockedsize* and *transparent* flags are under programmer control and enable or disable behaviors related to displaying and interacting with a gadget.

6.2 Message Broadcasting and Forwarding

Message broadcasting is the term used for forwarding messages through the display space. It is initiated by a procedure called *Display.Broadcast*, which has approximately the following implementation:

```
PROCEDURE Broadcast(VAR M: Display.FrameMsg);
VAR f: Display.Frame;
BEGIN
  (* Initialize message fields *)
  Objects.Stamp(M); (* assign a unique time-stamp to M.stamp *)
  M.res := MIN(INTEGER); (* validate message *)
  M.x := 0; M.y := 0; (* set screen origin *)
  f := root.dsc; (* start at display root *)
  WHILE (f # NIL) & (f.res < 0) DO (* while not invalidated *)
    f.handle(f, M); f := f.next
  END
END Broadcast;
```

We now show in a general way how visual gadgets handle the frame messages (which are introduced in the following section). The handling of frame messages by an atomic gadget has the following pattern:

```
PROCEDURE Handle (F: Objects.Object; VAR M: Objects.ObjMsg);
VAR x, y, w, h: INTEGER;
BEGIN
  WITH F: Frame DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
        IF (M.F = NIL) OR (M.F = F) THEN
          x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;

          (* handle the frame messages here *)

        ELSE (* message not for me *)
        END
      END
    ELSE (* handle object messages *)
    END
  END
END Handle;
```

We save unnecessary work by testing the destination frame *F* as soon as possible for a true broadcast or a directed send to this gadget. The local variables *x*, *y*, *w* and *h* are set to the absolute display coordinates of the visual gadget for later use in the message handler.

Only a slight modification of the handler is necessary in the case of a container gadget. This involves changing the origin of the message before it is forwarded, and also forwarding all “unknown” messages. Note that a minor variation of the scheme below would also allow the *monitoring* of forwarded messages.

```
PROCEDURE Handle (F: Objects.Object; VAR M: Objects.ObjMsg);
VAR x, y, w, h: INTEGER;
BEGIN
  WITH F: Frame DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
        u := M.x; v := M.y;
        x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
        IF (M.F = NIL) OR (M.F = F) THEN
          (* handle the frame messages here *)
        ELSE
          M.x := x; M.y := y + h - 1; (* adapt origin *)
          ToChildren(F, M)
        END;
        (* restore origin *)
        M.x := u; M.y := v;
      END
```

```
    ELSE (* handle object messages *)
    END
  END
END Handle;
```

Message origin. The origin is modified as a frame message travels through the display space. In the example above, this happens before the call to *ToChildren*, a procedure implemented similar to *Display.Broadcast* to forward a message to all children. Child coordinates are relative to the top-left corner of the parent, hence the addition of the frame height to the current frame position.

Note that as the origin of display operations in the Oberon system is the bottom-left display corner, it means that the *Y* coordinate of a gadget must be *negative* for it to be visible in a container. Unfortunately the choice of a cartesian plane as a display coordinate system turns out counter-intuitive when frames are nested in each other. For example, with a left-bottom coordinate origin for the screen and nested gadgets, increasing the height of a container moves the children downward relative to the top edge instead of staying put where they are. Until the coordinate system is flipped in Oberon, an amount of coordinate paranoia will always be present, as Y coordinates must be corrected (in Gadgets we favor *M.x + F.X, M.y + F.Y* instead of *M.x + F.X, M.y - F.Y* which "flips" the direction of the Y axis).

Message thread. The *message thread* is the path the message followed through the display space to a certain frame. Its use includes gadgets behaving differently depending on their context and visibility calculations (cf. 6.4.3). *Context dependent behavior* must be explicitly programmed into gadgets by traversing the message thread; the system does not provide further support. If needed, a gadget can inherit behavior dynamically, for example, to react differently depending on the class of camera-view they are viewed with. This leads to the interesting effect of one and the same gadget, behaving differently at different viewed positions.

The thread is realized as a linked list of visited objects connected by the *dlink* field. The head of the list is the *dlink* field of the received *FrameMsg*. We illustrate this with the following code fragment that "visits" the frames on the message thread.

```
PROCEDURE FollowThread(VAR M: Display.FrameMsg);
VAR obj: Objects.Object;
BEGIN
  obj := M.dlink;
  WHILE obj # NIL DO
    ...
    obj := obj.dlink
  END
END FollowThread;
```

One use for such a loop is to detect if an ancestor gadget is locked (i.e. its *lockedcontents* flag is set), and accordingly deactivate part of a gadget's editing capabilities.

The construction of the message thread is simple. Before forwarding a message *M* to a child, we append the container *F* into the linked list. Note that this can only work when no recursive broadcasts occur (i.e. one broadcast that initiates another): the *dlink* field of objects is a global state that is overwritten by successive broadcasts. The solution is to save the *previous* value of *dlink* on the stack before forwarding the message. Combining this approach with the saving and restoring of the message *dlink* results in the following implementation of *ToChildren*. Note that a direct send from parent to a child has the overhead of at least six pointer assignments. Also, the global state of the message thread still rules out concurrent broadcasting in the display space, as one process would overwrite the message thread of another process.

```
PROCEDURE ToChildren(F: Frame; VAR M: Display.FrameMsg);
VAR f: Display.Frame; a, b: Objects.Object;
BEGIN
  a := F.dlink; b := M.dlink; (* save *)
  F.dlink := M.dlink; M.dlink := F; (* append self *)
  f := F.dsc;
  WHILE (f # NIL) & (M.res < 0) DO
    f.handle(f, M); f := f.next
  END;
  F.dlink := a; M.dlink := b (* restore *)
END ToChildren;
```

Message cycles. An inherent danger of message broadcasting is that of an endless forwarding cycle occuring. The technique for detecting cycles sketched in section 4.3.2 can be implemented in several ways, the most interesting being the exchange of the message handler of the parent gadget *P* for the short duration of the test.

```
TYPE
  RecursiveMsg = RECORD (Display.FrameMsg)
    flag: BOOLEAN
  END;

PROCEDURE RHandler(obj: Objects.Object; VAR M: Objects.ObjMsg);
BEGIN IF M IS RecursiveMsg THEN M(RecursiveMsg).flag := TRUE END;
END RHandler;

(* Does inserting the child f into the container F build a message cycle? *)
PROCEDURE Recursive(F, f: Objects.Object): BOOLEAN;
VAR old: Objects.Handler; M: RecursiveMsg;
BEGIN
  old := F.handle; F.handle := RHandler;

  M.flag := FALSE; f.handle(f, M);

  F.handle := old;
  RETURN M.flag;
END Recursive;
```

6.3 The Frame Messages

The frame messages and their interpretation form the heart of the Gadgets system. By convention, all frame messages are broadcasted through the whole or part of the display space. On receiving a frame message, a visual gadget knows its absolute display coordinates and the message path from the display root. As objects in Oberon are *passive* (compared to *active* ones that have an own thread of control), visual gadgets only obtain the latter information through a broadcast. This means that gadgets do not know where they are located on the display or in what context they appear, except when receiving a frame message.

On the one hand, this is an advantage as visual gadgets cannot obtain a global view of their state except in co-operation with their parents. By feeding a visual gadget with frame messages, the parent creates a “shell” around it which allows the gadget to operate correctly. As long as the shell works correctly, a gadget can be integrated into any environment, or even in more than one environment. On the other hand, the “hiding” of information from a gadget by its parent does not support applications that require continuous and fast access to coordinates and context, like for example, a video player gadget (or other examples of active objects)

The following sections present the definitions and semantics of an important sub-set of the frame messages. We will restrict our discussion to those frame messages applicable to the largest number of visual gadgets.

6.3.1 The Display Message

```
TYPE
  DisplayMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* frame, area *)
    u, v, w, h: INTEGER
  END;
```

The display message is a request to a visual gadget to display itself either completely (variant *frame*), or to display a rectangular area of itself (variant *area*). The latter message variant specifies the area in *u*, *v*, *w*, and *h*. The destination frame *F* is either set to a gadget or to NIL, which acts as a wild-card meaning all visual gadgets. This allows updating hierarchies of containers without modifying *F* during message forwarding.

Broadcasting a display message results in all visible instances of the destination frame being displayed (remember that the gadget may be visible through two or more camera-views). We accordingly say that the gadget receives the message once for each of its *display instances*.

The handling of the display message is illustrated by the following code fragment that is a part of many elementary gadgets.

```
PROCEDURE Restore (F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
BEGIN
  (* bounding box of gadget is x, y, w, h *)
  (* draw gadget contents through mask Q here *)
  IF Gadgets.selected IN F.state THEN (* gadget selected ? *)
    Display3.FillPattern(Q, Display3.white, Display3.selectpat,
      x, y, x, y, w, h, Display.paint)
  END
END Restore;

PROCEDURE Handle(F: Objects.Object; VAR M: Objects.ObjMsg);
VAR Q: Display3.Mask;
BEGIN
  ...
  IF M IS Display.DisplayMsg THEN
    WITH M: Display.DisplayMsg DO
      IF (M.id = Display.frame) OR (M.F = NIL) THEN
        Gadgets.MakeMask(F, x, y, M.dlink, Q);
        Restore(F, Q, x, y, w, h)
      ELSIF M.id = Display.area THEN
        Gadgets.MakeMask(F, x, y, M.dlink, Q);
        Display3.AdjustMask(Q, x + M.u, y + h - 1 + M.v, M.w, M.h);
        Restore(F, Q, x, y, w, h)
      END
    END
  END
  ...
END Handler;
```

The procedure *Gadgets.MakeMask* hides the calculation of the *display mask* through which a gadget must draw itself. Note that *MakeMask* is passed the absolute coordinates and the message thread, hence visual gadgets can request this calculation on receiving any frame message.

The area variant of the display message requires the reduction of the display mask to the area to be displayed (*Display3.AdjustMask*). The *Restore* procedure displays the contents of the gadget using the display primitives of module *Display3*. As explained in section 6.4.3, a pre-calculation of invariant parts of a gadget's display mask makes the implementation of *MakeMask* efficient, in the order of O(n) operations, where n is the number of camera-views in the message thread. The calculation of *AdjustMask* is efficient as it only involves setting a rectangular *clipping port* in the mask.

An opportunity for optimizing display is not issuing drawing primitives when they draw outside an update area, especially when many objects must be drawn. On the other hand, the *Display3* clipping routines are efficient enough to allow to do many display operations even if most pixels are clipped away. This can save coding at the cost of more clipping operations at run-time. This leads to the strategy of optimizing redraws in panels and text gadgets, and no optimizations in simpler elementary gadgets.

6.3.2 The Print Message

```
TYPE
  PrintMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* contents, view *)
    pageno: INTEGER; (* current page number *)
  END;
```

The print message variant *view* requests a gadget to print a *display approximation* or *snapshot* of itself on the printer. The print message variant *contents* requests a gadget to print its data contents. This can be a multi-page document, as for example in the case of a text document.

A module called *Printer3*, modeled along the lines of *Display3*, provides printing primitives that print clipped to a mask. A procedure *Gadgets.MakePrinterMask* calculates a printer mask for a gadget in a similar way as *Gadgets.MakeMask*. Visual gadgets are expected to make conversions from display coordinates to printer coordinates themselves. The only deviation from the display message handling is that the message origin is in printer coordinates and that frame coordinates have to be converted to printer coordinates from pixel coordinates.

Note that it is possible to introduce abstract output devices for display and printer, and so replace the display and print messages with a single "draw on device" message. Unfortunately, the display and printer drivers of Oberon are quite low-level and do not, for example, allow the arbitrary scaling of fonts. Hence, it was decided to work in either display or printer coordinates and to optimize the look of gadgets depending on the output device. This gives the best visual results as scaling from lower resolution display coordinates to higher resolution printer coordinates, and vice-versa, is not linear.

The approach has a drawback, due to the fact that the display and printer subsystems are not independent in the case of WYSIWYG documents. As display fonts are not a linearly scaled-down versions of printers fonts, different number of characters fit on the same line of a text document on the printer and on the display. A choice if a text document should look good on the display or on the printer must thus be made; the resulting documents often have different line breaks depending on the choice.

To reduce this problem, the text gadgets use a line breaking algorithm inspired by the Script text editor [Som94] that calculates the line lengths on display and printer concurrently, and inserts a line break at the position where the first overflow of either of these occur. The inter-word spacing is then adjusted differently according to the output device. Hence line breaks are correct but the inter-word spacings are different. This can lead to larger than normal inter-word spaces on the display when the printer font is much wider than the display font. The only way to correct such a document is not using the block-adjust formatting style and keep with left flush formatting. Note that the problem is also present for shorter text strings, like those used for text captions, and the caption lines of icons. This can lead to unexpected overlapping of printed captions with other elements in a diagram, even though no overlapping is seen on the display.

6.3.3 The Locate Message

```
TYPE
  LocateMsg = RECORD (Display.FrameMsg)
    loc: Frame;
    X, Y: INTEGER;
    u, v: INTEGER
  END;
```

The locate message determines what frame is located at absolute position *X*, *Y* on the display. The message is forwarded according to the *X*, *Y* coordinates. First the receiver checks if *X*, *Y* is located inside itself. If so, the message is forwarded to a child located at this position. If the latter does not respond, the receiver returns itself in the *loc* field. The fields *u* and *v* return the relative position of *X*, *Y* in frame *loc*. Note that a gadget might refuse to answer truthfully even though *X*, *Y* is located inside its bounding box. This will switch off the flow of input messages to the gadget, instead passing it to other gadgets that responded at the same position.

The locate message is handled in the following way by *Gadgets.framehandle*:

```
IF M IS Display.LocateMsg THEN
  WITH M: Display.LocateMsg DO
    IF (M.loc = NIL) & Effects.Inside(M.X, M.Y, x, y, w, h) THEN
      M.loc := F;
      M.u := M.X - x; M.v := M.Y - (y + h - 1);
      M.res := 0
    END
  END
```

6.3.4 The Input Message

```
TYPE
  InputMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* consume, track *)
    keys: SET;
    X, Y: INTEGER;
    ch: CHAR;
    fnt: Fonts.Font;
    col, voff: SHORTINT
  END
```

The input message delivers mouse (variant *track*) and keyboard events (variant *consume*) to the display space. The *keys*, *X* and *Y* fields specify the mouse buttons pressed and current mouse position. The fields *ch*, *fnt*, *col* and *voff* give details about the keyboard event.

The input message is a true broadcast terminated when the event is handled. The mouse events are forwarded by containers according to the mouse coordinates and in combination with responses to the locate message. Containers typically only forward the message if the child responds positively on the locate message. A parent may take control of the mouse when a child does not respond on the event.

The keyboard event travels through the display space until the *focused* frame *consumes* it. To prevent fighting over the keyboard input, only a single frame must have the focus. To obtain the focus, a gadget broadcasts an *Oberon.ControlMsg* to indicate that all other gadgets must relinquish the focus; afterwards the frame assumes that it owns the focus.

The default mouse behavior is conveniently handled by *Gadgets.framehandle*, which takes care of moving and resizing the gadget. The following procedure summarizes the possibilities.

```
PROCEDURE TrackFrame(F: Gadgets.Frame; VAR M: Oberon.InputMsg);
VAR x, y, w, h: INTEGER;
BEGIN
  IF (middle IN M.keys) & ~(selected IN F.state) & ~Locked(M.dlink) THEN
    x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
    IF Effects.InBorder(M.X, M.Y, x, y, w, h) THEN
      IF Effects.InCorner(M.X, M.Y, x, y, w, h) & ~(lockedsize IN F.state) THEN
        SizeFrame(F, M)
      ELSE
        MoveFrame(F, M)
      END
    END
  END
END TrackFrame;
```

Only the middle mouse button is handled by the default handler. Once selected, a gadget does not respond to mouse events; the parent takes control of these events. Furthermore, the parent must not be locked (procedure *Locked* inspects the *lockedcontents* flag of its argument). Nothing happens when the mouse is located inside the editing area. In the control area around the gadget (*InBorder*), the gadget is resized when the mouse is located in a corner, otherwise it is moved. Notice how the *lockedsize* flag allows the gadget programmer to switch off the resizing behavior. To complete the example, the procedures *MoveFrame* and *SizeFrame* are discussed later with the consume message and modify message. The handling is only slightly more complicated than the example in the actual implementation, where clicking inside the editing area can execute a *Cmd* attribute, and the movement of the mouse cursor must be done.

6.3.5 The Modify Message

```
TYPE
  ModifyMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* reduce, extend, move *)
    mode: INTEGER; (* display, state *)
    dX, dY, dW, dH: INTEGER;
    X, Y, W, H: INTEGER
  END;
```

The modify message informs the destination frame that it is to be translated or resized in its container. As visual gadgets edit themselves, it is often the case that the destination frame is also the one that broadcasted the modify message in the first place. According to the principle of parental control, the modify messages addressed to children are monitored by their containers. A container can refuse the modification (for example when the position is outside the container) or adjust the parameters (as in the case of a constraint solver). In addition, the parent has to take the responsibility of updating the previous location, and if necessary, adjust the visibility of other children.

The message fields of the modify message have the following meaning. The *id* field gives information about what type of modification is taking place: height increase, height decrease , or translation. The exact meaning of the *id* field is tied to the tiled viewer system, in which frames are only translated vertically. The gadgets have a loose interpretation of the field: *reduce* or *extend* means a change in width or height, perhaps coupled with a translation, and *move* means a translation without a size change.

The fields *X*, *Y*, *W*, *H* specify the new gadget location (in relative coordinates). The fields *dX*, *dY*, *dW* and *dH* indicate the change in coordinates from the original gadget location. The fields are required so that a gadget can determine both its old and new location. The *mode* field is used for optimizing the modification involving many gadgets, as for example when many gadgets are moved together. When *mode* is set to *display*, the destination frame will display itself immediately after adjusting its coordinates. When *mode* is set to *state*, the latter step is omitted. The idea of using this mode is that redrawing a container once is faster than drawing each of the children immediately, especially in the case of many children being modified. Note that by setting *mode* to *state*, the sender implicitly assumes the responsibility for redrawing the destination frame.

The default *Gadgets.framehandle* handles the modify message in the following way:

```
PROCEDURE Adjust(F: Display.Frame; VAR M: Display.ModifyMsg);
VAR D: Display.DisplayMsg;
BEGIN
  IF (F.X # M.X) OR (F.Y # M.Y) OR (F.W # M.W) OR (F.H # M.H) THEN
    (* first adjust *)
    F.X := M.X; F.Y := M.Y; F.W := M.W; F.H := M.H;
    (* invalidate F's mask here *)
  END;
  IF (M.mode = Display.display) & (F.H > 0) & (F.W > 0) THEN
    D.F := F; D.x := M.x; D.y := M.y;
    D.id := Display.frame; D.dlink := M.dlink;
    D.res := -1; Objects.Stamp(D);
    F.handle(F, D)
  END
END Adjust;
```

The imaging model requires that the display mask of the destination be *invalidated* when the gadget changes it size. This only needs to be done once (remember that the modify message can arrive multiple times at its destination). Eventually during the processing of the display message, a call to *MakeMask* causes the display mask to be recalculated. Interesting in the handling of the modify message is that each display instance receives the modify message and updates its representation itself.

Clearly, the default handling of the modify message is not efficient as everything is redrawn instead of parts being copied in the frame buffer. There are however two possibilities for optimization. Either we can defer optimization to the parent, which typically knows more about the layout of the child and its relationship to its environment, or we can let the gadget itself attempt the optimization. The latter is (partially) possible because the destination gadget can request its display mask both at the old and the new position, make an intersection of the area, and copy only the resulting visible areas from the old to the new position. The difference of the former mask and the new mask determines the 'missing" areas that still have to be redrawn. The technique is however not used in the Gadgets framework: it only works well with a single display instance, as a second display instance would have to invalidate the mask at the new location to (again) obtain the mask at the old location. A more serious obstacle is that, in the current implementation, masks do not completely specify what area of a gadget is visible. This is related to the handling of *transparent* gadgets (for more details see section 6.4.3).

The default handling of resizing a gadget illustrates how a client can make use of the modify message. In the example below, *Effects.SizeRect* does the visual feedback of spanning the resizing rectangle, and *Input.Mouse* determines the pressed mouse buttons and the pointer location.

```
PROCEDURE SizeFrame(F: Display.Frame; VAR M: Oberon.InputMsg);
VAR x, y, w, h, X, Y: INTEGER; keys: SET; A: Display.ModifyMsg;
BEGIN
  x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
  Input.Mouse(keys, X, Y);
  Effects.SizeRect(NIL, keys, X, Y, x, y, w, h, NIL);
  IF keys # cancel THEN
    A.id := Display.extend; A.mode := Display.display;
    A.X := x - M.x; A.Y := y - M.y; A.W := w; A.H := h;
    A.F := F; A.dX := A.X - F.X; A.dY := A.Y - F.Y;
    A.dW := A.W - F.W; A.dH := A.H - F.H;
    Display.Broadcast(A);
    M.res := 0
  ELSE M.res := 1
  END
END SizeFrame;
```

6.3.6 The Consume Message

```
TYPE
  ConsumeMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* drop, integrate *)
    u, v: INTEGER;
    obj: Objects.Object
  END;
```

The purpose of the consume message is to make structural changes in the display space. A drag and drop operation is translated into a consume message with variant *drop*, which requests the destination container to insert or consume gadgets. The *integrate* variant of the consume message inserts one or more gadgets into the focus frame (the frame that owns the keyboard focus).

The list of gadgets to be consumed is passed in the *obj* field; they are linked with each other with the *slink* field of *Objects.Object*. In the case of the *drop* variant, the insertion point in the destination frame is specified by *u*, *v*. In the case of the *integrate* variant, no destination frame is specified. Acceptance by a container of consumed gadgets is signified by invalidating the consume message. We also require that the destination container remove the consumed gadgets from any previous container they might have been in (cf. 6.3.9).

It is now possible to present the implementation of *MoveFrame*, as used by *Gadgets.framehandle*, which compactly illustrates the use the copy, modify, and consume message.

```
PROCEDURE MoveFrame(F: Display.Frame; VAR M: Oberon.InputMsg);
  VAR
    x, y, w, h: INTEGER;
    keys: SET; X, Y: INTEGER;
```

```
    A: Display.ModifyMsg; C: Display.ConsumeMsg; CM: Objects.CopyMsg;
    f: Display.Frame; u, v: INTEGER; old: Objects.Handler;
BEGIN M.res := 0;
  x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
  Input.Mouse(keys, X, Y);
  Effects.MoveRect(NIL, keys, X, Y, x, y, w, h);
  old := F.handle; F.handle := EmptyHandler;
  ThisFrame(X, Y, f, u, v);
  F.handle := old;
  IF keys = {middle, right} THEN (* copy frame *)
    IF f # NIL THEN
      CM.id := Objects.shallow; Objects.Stamp(CM);
      F.handle(F, CM);
      C.id := Display.drop; C.obj := CM.obj; C.F := f;
      C.u := u + (x - X); C.v := v + (y - Y);
      Display.Broadcast(C)
    END
  ELSIF keys = {middle, left} THEN (* consume frame *)
    IF f # NIL THEN
      C.id := Display.drop; C.obj := F; C.F := f;
      C.u := u + (x - X); C.v := v + (y - Y);
      Display.Broadcast(C)
    END
  ELSIF keys = {middle} THEN (* move frame *)
    A.F := F; A.id := Display.move; A.mode := Display.display;
    A.X := x - M.x; A.Y := y - M.y; A.W := w; A.H := h;
    A.dX := A.X - F.X; A.dY := A.Y - F.Y;
    A.dW := A.W - F.W; A.dH := A.H - F.H;
    Display.Broadcast(A)
  END
END MoveFrame;
```

Procedure *Effects.MoveRect* provides the visual feedback for moving a rectangular outline *x*, *y*, *w*, *h* (as in-out parameters). Procedure *ThisFrame* uses the locate message to determine what is located at the position *X*, *Y* where the mouse buttons are released. As frame *F* is being moved, and it is still embedded in its container, we do not want it to answer itself on the locate message. This is the reason why its message handler is temporarily exchanged with an empty implementation. The case of a move operation translates directly into a modify message. In the case of a drag and drop operation, we have to broadcast a consume message to the destination gadget. The statements $x - X$, respectively $y - Y$, make a correction depending on the position of the mouse inside the box *x*, *y*, *w*, *h*. The middle-right inter-click mouse combination is a request to consume a copy of the frame—a shallow copy is then made.

6.3.7 The Select Message

```
TYPE
  SelectMsg = RECORD (FrameMsg)
    id: INTEGER; (* get, set, reset *)
    time: LONGINT;
    sel: Frame;
    obj: Objects.Object
  END;
```

The select message comes in three variants: the *set* variant informs the destination gadget that it is being selected, the *reset* variant informs the destination gadget that it is being unselected, and the *get* variant returns the latest selection. The current gadget setting is remembered with the flag *selected* in the *state* field of *Gadgets.Frame*. The *get* variant is used in a true broadcast to return the latest gadget selection. As the message travels though the display space, the *time* field is updated to contain the time of the selection, the *obj* field the selected list of gadgets, and the *sel* field the container containing the selection. Each container compares its time of selection against *time* in the message, and updates *time*, *sel*, and *obj* should its own selection be newer.

The default handling of the select message by *Gadgets.framehandle* is the following. Note that containers typically interpret pressing the right mouse button as selecting a gadget.

```
IF M IS Display.SelectMsg THEN
  WITH M: Display.SelectMsg DO
    IF (M.id = Display.set) & (M.F = F) THEN
      INCL(F.state, selected); M.res := 0
    ELSIF (M.id = Display.reset) & (M.F = F) THEN
      EXCL(F.state, selected); M.res := 0
    END
  END
```

6.3.8 The Update Message

```
TYPE
  UpdateMsg = RECORD (Display.FrameMsg)
    obj: Objects.Object;
  END;
```

The update message forms the base type of change notification messages in the Gadgets framework. The field *obj* is the object whose state has changed. On receiving the update message, a view determines if its model is involved, and updates itself accordingly.

A further use of the update message is to display a set of visual gadgets headed by *obj* and linked together with the *slink* field. Such an operation is required as the semantics of the display message does not allow a list of destination frames to be specified—a "group update" of visual gadgets is required when the inspector changes attributes of the selection. Instead of having the default handler check if the gadget is in the update list, we expect containers to check if their children are found in the list. List searching is minimized with the invariant that the update list only contains children that have the same parent.

The level-1 programmer is the principal client of the update message. He or she must broadcast the update message *by hand* each time the state of a model is changed (if not integrated as part of the model update already). The update of gadgets is simplified by the utility procedure *Gadgets.Update* that, depending on the type of its argument, broadcasts either a display or an update message.

```
PROCEDURE Update(obj: Objects.Object);
VAR M: UpdateMsg; D: Display.DisplayMsg;
BEGIN
  IF obj IS Display.Frame THEN
    D.id := Display.frame;
    D.F := obj(Display.Frame);
    Display.Broadcast(D)
  ELSE
    M.obj := obj; M.F := NIL;
    Display.Broadcast(M)
  END
END Update;
```

6.3.9 The Control Message

```
TYPE
  ControlMsg = RECORD (FrameMsg)
    id: INTEGER; (* remove, suspend, restore *)
  END;
```

The control message unifies two unrelated functions in a single message. The *remove* variant removes the destination gadget from the display space. The *restore* and *suspend* variants are of an informational nature, as sketched below.

To remove many gadgets from the display space, the destination frame F is interpreted as the head of the list of *slink* connected frames to be removed. Notice that this is a break of style as the destination frame is interpreted as a list. Again, the invariant of the list of gadgets belonging to the same container can save the containers from unnecessary list searching. After removing the necessary children from its *dsc-next* list, the container updates its display representation.

The *restore* and *suspend* variants of the control message inform gadgets when they enter or leave the broadcasting area of the display space. As soon as a gadget leaves, it stops receiving message broadcasts. As containers can temporarily "hide" children away when they are not visible, and then later make them reachable by broadcast again, a gadget might have "missed" some important broadcasts. The *suspend* variant *warns* a gadget that it will not receive messages for a certain period of time. The *restore* variant informs the gadget that it is about to be reached by broadcast again, allowing it to re-synchronize with its model.

6.3.10 The Priority Message

```
TYPE
  PriorityMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* top, bottom, visible *)
    passon: BOOLEAN;
  END;
```

The priority message is a request to the destination frame to adjust its overlapping priority amongst its siblings in a container. Although addressed to the child itself, the message is only of interest to the destination frame's container, who monitors the message. The priority message variants indicate if the destination frame should be given the highest priority (*top*), the lowest priority (*bottom*), or if it should be adjusted to highest priority when it is partially overlapped by another child (*visible*).

The *passon* flag indicates if the priority adjustment should be applied to the container itself, and so on recursively, for all ancestors of the destination gadget. As containers do not know about their remote descendants, such an adjustment cannot be made when the message travels down through the display space. The destination's container is thus required to broadcast a recursive priority message for itself, again so for its parent, and so on. As can be expected, this is an expensive operation that should be avoided whenever possible.

6.3.11 Other Frame Messages

The previous sections introduced the most important frame messages as used in the standard Oberon distribution. Many applications add their own set of frame messages. There are however some additional messages that belong to the Oberon distribution whose semantics do not significantly contribute to our discussion, which we leave out from this discussion. For completeness sake, and to illustrate the idea of message families, they are only shortly mentioned.

The *Oberon.ControlMsg* controls the handling of the focus frame, marking frames with the star marker, and removing all marks like the selection, caret and marked frame. The family of messages related to text handling is distributed over modules *Oberon* and *Texts*. The *Texts.UpdateMsg* is the change notification message of the abstract data type text. The *Oberon.CaretMsg* and *Oberon.SelectMsg* control the setting and retrieving of the caret and the text selection in and from text editors. The *Oberon.ConsumeMsg* controls the copying over of text stretches between text editors. The *Oberon.RecallMsg* controls the recalling of the last deleted text stretch. Important regarding the family of text related messages is that they are not bound to any specific text editor. This allows the integration of cut, copy, and paste over different text editors.

A summary of the collection of frame messages belonging to the standard Oberon distribution is presented in Table 6.1. The table includes the messages from module *Display3*, which are discussed in the following section.

6.4 The Imaging Model

6.4.1 Motivation

Important criteria when picking an imaging model for an interactive system is the quality and efficiency of the displayed images. As the pixel sizes of raster output devices like the screen are relatively coarse grained, the presentation of images and figures often result in a loss of fidelity. The efficiency of an imaging model is determined by how fast images can be presented and how much memory is required in the process.

The quality and efficiency issue is addressed in the Gadgets system by using device coordinates and arbitrary clipping regions for all imaging primitives. Device coordinates, although often tedious to work with, have the advantage that a hand-tuned presentation of low resolution user interface components is possible. The use of arbitrary clipping regions for display primitives is motivated by the unwritten rules in the Oberon system to be both memory efficient and to avoid the first sin of the user interface programmer, namely *display flicker*.

Display flicker occurs when overlapping graphical components are displayed one after the other in quick succession during display update, as for example occurs when graphics are painted on the screen using the painter's algorithm [FvDFH91]. For example, drawing the background of a panel followed by drawing the children of the panel on top of it, results in many pixels being written twice: humans notice this as an irritating flash of the "wrong" color. When lots

Type	*Variants*
Display.ControlMsg	remove, suspend, restore
Display.ModifyMsg	reduce, extend, move
Display.DisplayMsg	frame, area
Display.PrintMsg	contents, view
Display.LocateMsg	
Display.SelectMsg	get, set, reset
Display.ConsumeMsg	drop, integrate
Texts.UpdateMsg	
Oberon.ControlMsg	defocus, neutralize, mark
Oberon.InputMsg	consume, track
Oberon.CaretMsg	get, set, reset
Oberon.SelectMsg	get, set, reset
Oberon.ConsumeMsg	drop, integrate
Oberon.RecallMsg	
Display3.OverlapMsg	
Display3.MakeMaskMsg	
Gadgets.PriorityMsg	top, bottom, visible
Gadgets.UpdateMsg	

Table 6.1: Frame message summary

of memory is available, the problem of display flicker is solved by drawing to an off-screen bitmap (which nobody sees) and then copying the result to the display. This is called *double buffering*, and it is both expensive in the number of pixels written in the off-screen bitmap and display, and the amount of additional memory required for the off-screen bitmap. As Oberon was designed to operate with very limited resources, the use of off-screen bitmaps was ruled out, and we were forced to adapt clipping regions, or as we refer to them, *clipping masks* or *display masks*.

A *clipping mask* is an abstract data-structure that indicates to the device driver where it may or may not write pixels. The simplest clipping mask is a rectangular area—it is like a rectangular stencil that passes through selected paint of the image. Rectangular clipping of primitives like rectangular block fills and block copies are easy and only involve a preprocessing of coordinates with a few tests and assignments; often the overhead is negligible in comparison to the amount of work done for the actual drawing.

Clipping can be much more efficient than double buffering, as each pixel clipped away (not drawn) saves at least one device memory access (and clipping away a small block of 10 by 10 pixels is already a huge saving). This assumes that many pixels can be clipped away, which is not always the case. For example, when displaying text on a colored background, using a clipping mask for drawing the background can be at least or even more inefficient than an off-screen bitmap. A trade-off is to use clipping masks for larger areas and double buffering for complicated smaller figures. If we take into account that the eye is more sensitive to flicker of larger areas than smaller ones, we might be willing to accept a little flicker, and thus get rid of double buffering completely. This is the motivation for picking a device-dependent imaging model based on clipping in the Gadgets system.

6.4.2 Shape Algebra

The progression from rectangular clipping to arbitrary region clipping is not so difficult as it might seem at first. What is required is a *shape algebra* for arbitrary regions or shapes [FvDFH91]. The module *Display3* provides operations to create a new shape, add or remove a rectangular area to or from a shape, intersect shapes with each other, subtract one shape from another, and to enumerate the form of a shape. Complicated figures like lines and splines are approximated by many smaller rectangles. The following paragraphs discuss the implementation of these operations.

A shape consists of a set of rectangles whose union determines the shape's

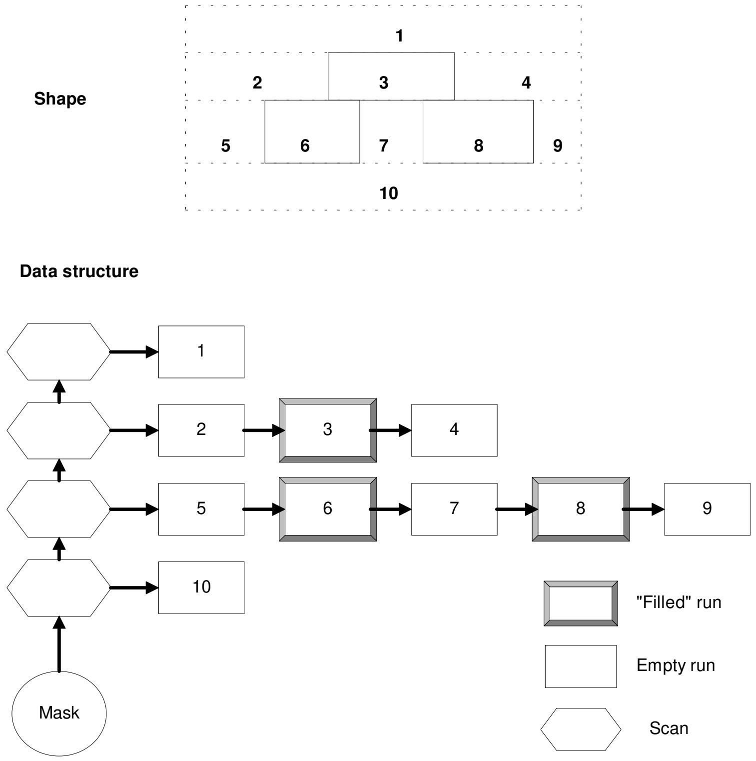

Figure 6.1: The shape structure

outline. For efficiency, rectangles are organized in such a way that they do not overlap. To quickly access a specific coordinate in the shape, the rectangles are collected in lists called *scans*. Rectangles in a scan are called *runs*, and are ordered by *X* coordinate. The runs of a scan have the same height but different widths. An open area between two rectangles in a scan is filled by an empty run, thus each run requires a flag specifying if it belongs to the shape or not. Scans are logically stacked upon each other and ordered according to their *Y* coordinate. The resulting scan and run structure of a simple shape is illustrated in Figure 6.1.

A straightforward translation of the scan and run structure results in the following data structure. We add an additional type *Mask* to make the structure opaque.

```
TYPE
  Run = POINTER TO RunDesc;
  RunDesc = RECORD
    next: Run;
    x, w, right: INTEGER;
    visible: BOOLEAN;
  END;

  Scan = POINTER TO ScanDesc;
  ScanDesc = RECORD
    next: Scan;
    y, h, top: INTEGER;
    run: Run;
  END;

  Mask = POINTER TO MaskDesc;
  MaskDesc = RECORD
    scan: Scan;
  END;
```

The scans are connected vertically with *next* in increasing *y* coordinate. The runs are connected horizontally with *next* in increasing *x* coordinate. The *w* and *h* fields specify the width of a run and the height of the scan. The *right* and *top* fields are present for convenience; they specify the right and top coordinates of a run and a scan.

We initialize the data structure in such a way that a single scan and run covers a very large surface, larger than any shape, so as to act as a sentinel.

```
PROCEDURE Init(M: Mask);
VAR s: Scan; r: Run;
BEGIN
  NEW(s); NEW(r);
  M.scan := s; s.run := r;
  s.y := -MAX; s.h := MAX * 2 + 1; s.top := MAX;
  r.x := -MAX; r.w := MAX * 2 + 1; r.right := MAX;
  r.visible := FALSE;
END Init;
```

Most shape operations are based on an algorithm to intersect a rectangle with a shape. This algorithm has the effect of cutting up the rectangle into smaller pieces according to the intersection with the shape boundary. As a block fill primitive has a rectangular area on the display, this intersecting algorithm is exactly what we need to draw a rectangular block clipped to a shape. Its implementation is as follows:

```
PROCEDURE Intersect(M: Mask; X, Y, W, H: INTEGER);
VAR s: Scan; r: Run; x, y, w, h, height, width: INTEGER;
BEGIN
  y := Y; s := M.scan;
  WHILE s.top < y DO s := s.next END;

  h := H;
  WHILE h > 0 DO
    height := Min(h, s.top - y + 1);
    x := X; r := s.run;
    WHILE r.right < x DO r := r.next END;

    w := W;
    WHILE w > 0 DO
      width := Min(w, r.right - x + 1);
      (* Output x, y, width, height *)
      DEC(w, width); INC(x, width);
      r := r.next
    END;

    DEC(h, height); INC(y, height);
    s := s.next
  END
END Intersect;
```

The algorithm delivers in the inner loop one sub-rectangle after another of the rectangle X, Y, W, H intersected with the shape. The flag *r.visible* indicates at this point if the rectangle is part of the shape or not. The algorithm consists of similar code for traversing the scans and the runs, except that they operate in different directions of the plane. First we have to locate the scan in which Y lies. The calculation then has to proceed for h until the whole rectangle is processed. We can calculate how much further to continue with the current scan by determining the distance to the top of the rectangle, and to the top of the scan. The minimum of both is called *height*, and is used to advance through the scans. A similar process takes place with the runs.

Profiling of the above code shows that most time is spent locating the scan and run in which *X*, *Y* lies. As long as the data structure does not change, we can imagine keeping a cache of this run between different invocations, which brings a speedup when consecutive accesses are in the same vicinity.

We will now extend the algorithm to insert or remove a rectangle from a shape. We require a procedure to split a scan horizontally at a specific *y* coordinate, and a procedure to split a run vertically at a specific *x* coordinate. Splitting a scan involves making an exact duplicate of its run structure and inserting the duplicate scan *after* the current scan. The exact implementations are left to the reader's imagination.

```
PROCEDURE SplitS(s: Scan; y: INTEGER);
PROCEDURE SplitR(r: Run; x: INTEGER);
```

The procedure *Flip* that follows toggles the *visibility* flag of rectangle *X*, *Y*, *W*, *H* to *state*. For each scan visited, we calculate if we might need to split the scan at the bottom of the rectangle ($y > s.y$) and at the top of the rectangle ($y + height <= s.top$). Note that if the rectangle lies completely inside a scan, we might have to split it twice.

```
PROCEDURE Flip(M: Mask; X, Y, W, H: INTEGER; state: BOOLEAN);
VAR s: Scan; r: Run; x, y, w, h, height, width: INTEGER;
  splitT, splitB: BOOLEAN;
BEGIN
  y := Y; s := M.scan;
  WHILE s.top < y DO s := s.next END;
  h := H;
  WHILE h > 0 DO
    height := Min(h, s.top - y + 1);
    splitB := y > s.y; splitT := y + height <= s.top;
    x := X; r := s.run;
    WHILE r.right < x DO r := r.next END;
    w := W;
    WHILE w > 0 DO
      width := Min(w, r.right - x + 1);
      IF r.visible # state THEN
        IF splitT THEN
          SplitS(s, y + h); splitT := FALSE
        END;
        IF splitB THEN
          SplitS(s, y); splitB := FALSE;
```

```
        x := X; s := s.next; r := s.run;
        WHILE r.right < x DO r := r.next END;
      END;
      IF x > r.x THEN (* split left *)
        SplitR(r, x);
        width := 0
      ELSIF x + width <= r.right THEN (* split right *)
        SplitR(r, x + width);
        r.visible := state
      ELSE r.visible := state
      END
    END;
    DEC(w, width); INC(x, width);
    r := r.next
  END;
  DEC(h, height); INC(y, height);
  s := s.next
 END
END Flip;
```

The modification of our previous intersection algorithm involves the case when the span is not in the state we require—only then we need to split the scan. To prevent splitting for each run visited, we reset the split flags after the first split. Note that after splitting at the bottom, we have to advance to the next scan, and thus repeat the search for the correct run. Once the scan has been split, we might have to split the run on the left or on the right. Also note that when $x > r.x$, the following pass through the loop will modify the state of the following (and new) run.

The drawback of this implementation is that we keep on splitting scans and runs, which eventually results in a deterioration of the structure when many adds and subtracts are made. One possibility is to make a pass over the data structure and merge neigboring runs that have the same state, and neighboring scan lines that have the same runs. Such a *compact* operator can be called periodically to improve the structure. Another possibility is to attempt to optimize the structure as it is being modified. The opportunities for optimization occur when the state of *r.visible* is changed. We imagine replacing the assignments to *r.visible* above by a procedure *FlipRun* that changes the state and does some optimizations:

```
PROCEDURE FlipRun(VAR r: Run; state: BOOLEAN);
```

The run is declared as VAR parameter as the run might be optimized "away". One implementation of *FlipRun* involves introducing a *previous* pointer in the run structure, connecting runs in the opposite direction too. This makes it easier to determine if the previous run can be merged with the current one. The next pointer is used to check if we can merge with the following run. As *r* is changed during the optimization, we adapt the advancement of the pointer at the end of the loop to:

```
IF x > r.right THEN r := r.next END
```

A further low-cost improvement is to merge the runs of the new scan in *SplitS*. Simple optimizations like these keep the runs in good shape but do not merge identical scans, so a second pass to merge scans is still required. The *Display3* module also optimizes the case when the shape is a single rectangle, as this case appears so often. There are several other opportunities for optimization, including dropping one of the *x* or *right* fields, and a more efficient packing of runs in memory.

Based on similar algorithms, it is for example possible to derive an intersection with a rectangle operator. Union and difference of shapes are easy by incrementally adding or removing areas. Enthusiasm for the approach should however be kept in bounds; the performance deteriorates with a multitude of scans a single pixel high, like when geometric figures like lines and circles are defined as shapes.

6.4.3 Display Masks

Display masks are the clipping regions through which a visual gadget draws itself. Each and every visual gadget has its own display mask cached in its *mask* field. It consists of a shape that specifies which parts of the gadget are visible. The module *Display3* provides a collection of drawing primitives that operate with masks, just as module *Printer3* provides a similar collection for printing.

As the calculation of a display mask can be an expensive process, the Gadgets framework adopts a *demand-driven* approach for generating masks. At any one instance, a gadget either has a valid display mask, or it has no mask at all. When a modification is applied to a gadgets shape, its mask is dropped by a process called *invalidating*. Only when a visual gadget wants to draw itself, it requests a new display mask by calling a procedure *Gadgets.MakeMask* to activate the mask generation process. This strategy means that a gadget can operate for long periods of times without a valid mask, at least until it wants to draw something on the display.

To simplify our initial discussion, we can assume that we would like to calculate the masks for a sub-section of the display space that does not contain any camera-view gadgets. The situation can be imagined as overlaying the whole display with a large container gadget. This root container is completely visible, hence its display mask is a single rectangle. From its own display mask, the container is expected to generate masks for its children. The following procedure shows that generating masks for children involves intersecting the container's mask with the bounding box of the child, and clipping away all the overlapping parts of higher priority children.

```
PROCEDURE BuildMasks(F: Display.Frame);
VAR f, g: Display.Frame; R: Display3.Mask;
BEGIN
  f := F.dsc;
  WHILE f # NIL DO
    Display3.Copy(F.mask, R); (* copy the mask *)
    Display3.Intersect(R, f.X, f.Y, f.W, f.H);
    g := f.next;
    WHILE g # NIL DO
      IF Effects.Intersect(f.X, f.Y, f.W, f.H, g.X, g.Y, g.W, g.H) THEN
        Display3.Subtract(R, g.X, g.Y, g.W, g.H)
      END;
      o := o.next;
    END;
    (* Set mask of f to R here *)
    f := f.next
  END
END BuildMasks;
```

For n children, this algorithm requires

$$\frac{n(n-1)}{2}$$

bounding box comparisons, which is a run-time complexity of $O(n^2)$. Luckily the average situation is not so bad as children do not always overlap each other and thus the expensive *Display3.Subtract* is not always invoked. Note that we are working in relative coordinates; all masks are specified in the relative coordinates of the gadget itself, and thus save us from recalculating masks when moving a container around. Also note that setting the mask of *f* involves a coordinate translation of the mask from parent to child coordinates (which is not shown here).

We can now complete the mask implementation by adding a *translation vector* and a *clipping port*.

Using relative mask coordinates requires a way to position a mask at any location of the display. For this purpose, a translation vector *x*, *y* is added to the type *Display.Mask*. The efficient translation is possible by modifying our shape algebra with coordinate accumulation during scan and run traversal. A mask is automatically translated to the current gadget position when *Gadgets.MakeMask* is called (cf. 6.3.1).

The handling of the display message dictates that we sometimes only redisplay a rectangular sub-part of a gadget (cf. 6.3.1). To keep this a light-weight operation that does not modify the shape of the display mask, each display mask has a *clipping port*. The clipping port is a rectangular clipping area in addition to the shape itself. Display primitives are first clipped to the clipping port and then to the shape. As the final definition of the type *Display3.Mask* shows below, the clipping port is specified by four INTEGERs that can be easily changed without modifying the mask shape[1]. We can also indirectly modify the clipping port using a call to a utility procedure *Display3.AdjustMask*, as was shown earlier in the handler of the display message (cf. 6.3.1).

```
TYPE
  Mask = POINTER TO MaskDesc;
  MaskDesc = RECORD
    x, y: INTEGER;
    X, Y, W, H: INTEGER;
  END;
```

The mask protocol. *Mask calculation* is a service a container provides for its children. A child can request its parent to quickly manufacture a mask for it, should it discover on displaying that it has no valid mask. The module *Display3* defines the messages *OverlapMsg* and *UpdateMaskMsg* for this purpose.

```
OverlapMsg = RECORD (Display.FrameMsg)
  M: Mask;
END;

UpdateMaskMsg = RECORD (Display.FrameMsg)
END;
```

The *OverlapMsg* is sent directly from parent to child, informing the latter that its new mask is *M*. A NIL value of *M* indicates that the child's existing mask is invalid. The default handling of the *OverlapMsg* in *Gadgets.framehandle* involves setting the *mask* field of the visual gadget to *M*. In the case of container gadgets, the *Overlap* message activates the mask generation for its children, as sketched earlier.

A call to *Gadgets.MakeMask* sometimes finds a gadget without a valid mask. Then the *UpdateMaskMsg*, with destination set to the gadget, is broadcast into the display space. All containers monitor the *UpdateMaskMsg*, checking if it

[1]The porting of Gadgets to the Macintosh identified a design oversight regarding these fields. As the fields are not procedurally encapsulated, a field modification is not directly detectable, which makes it slightly more difficult to map the shapes into the MacOS region primitives.

is addressed to one of their children. Should one of their own children be involved, its mask is calculated and updated with the *OverlapMsg*. In the case of *Gadgets.MakeMask* finding a gadget with a valid mask, nothing needs to be done except to return the cached mask to the caller.

To prevent numerous *UpdateMaskMsg*'s from being broadcast when many children have invalid masks, a container automatically recalculates all invalid masks of its children when the first *UpdateMaskMsg* arrives. This is under the assumption that if one child requires a mask, the others will do as well in the near future.

Masks and camera-views. The mask protocol, as sketched in its simplistic form above, only works correctly when no camera-view gadgets are in the display space. An interesting complication is when a gadget is viewed by two or more camera-views, with potentially different parts visible. Let's trace how the display is redrawn to illustrate the solution to the problem. Drawing a container involves drawing the container itself, followed by sending a *display* message to each of its children, and so on recursively. The situation with camera-views is the same except that the display message is forwarded to the visual model instead. As many camera-views might have the same model, the latter will receive the message once from each of its camera-views. It is clear that the visibility of the gadget is determined by the path that the message followed to reach it. The procedure *Gadgets.MakeMask* provides a means to efficiently calculate a *clipping mask* from the latter, through which a gadget can then display itself.

First, we define that a camera-view creates a clipping mask for its (visual) model without regard to its own clipping mask, i.e., that the model has a clipping mask as if it was completely visible. The actual clipping mask is then the combination of a gadget's cached mask, intersected with the masks of all the camera-views found on the message path. Conceptually, the procedure *Gadgets.MakeMask* would need to make a copy of the cached mask and intersect it with the masks of all the camera-views on the message thread.

The intersection operation requires the absolute screen position of the camera-views in the message thread (remember that the camera-view mask is relative too). For this purpose, a base type for camera-view gadgets called *Gadgets.View* is defined. The fields *absX*, *absY* of camera-view instances are set to the camera-view display coordinates each time a frame message is forwarded to its model.

```
View = POINTER TO ViewDesc;
ViewDesc = RECORD (FrameDesc)
  absX, absY: INTEGER;
  ClipMask: PROCEDURE (v: View; M: Display3.Mask);
END;
```

As the intersection operation is more expensive than the adjustment of the clipping port, the implementation of *Gadgets.MakeMask* attempts to reuse the cached mask instead of creating a new mask for each intersection. The following procedure sketches the approximate implementation:

```
PROCEDURE MakeMask(F: Frame; X, Y: INTEGER; dlink: Objects.Object;
          VAR M: Display3.Mask);
VAR
  U: Display3.UpdateMaskMsg; f: Objects.Object;
  x, y, w, h: INTEGER;
  R: Display3.Mask;
BEGIN
  IF F.mask = NIL THEN (* invalid mask *)
    U.F := F; Display.Broadcast(U);
    ASSERT(F.mask # NIL)
  END;

  M := F.mask; (* cached mask *)

  (* translate mask to display coordinates *)
  M.x := X; M.y := Y + F.H -1;

  (* default clipping port *)
  M.X := X; M.Y := Y; M.W := F.W; M.H := F.H;

  v := dlink; (* visit camera-views *)
  WHILE v # NIL DO
    IF v IS View THEN
      WITH v: View DO
        (* position camera-view mask *)
        v.mask.x := v.absX; v.mask.y := v.absY + v.H + 1;

        IF Display3.Rectangular(v.mask, x, y, w, h) THEN
          Display3.AdjustMask(M, x, y, w, h)
        ELSE
          Display3.IntersectMasks(M, v.mask, R);

          (* copy clipping port *)
          R.X := M.X; R.Y := M.Y; R.W := M.W; R.H := M.H;
          M := R
        END
      END
    END;
    v := v.dlink
  END
END MakeMask;
```

Note how the fields *absX*, *absY* are used to position the mask of the camera-view before the intersection. The procedure *Display3.Rectangular* returns TRUE if a mask is a single rectangular shape, returning its form in *x*, *y*, *w*, *h*. The procedure *Display3.IntersectMasks* intersects two masks and delivers a third mask as a result.

The implementation of *Gadgets.MakeMask* illustrates that the mask calculation process is optimized for the case when camera-views are completely visible (in which case their mask is rectangular). When partially overlapped, a performance penalty is paid as the cached mask has to be copied and then modified. The latter case occurs more often in the overlapping viewer model of desktops.

The only situation that we have not covered so far is when the camera-view itself clips away parts of its visual model. For this case, the procedure variable *ClipMask* is provided, which is filled-in by the camera-view implementation with a procedure that intersects the mask with the visible area. We only require an additional call *v.ClipMask(v, M)* in the loop to handle the situation (not shown above).

Transparent visual gadgets. The procedure *BuildMasks* presented earlier assumes that all children are rectangular in shape, i.e. each visual gadget completely fills its bounding box. It is however useful to have irregular shaped visual gadgets too (like for example the icon gadgets on the desktop in Figure 3.11). These irregularly shaped visual gadgets are called *transparent* gadgets because sub-sections of their bounding box are transparent and show what lies below them (i.e. parts that are below "shine through").

On the one hand, transparent gadgets are useful for making geometric figures first-class citizens, but on the other hand they complicate the imaging model. Conceptually, a transparent gadget must cut out its shape from the masks of the gadgets that lie below it. In cases like the icons, their rectangular nature makes this not too problematic. In the extreme though, a transparent text caption might have to cut out its character form from the masks below. If we are willing to pay for this exact implementation of the imaging model, we can imagine a third message in the mask protocol that requests a gadget for a mask specifying its exact form. The latter can then be used in the subtract operation in procedure *BuildMasks*.

As we had our doubts that this implementation would be efficient enough and also implementable in the current Oberon system, a simpler approach was selected for handling transparent gadgets. It is based on the observation that parents can monitor the display messages broadcasted to their children. In the case of transparent gadgets, the parent can apply the painters' algorithm in the area of the child. First the parent draws its own background, followed by the gadget itself, followed by all transparent gadgets that lie on top of it. To aid in the process, the containers identify transparent gadgets by the *Gadgets.transparent* flag in the *state* field. It is the task of the programmer to set this flag correctly. Furthermore, the build mask procedure is modified in such a way that only non-transparent gadgets are subtracted from a gadget's mask. This means that an opaque gadget can influence the mask of any other lower priority gadget (transparent or opaque), whereas a transparent gadget cannot.

The advantage of this approach is that transparent gadgets are first-class citizens. The disadvantage is that containers must be programmed specially to handle this case. Also, because transparent gadgets do not affect the masks of other gadgets, a gadget cannot be 100% sure that the mask it obtains with *Gadgets.MakeMask* is its true visible area. This hampers optimizations involving copying pixel blocks on the display.

As the author sees it, the only clean solutions to this problem are either to ban transparent gadgets, or to pay the penalty of calculating the exact shape of each and every gadget. The latter option seems the best, but requires, amongst others, a way to turn a font into a collection of shapes, an operation that was not foreseen in the design of the Oberon system.

6.5 Examples

The following sections present a selection of examples related to messages and gadget programming.

6.5.1 Messages

Although messages seem simple by themselves, their combination can be arbitrarily complicated and revealing at the same time. A good example of how messages are combined with each other is tracing the message flow during a drag and drop operation. In the following example, we assume that visual gadget *G* located in panel *P* is picked up with the mouse, and moved to panel *Q*.

The major steps are the following:

1. The Oberon *loop* notices that the mouse pointer has moved or that a mouse button has been pressed or released, and broadcasts into the display space the input message (cf. 6.3.4), variant track, with the current mouse position.

2. The input message is forwarded from container to children depending on the mouse coordinates. The locate message (cf. 6.3.3) determines if a gadget is interested in the message or not (it is sent from parent to child *before* the input message is forwarded). The first notice of the impending arrival of an input message at *G* is thus the arrival of a locate message.

3. The input message arrives at *G*. *G* detects that the middle mouse button is pressed inside the move control area. The default handler provides the visual feedback of a rectangle moving across the display until all the mouse buttons are released.

4. *G* determines the position where the mouse was released by broadcasting a locate message.

5. Panel *Q* receives the locate message and returns itself.

6. Having determined the destination *Q*, gadget *G* broadcasts a consume message (cf. 6.3.6) with *Q* as destination and itself as object to be consumed.

7. *Q* receives the consume message and decides that it is willing to accept the new gadget *G*. Before *G* can be accepted, it must be made an orphan for a short period of time. This is done by broadcasting a control message, variant remove, addressed to *G*.

8. *P* receives the control message and notices that its child *G* is to be removed. It removes *G* from its *dsc-next* list. Furthermore, all the children lying below *G* are sent a *Display3.OverlapMsg* to invalidate their masks. *P* broadcasts a display message to redraw its modified parts.

9. When updating parts of *P*, some of the underlying children notice that they have no valid mask. These children broadcast an *UpdateMaskMsg* to their parent, which calculates the new masks and forward them to the children using the *Display3.OverlapMsg*. The children send a *DisplayMsg* directrly to themselves.

10. *Q* inserts *G* into its own *dsc-next* list as a new child. Then *G*'s mask is invalidated with a *Display3.OverlapMsg*. The children's masks below *G* are invalidated.

11. *Q* broadcasts a display message to update the area that the child *G* now occupies.

12. *G* receives the display message and requests that its mask be calculated. As it has an invalid mask, a *Display3.UpdateMaskMsg* is broadcast.

13. *Q* receives the *Display3.UpdateMaskMsg* and realizes that *G* requires a new mask. Procedure *BuildMask* (cf. 6.4.3) is invoked to do the mask calculation, and the *Display3.OverlapMsg* informs *G* of its new mask.

14. Gadget *G*, now with valid mask, displays itself.

15. Control returns to *Q* that invalidates the consume message.

16. Control returns to *G* that invalidates the input message.

Another interesting example is how to separate model and view using a message protocol as data exchange mechanism. Lets look at the simplified situation of a textfield gadget *T* displaying the value of an integer model gadget *I*:

1. The user clicks with the left mouse button to set the focus in the textfield gadget. The Oberon event loop broadcasts an input message of variant track specifying the left mouse button. It arrives (in the manner sketched before) at *T*.

2. *T* discovers that it does not have the keyboard input focus yet, and broadcasts an *Oberon.ControlMsg* to force other gadgets to relinquish it. *T*, assuming that it has obtained the keyboard focus, sets an internal flag saying that it has the caret, and broadcasts a message to display itself. On receiving the display message, it draws the newly set caret.

3. The user types something on the keyboard. The Oberon event loop broadcasts an input message with variant consume into the display space. It arrives at *T*, who consumes the message and inserts the character in an internal buffer of typed characters. *T* broadcasts a display message to itself, and so displays the contents of the buffer on the screen.

4. The user presses RETURN to signify that the model and view need to be updated. The event arrives at *T* as yet another input message. *T* takes its internal character buffer and packs it into an attribute message. The attribute message in variant *set* is sent directly from *T* to *I*.

5. *I* receives the attribute message, does a conversion from string format to integer, and updates its own state.

6. *T*, who has just changed the state of *I*, broadcasts an update message to say that the value of *I* has changed.

7. *T* receives the update message and notices that its model has changed. *T* sends an attribute message, variant *get*, to *I* to retrieve the current state of *I*. As *I* returns an integer, *T* does the necessary conversion to string, copies it into its internal buffer, and redisplays itself immediately.

Note that in this protocol both the model and view have representations of the same data in different formats, namely integer and string respectively, which they synchronize using the attribute message. Consequently, *T* and *I* make now assumptions about each other except that they understand the update and attribute message. This is an important criteria for truely pluggable components.

6.5.2 Elementary Gadgets

Programming an elementary gadget is a matter of combining the code fragments presented in this and in the previous chapter. As most of the message handling is taken over by the default message handler, there is little left to do but to specify how the gadget should display itself and how it interacts with the user. The following example code implements a colored block that can be moved and resized. Clicking on the block executes a command. The block has a *Color* and *Cmd* attribute. As with the component example in section 5.6, the differences from a predefined code skeleton are shown in bold.

```
MODULE Blocks;
  IMPORT Files, Display, Display3, Printer, Printer3, Effects, Objects, Gadgets, Oberon;

  TYPE
    Frame* = POINTER TO FrameDesc;
    FrameDesc* = RECORD (Gadgets.FrameDesc)
      col*: INTEGER;
      cmd*: ARRAY 64 OF CHAR;
    END;

  PROCEDURE Attributes (F: Frame; VAR M: Objects.AttrMsg);
  BEGIN
    IF M.id = Objects.get THEN
      IF M.name = "Gen" THEN
        M.class := Objects.String; COPY("Blocks.New", M.s);
        M.res := 0
      ELSIF M.name = "Color" THEN
        M.class := Objects.Int; M.i := F.col;
        M.res := 0
      ELSIF M.name = "Cmd" THEN
```

```
      M.class := Objects.String; COPY(F.cmd, M.s);
      M.res := 0;
    ELSE Gadgets.framehandle(F, M)
    END
  ELSIF M.id = Objects.set THEN
    IF M.name = "Color" THEN
      IF M.class = Objects.Int THEN
        F.col := SHORT(M.i);
        M.res := 0
      END
    ELSIF M.name = "Cmd" THEN
      IF M.class = Objects.String THEN
        COPY(M.s, F.cmd);
        M.res := 0
      END
    ELSE Gadgets.framehandle(F, M);
    END
  ELSIF M.id = Objects.enum THEN
    M.Enum("Color"); M.Enum("Cmd");
    Gadgets.framehandle(F, M)
  END
END Attributes;

PROCEDURE Restore (F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
BEGIN
  Display3.ReplConst(Q, F.col, x, y, w, h, Display.replace);

  IF Gadgets.selected IN F.state THEN
    Display3.FillPattern(Q, Display3.white, Display3.selectpat,
      x, y, x, y, w, h, Display.paint)
  END
END Restore;

PROCEDURE Copy(VAR M: Objects.CopyMsg; from, to: Frame);
BEGIN to.col := from.col; Gadgets.CopyFrame(M, from, to)
END Copy;

PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y, w, h: INTEGER; F1: Frame; Q: Display3.Mask;
BEGIN
  WITH F: Frame DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
        IF (M.F = NIL) OR (M.F = F) THEN
          x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
          IF M IS Display.DisplayMsg THEN
            WITH M: Display.DisplayMsg DO
              IF (M.id = Display.frame) OR (M.F = NIL) THEN
                Gadgets.MakeMask(F, x, y, M.dlink, Q);
                Restore(F, Q, x, y, w, h)
              ELSIF M.id = Display.area THEN
                Gadgets.MakeMask(F, x, y, M.dlink, Q);
                Display3.AdjustMask(Q, x+M.u, y+h-1+M.v, M.w, M.h);
```

```
            Restore(F, Q, x, y, w, h)
          END
        END
      ELSIF M IS Oberon.InputMsg THEN
        WITH M: Oberon.InputMsg DO
          IF (M.id = Oberon.track) & Gadgets.InActiveArea(F, M) THEN
            REPEAT
              Effects.TrackMouse(M.keys, M.X, M.Y, Effects.PointHand);
            UNTIL M.keys = {};
            Gadgets.Execute(F.cmd, F, M.dlink, NIL, NIL)
            M.res := 0
          ELSE Gadgets.framehandle(F, M)
          END
        END
      ELSE Gadgets.framehandle(F, M)
      END
    END
  END
  ELSIF M IS Objects.AttrMsg THEN
    WITH M: Objects.AttrMSG DO
      Attributes(F, M)
    END
  ELSIF M IS Objects.FileMsg THEN
    WITH M: Objects.FileMsg DO
      IF M.id = Objects.store THEN
        Files.WriteInt(M.R, F.col); Gadgets.framehandle(F, M)
      ELSIF M.id = Objects.load THEN
        Files.ReadInt(M.R, F.col); Gadgets.framehandle(F, M)
      END
    END
  ELSIF M IS Objects.CopyMsg THEN
    WITH M: Objects.CopyMsg DO
      IF M.stamp = F.stamp THEN M.obj := F.dlink
      ELSE
        NEW(F1); F.stamp := M.stamp; F.dlink := F1;
        Copy(M, F, F1); M.obj := F1
      END
    END
  ELSE Gadgets.framehandle(F, M)
  END
 END
END Handle;

PROCEDURE Init* (F: Frame);
BEGIN F.W := 50; F.H := 50; F.col := 1; F.handle := Handle
END Init;

PROCEDURE New*;
VAR F: Frame;
BEGIN NEW(F); Init(F); Objects.NewObj := F
END New;

END Blocks.
```

The only aspect of the code that has not been discussed in one form or another before, is that of handling the input message. The procedure *Gadgets.InActiveArea* returns true if the mouse is located inside the gadget area reserved for use (cf. 3.4.1). The repeated calls to *Effects.TrackMouse* are to draw the mouse pointer. Procedure *Gadgets.Execute* executes a command. The other parameters passed include the object executing the command and the current context, both of which are used by the callee. Note that mouse tracking is not event-oriented—gadgets take direct control of the mouse after receiving the input event.

6.5.3 Container Gadgets

The duties of containers include displaying children, calculating masks, consuming and removing children, handling child modification, and forwarding of messages. Most of these duties are simple to program and will not be repeated here. There are however some aspects of containers that are of interest. In the following discussions, examples from a panel-like container are used.

Perhaps the most interesting aspect of panels is the way in which the modify message (cf. 6.3.5) is implemented. To repeat shortly, each display instance of a panel receives a *Display.ModifyMsg* for a child that is being adjusted. The default handling of the modify message (by the gadget itself) is to invalidate the gadget's mask, change its coordinates, and send a display message to itself. As the modified gadget displays itself, the only duty of the panels is to redisplay the area where the gadget used to be before.

We require the following procedures, the implementation of which is left to the imagination.

```
PROCEDURE InvalidateMasks(F, f: Display.Frame; x, y, w, h: INTEGER);
PROCEDURE ToChild(F, f: Display.Frame; VAR M: Display.FrameMsg);
PROCEDURE RestoreRegion(F: Display.Frame; x, y: INTEGER;
        dlink: Objects.Object; R: Display3.Mask);
```

Procedure *InvalidateMasks* invalidates the masks of *F*'s children that intersect with the area *x*, *y*, *w*, *h*, except for the specific child *f*. Procedure *ToChild* forwards the message *M* from *F* to *f*. Procedure *RestoreRegion* restores the area *R* of the panel *F*.

The panel handler is programmed in a way to call the following procedure each time it receives a modify message for one of its children. For the purpose of illustration, I simplified the implementation to only handling the modify message with *mode* set to *display*, without handling of transparent gadgets, and without optimizations.

```
PROCEDURE AdjustChild(F: Display.Frame; VAR M: Display.ModifyMsg);
VAR
  x, y: INTEGER;
  nx, ny, nw, nh, ox, oy, ow, oh: INTEGER;
  R: Display3.Mask;
BEGIN
  x := M.x + F.X; y := M.y + F.Y;
  nx := M.X; ny := M.Y; nw := M.W; nh := M.H;
  ox := M.X - M.dX; oy := M.Y - M.dY;
  ow := M.W - M.dW; oh := M.H - M.dH;

  IF M.stamp = F.stamp THEN
    InvalidateMasks(F, M.F, ox, oy, ow, oh);
    InvalidateMasks(F, M.F, nx, ny, nw, nh)
  END;
  F.stamp := M.stamp;

  ToChild(F, M.F, M);

  NEW(R);
  Display3.Open(R);
  Display3.Add(R, ox, oy, ow, oh);
  Display3.Subtract(R, nx, ny, nw, nh);
  RestoreRegion(F, x, y, M.dlink, R)
END AdjustChild;
```

The local variables *nx*, *ny*, *nw*, *nh* and *ox*, *oy*, *ow*, *oh* are initialized to the new and old relative position of the gadget. Note that we have to use the delta values of the message to calculate the old position of the gadget, as the child would have already adjusted its own coordinates when the message is received the second or more time around.

The first arrival of the modify message requires that we invalidate the masks of the gadgets that intersect the old and the new position of the child. There is no need to invalidate the mask of the child itself; it does so itself. The next step is to forward the message to the child. This results in its coordinates being updated and mask invalidated the first time around, followed by the child displaying itself (often left to the default handler). Afterwards, we have to build the difference of the old and the new location to determine the area that must be redisplayed. Note that *RestoreRegion* directly draws the correct area, once for each modify message received at each display instance. The handling of the modify message thus requires at least the broadcast of the modify message, and another broadcast by the child to update its mask (*Display3.UpdateMaskMsg*, cf. 6.4.3). As the panels recalculate all invalidated masks in one pass on receiving the latter message, at most two broadcasts are involved.

The actual implementation is complicated by the handling of the modify messages with *mode* set to *state*. This setting suppresses the automatic redisplay of a gadget after a modification. It thus becomes the duty of the container to keep track of areas that have to be redrawn. The simplest approach is that on noting the *dirty rectangle*, which is the smallest enclosing rectangle that includes all the areas that must be redrawn. The special handling of transparent gadgets is another complication, that involves sending a display message (variant *area*) to all transparent gadgets located above the new location of the child gadget. Also, one optimization not shown above, is to separate the case when the gadget's previous and new locations overlap or not; in the latter case, we can save building the restore region as it is rectangular.

Panels extensions. Although programming a container class is not extraordinarily difficult with a solid background in message semantics, most programmers shy away from the extra work, preferring to extend an existing container. It turns out that the most interesting extensions are based on panels, and either involve changing their controllers, or adding additional semantics that involve the children. For example, preventing a panel from accepting certain children, or defining some constraint between children falls in these categories. The remainder of the panel implementation is often left untouched. Unfortunately, as Oberon messages are often on a higher level of abstraction than those found in typical object-oriented languages, this is often difficult to do. Let's say the derived class would like to change the way the drag-and-drop operation is programmed, for example, to always make a copy of the gadget instead of moving the original. The only hook that the extender has is over-riding the handling of the input message, which means duplicating all procedures called in the process, as most of them are statically bound.

As defining more messages would increase the message discrimination time, it is preferable to provide a special client interface for extenders that is more efficient. Panels use a method block, a similar technique as used in the Draw system [WG92], that looks approximately as follows:

```
TYPE
  Panel = POINTER TO PanelDesc;
  Methods = POINTER TO MethodsDesc;

  PanelDesc* = RECORD (Gadgets.FrameDesc)
    ...
    do: Methods;
  END;

  MethodsDesc = RECORD
    RestoreBackGround: PROCEDURE (F: Panel; x, y: INTEGER; R: Display3.Mask);
```

```
  RestoreCaret: PROCEDURE (F: Panel; x, y: INTEGER; R: Display3.Mask);
  (* ... and so on for other methods. *)
END;
```

Defining a suitable interface for extending panels was only possible in the second iteration. The calling dependencies between methods are so complicated that the author has his doubts if a complicated extension can be made without the source code of the panel class. This is also the experience of [TGP89], where highly object-oriented code that makes extensive use of late binding is difficult to explain or to extend. Code inheritance, one of the primary advantages of object-oriented languages, is thus also the major problem.

If code inheritance is left to the expert, then extension of functionality leaves the casual programmer behind, except if a way can be found to redistribute tasks between container and child. The latter option is especially interesting when the extra functionality involves dependencies between children. Concretely, it involves redistributing tasks that are usually part of the container to the children. For example, a schematic editor gadget is a container that has to ensure that the chips remain wired together when the chips are moved by the user.

It is here that the broadcasting technique shows an interesting possibility. If a child, like say a "wire gadget", could monitor the message traffic between its parents and its brothers, it might obtain enough information to adapt itself accordingly. This involves the wire gadget adjusting its own position according to the modify message sent to the chip it is connected to. A simple, but inefficient implementation, involves forwarding the modify message after the standard handling of the event to all other children except for the child that was modified (which already received the message).

```
PROCEDURE AdjustChild((F: Display.Frame; VAR M: Display.ModifyMsg);
BEGIN
  (* normal message handling *)

  SelectiveForward(F, f, M)
END AdjustChild;
```

As we see, brothers can "listen along" for modifications, and thus broadcast recursively a modify message regarding themselves. This approach is interesting because the gadgets that change need not be aware of the gadgets that monitor them. It would thus be possible to wire existing gadgets together even without them being aware of the wires in the first place.

This approach would show some promise if it was not for the inefficient display update that results when a long chain of dependencies is present (which can be endless too if cyclic dependencies are created). As panels already manage a dirty rectangle for keeping track of the redraw area resulting from display messages with a *state* mode, a modification of the adjust procedure to detect recursive broadcasts of the *ModifyMsg*, and update the dirty rectangle accordingly, can solve the inefficiency problem. This modification is currently included in the panel implementation, and involves setting a flag in the container just before forwarding the modify message. Should another modify message arrive while the flag is set, we update the dirty rectangle and *change* the mode to *state* before forwarding the second modify message. Just before resetting the flag, we check if the dirty rectangle must be redrawn, and broadcast an update message to all panel instances.

Although the display update efficiency problem is solved, another source of potential inefficiency is the broadcast of a recursive modify message initiated by dependent gadgets. If complicated inter-depencies are present, many broadcasts have to be made to update the dirty rectangle. In this case, it is worthwhile asking if broadcasting does a good job. This is because a direct message send from child to parent—"upward" in the display space—also works in this specific case. The modify message, mode *state* is an example of a message that need not strictly be broadcasted, as the message sender assumes responsibility for updating of the destination. Not to invalidate our stated belief in broadcasting, we refrained from implementing this optimization.

6.5.4 Camera-views

Camera-view gadgets are type extensions of type *Gadgets.View*. The most noted gadget belonging to this class, the *camera-view*, allows the user to adjust the viewpoint by dragging the visual gadget being viewed. There are only two interesting aspects regarding programming camera-views (the remaining bulk of the implementation consisting of similar code as that of the container gadgets and also the implementation of several optimization for reducing screen flicker).

First, the display position of the viewed gadget is varied by adding a translation vector to the origin *M.x*, *M.y* passed in each frame message forwarded to the viewed gadget. As the display position of the gadget is the sum of the message origin and the frame's coordinates, this fools the gadget in believing that it has a new frame position. Each camera-view gadget thus keeps track of the current translation vector applied to its viewed gadget. In the current implementation of camera-views, the translation vector is relative to the top left corner of the camera-view, thus requiring the actual viewed gadget's coordinates to be "subtracted out" from the origin for the gadget to appear at the correct position.

Second, the camera-view has to store its absolute display coordinates in the fields provided in the type *Gadgets.View* before passing a frame message down to the viewed gadget. This is required for the imaging model, as discussed earlier in section 6.4.3.

6.6 Summary

This chapter introduced the visual gadgets, the message protocols they implement, and the imaging model based on display masks. The chapter concluded with examples of how messages cooperate, and discussed interesting aspects related to programming elementary, container, and camera-view gadgets.

Chapter 7

Documents as Objects

7.1 Documents

In the Gadgets system, application user interfaces and application documents are unified with the concept of *documents*. The consequent application of this idea—common in document-based user interfaces— blurs the distinction between applications and documents that are edited with applications.

A document consists of a run-time part with its visual representation, and a data part with its contents. The run-time part of a document is called a *document gadget*, a visual gadget that displays the documents contents. The data part of a document is an entity identified by a *document name*. *Loading* a document maps the document name into a document gadget, and *storing* a document writes the contents to the entity identified by the document name. The exact interpretation of load and store is a matter of the document class, and often involve more than just reading or writing data from and to a file.

In the following discussions we will refer to both document gadgets and document data as documents, making clear in the context which is meant. Furthermore, to simplify the discussion, we initially assume that document data is stored on secondary storage as a file.

Documents as containers. Documents are container gadgets with a single visual gadget as child. For example, in the case of panel documents, the document contains a panel, and in the case of text documents, the document contains a text gadget. The child *fills* the complete surface of the document gadget, making it optically impossible to distinguish between a document and its contents. This is

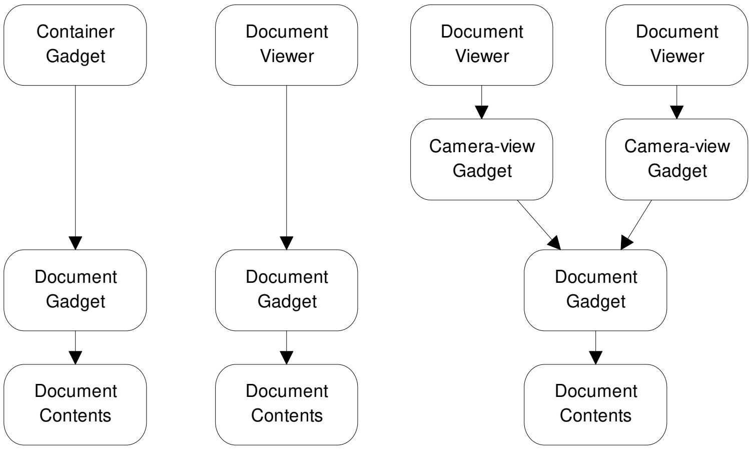

Figure 7.1: Embedding documents in the display space

an advantage when documents are embedded transparently inside each other.

The tight coupling of document gadget and its contents might make the reader believe that, for example, a panel should be a document. This is however false when we take into account that this unification means that nested panels would always be stored in different files. Instead, we distinguish between the orthogonal dimensions of nesting and persistency. Document gadgets collect their contents into persistent entities by *addition* of features, for example a document name, a way to print, load and store to a file. These features are not part of containers.

Let's investigate what happens when we insert a document gadget *inside* a panel, where storing the panel in a library requires that we store the embedded document gadget too. As the embedded document physically belongs to a different file, we only need to store the *document name* instead of copying the document contents into the library. On reloading the panel from the library, the document gadget loads its contents by mapping the retrieved document name into the document contents (found in the file with that name), and inserting it into the document gadget. The effect is that opening one document causes nested documents to "fill in" their contents too.

The other way to add a document to the display space—in addition to direct embedding in a container—is to insert it into a viewer. The viewer menu bar shows the document name and push buttons applicable to the document, and the viewer contents is the document gadget itself. But where does the menu bar come from? The viewer manager *requests* the document gadget to generate a menu bar, which the viewer assumes the responsibility for showing. The menu bar and document gadget are thus two separate but related entities, by way of the document gadget generating a menu for itself when so requested.

Viewers have two options for integrating the document gadget. One way is to embed the document gadget directly as the main frame of the viewer. The other way is to add a camera-view between the viewer and the document gadget. The latter feature is useful when the document gadget has a rigid size that cannot adapt to the outline of the viewer. Document gadgets that can adjust their size according to the viewer size are called *adaptive* documents, and belong to the class of gadgets that can be directly embedded in a viewer. From left to right in the Figure 7.1, we have a document embedded in a container gadget, directly in a viewer, and with a camera-view between the document gadget and the viewer.

The choice of how document gadgets are embedded in viewers is also dependent on the required copying behavior of a document. By convention, viewers make shallow copies of their menu bar and contents when the user presses the *Copy* button in the menu bar. Hence, when a viewer with a camera-view is copied, we have two views of the same visual gadget. Otherwise, we have separate copies of the viewer contents. In practice, we integrate text gadgets directly into viewers, and rigid objects like panels with camera-views.

The document type.

```
TYPE
  Document = POINTER TO DocumentDesc;
  DocumentDesc = RECORD (Gadgets.FrameDesc)
    name: ARRAY 128 OF CHAR;
    Load: PROCEDURE (D: Document);
    Store: PROCEDURE (D: Document);
  END;
```

The module *Documents* provides the base type of document gadgets and their default message handler. The type *Documents.Document* exports fields for the document *name* and methods to *Load* or *Store* the document. Calling the *Load* method loads the document contents from the entity identified by the document name into the *dsc* field; calling the *Store* method writes the contents to the entity identified by the document name.

The load and store methods must not be confused with the file message introduced earlier (cf. 5.4). The only situation where a document receives the file message is when it is embedded in a container gadget, which is about to be loaded or stored. The handling of file message involves loading or storing the size, position and name of the document gadget. Consequently, loading an embedded document happens in two phases. First, the latter data is read, followed by the document gadget invoking its own load method to fill in its contents.

The links and attributes of a document provide information to the document manager and the viewers documents are integrated in. For example, a viewer can request a document to return its menu bar with the link message. Documents are expected to generate three menu types: a menu for the system track, the user track, and for the desktop. These distinguish themselves mainly in the size and number of their push buttons. Documents have freedom in how menu bars are created. They can either generate menus on the fly, or use copies of prefabricated ones. The latter option is quite popular, as menus archived in public libraries can also be customized by end-users.

Other interesting attributes of a document include the document icon, specified by a string attribute that names a public object, and an attribute that specifies if the document is adaptive or not. The document icon is used by the desktop system to create a pictorial representation of the document, which the user can click on to open the document. The adaptive attribute is used internally by the viewer manager to make a decision if a camera-view gadget must be inserted between the document gadget and the viewer.

In all other regards, the documents behave as conventional containers. A display message displays the document, a print message prints the document, and so on. These features make it possible to integrate a document anywhere a visual gadget can be integrated.

Document classes. Before we can invoke the load method of a document, we need a document instance of the correct class. Hence on opening a document, we require a mapping of the document name to the document class, an operation that is not so harmless as it might seem. Ideally, we want a loose connection between the document name and the document class, so that users are able to name their documents anyway they see fit. There are several solutions to this problem, as the following paragraphs show.

With the exception mentioned in the following paragraph, a standard file format is used for all file-based documents (in EBNF):

```
Document = Header [ MetaData ] { DataBytes }.
Header    = Tag Generator X:2 Y:2 W:2 H:2.
Tag       = 0F7X 7X.
Generator = { CHAR } 0X.
```

The information we require to create an initial document instance, the document generator procedure, is part of a standard document header. The fields *X*, *Y*, *W*, *H* encode the document's display position, and act as a placement hint for opening the document. The document header is followed by optional meta-data and the document contents itself. The contents of the meta-data section is discussed later.

Loading from a file is implemented by reading the document generator from the header, creating a new instance of this document class by calling the generator using the module loader, copying the filename to the *name* field of the document instance, and invoking the *Load* method. The method re-opens the document file *name*, skips over the document header, and reads and internalizes the document data. The latter is often, but not necessarily, an anonymous library. Storing requires that the document writes its own document header, meta-data, and data contents.

A disadvantage is that the document header must be read at least twice; an improvement that involves passing a file *Rider* as parameter to the *Load* method was rejected on the observation that documents need not necessarily be stored in files (see the following paragraph titled network documents for details).

Compatibility. The standard document format works well for documents that actually have it, but does not allow the opening of old file formats with the universal document opening command. As an intermediate solution until all documents are converted to the new format, we maintain a small table of filename extensions and associated document classes in module *Documents*. On detecting an invalid document header in the document file, the database is consulted with the filename extension to generate an associated document gadget. If needed, storing the document again will provide the document with a valid document header. Of course, the limitation of this approach is that it only works if users stick to the Oberon conventions for file name extensions. In the case of a deviation from these guidelines, the document type casting possibility, as sketched in a following paragraph, can be applied to associate a document class with a data file.

Fortunately only a few file formats were in active use in the older Oberon system, so that the extension database has only half-a-dozen "compatibility" entries. Today, most documents are converted to the new document format.

Network documents. Documents need not necessarily be restricted to (local) files. With today's high level of workstation connectivity we can imagine accessing documents that are located on remote machines or are published on the Internet. Fortunately the problem of associating a document class with a document name is already addressed by the *Uniform Resource Locator* or *URL* syntax, the way in which services and documents are identified on the Internet. A URL name consists of a known number of prefixes, that specifies the service (for example "http", "ftp", or "gopher"), followed by the document location and name. Internet documents can thus be transparently included in our document framework by maintaining an additional database of service prefixes and associated document classes. The document manager is modified to detect the URL notation, and create the corresponding document class to fetch the document across the network.

In this model end-users have a unified model of local and remote documents. The same commands are used to open, store, print and manipulate any of these documents. Combined with the capability of the document manager to *exchange* documents in the viewer system *in place* with another, a hyper-text like browsing ability is a further intrinsic property of Oberon documents. We imagine the user clicking on a button or hyper link in a document, which is then replaced by another document fetched from the network.

A further interesting possibility is combining Internet services and Oberon documents. This involves publishing Oberon documents with Internet services like WWW and FTP. For example, a panel document on a WWW server is opened with the following command:

```
Desktops.OpenDoc "http://machine/Gadgets.Panel"
```

As the document class (panel document, in this case) is not known until after the document has been fetched across the network and the document header is read, the HTTP document class has temporarily to assume responsibility for transferring the data across the network, and then "transforming" or "replacing" itself with a panel document type. In our case, an embedded HTTP document must change to a panel document *after* the data was fetched from the network. This is problematic, as in concept, all references to the original HTTP document must be replaced by references to the panel document. The transformation of one type into another type is a feature not supported by statically checked programming languages like Oberon.

Here message handlers show welcome additional flexibility. The exchange of a document class is possible by exchanging the message handler and load and store methods (from those of the HTTP document to those of the panel document handler). Note that the exchange is associated with a drawback: we must not make type extensions of *Documents.Document*. This is because we cannot change the type of a gadget already referenced, as for example when it is embedded in a container. A solution might be to return another document with the load method; this is unfortunately too late, as the original document is already embedded inside a container when the load method is invoked. There is no way out of the dilemma for embedded document, hence our current approach to live with the restriction. The problem does however point to the fact that a means of type evolution at run-time can be useful, and should be investigated further.

Document casting. Once a document is provided with a document header, its default document class is "burned-in". In the case of a text document this means that the document can only be edited with a text gadget. Should another type of text editor be developed we might want to convert our text documents to this new format, and thus require a way to change the generator in the document header and also convert the document data format. This is foreseen by the system with a feature called document *casting*. To cast a document from one class to another, the end-user specifies the generator of the destination class *M.P* when opening a document:

```
Desktops.OpenDoc filename(M.P)
```

Instead of generating a document gadget belonging to the old class, we generate a document of the destination class and "point" it to the data file in the old format. The destination document will detect on loading that an unexpected document format has been encountered. The well-programmed ones might know this new format and convert it on the fly to the destination format. Should the format be unknown, the document reports an error and terminates the loading process. Storing a casted document converts it permanently to a new format.

Meta-data. In addition to their standard attributes and links, documents can be decorated with additional information just as any other gadget (cf. 5.5). As the number of Oberon documents are continually growing, these decorations can play an important role in organizing and locating documents. The idea is to attach information to documents, like author, keywords, objects that act as annotations, keep track of usage and statistics. As this information is attached to documents, in addition to the normal attributes and links, the information is managed by the default message handler for gadgets that store the information in the *attr* and *link* fields of the type *Gadgets.Frame*.

A drawback of document decorations is that they are intimately attached to the document gadget, so that accessing the information requires loading the document first. In the case of a search machine or document indexer, such an approach would lead to unnecessary inefficiency as documents that do not match the search criteria would immediately be discarded after opening. A more viable approach is to keep the decorations separate from the document data itself, by regarding them as *meta-descriptions* of the document. The section *MetaData* in the document file format is used for this purpose, and has the following file format.

```
MetaData = Tag len:4 Attributes Library.
Tag      = 0F7X 08X.
```

The *Attributes* section contains the attribute list of field *Gadgets.attr* and the *Library* section the objects of *Gadgets.link*. The format of the sections *Attributes* and *Library* are not shown above; service procedures in module *Documents* access these sections. The *len* field indicates the length of the following data so that, if necessary, external readers can jump over the data.

Unfortunately, document meta data is not extensively used in the Oberon system yet. The only interesting implementations at the time of writing include a decoration for document annotations, and a way to attach resources to a document (sketched in the following section).

7.2 Portable Documents

7.2.1 Motivation

The decomposition of large systems into modules and the separation of user interface and application modules create the problem insuring consistency of installed applications and documents. For example, when transporting documents to other machines we cannot always assume that all the necessary components to view that document are already installed. It might also be the case that some critical component of a large system is inconsistent with other components. This is a major problem as the *detection* of the inconsistency is often *delayed* until the system is used. An error condition only occurs once a missing or inconsistent component is required, which might only be when a seldom used operation is activated. The detection of a problem is made difficult by the fact that it is not possible to detect statically what the scope of an Oberon application is. This is an unfortunate side-effect of run-time module loading and the separation of user interface and application modules (remember that commands form a “hidden” link between user interface and application modules).

Although inconsistency is difficult to detect and thus to solve generally, an attempt can be made to solve the problem of *consistent distribution*. Consistent distribution means that when a user obtains a document, he or she can open and view it, even if it contains components not installed on that computer. Note that this is different from *consistency detection*, which attempts to determine if all the components an application requires are available and consistent with each other. First, we must assume that the creator of a document knows all the *resources* that this document requires as we cannot detect it by program. Resources might include auxiliary data files, libraries, documents, and modules.

A consistent distribution is made by attaching all the required resources to a document so that the document is internally consistent (with of course the previous assumption). Conceptually, a truly consistent document would require all parts of the Oberon system, even including the lowest modules in the module hierarchy. As we do not want a copy of the Oberon distribution in each document, we must assume that an Oberon installation at least consists of the parts required to open the document in the first place, which correspond at least to the most used parts of the Oberon system. For the purposes of our discussion, we assume that document developers are aware what components are part of a standard Oberon distribution; these shared components are available to everybody. Note that when source and destination machines are connected by network we can verify that the destination actually has all the components required, and if not, transmit them before sending the document itself, like in [Bah92]. In our implementation, we assume that no such network connection is available, and that documents are truly *self-contained*. This has the advantage that more conventional means of document distribution is applicable.

If we want self-contained documents to be usable on all Oberon platforms, effectively making them *portable documents*, we have the following problems to keep in mind:

- In addition to data resources, we have to include the platform independent implementations of components.
- Portable documents must be as compact as possible.
- Installing resources should be done only once, and not each time the document is used.
- We require a way to find out which resources are the newest and when new resources must be installed from the document.

- The inclusion of executable code in documents opens up the can of worms related to security and viruses. This is especially problematic in the Gadgets document model which allows access to remote documents in the same way as local documents. We must investigate ways to protect an Oberon installation from these unwelcome intrusions.

7.2.2 Resources

In our implementation document resources are restricted to Oberon modules and data files. Compressed versions of resources are attached to a document with a *document linker*. Each attached resource is given a *name* corresponding to the module of filename and a programmer-definable *version number* to distinguish between successive versions of the same resource. Resources and version numbers are packed into an object of type *Resources.Obj*, which is linked to a document as a decoration named “Resources”.

```
TYPE
  Resource = POINTER TO ResourceDesc;
  ResourceDesc = RECORD
    next: Resource;
    name: ARRAY 64 OF CHAR;
    version: LONGINT;
    F: Files.File;
    pos, len: LONGINT;
  END;

  Obj = POINTER TO ObjDesc;
  ObjDesc = RECORD (Gadgets.ObjDesc)
    res: Resource;
    signature: Crypt.RegisterPtr;
    key: Crypt.Key
  END;
```

As other document decorations, the resource object is stored in the meta-data section of the document file. On opening the document, the resource object is loaded and reconstructs its resource list. The resource fields *F*, *pos*, and *len* identify the resource inside the meta-data section of the document file. Note we are exploiting an interesting feature of the Oberon file system: should the file referenced by *F* be deleted, the file handle *F* still points to a copy of the original file which is made *anonymous*. The fields *signature* and *key* are part of the authentication mechanism for portable documents (cf. 7.3.1).

After the resource object has been loaded, but before the document is opened, we extract the required resources so that they become available before they are used by the document. In principle, the resource object could immediately install the resources just after the resource object is loaded. Instead of this approach we picked a mechanism where the document gadget sends a signal to all attached objects that they must *prepare* themselves for the imminent opening of the document, at which time the installation process takes place. This places the installation process under the control of the document and not the resource object.

Installation of a resource is dependent on its presence on the destination machine and its version number. A table of already installed resources and their version numbers is maintained in a file. Installation occurs according to the following pseudo code, executed each time the portable document is opened:

```
FOREACH document resource
  IF (not installed) OR (document resource newer than installed resource)
    Install resource
```

In the case of data resources, installing involves uncompressing the resource from the document file and writing it to a file corresponding to the resource name. We use the same compression mechanism as the Oberon archiver, namely LZSS [Zel93, Nel91c]. Module resources are installed with the mechanism described in section 7.2.3. Note that the resources remain in the document file until they are explicitly removed by the user. As portable documents always remain consistent, no problems occur should the user decide to delete the installed resources, for example to free up disk space. For exactly this purpose, resources are installed in a user-specified directory which can be purged any time. This also reduces the risk of resources over-writing system components.

Caveats. Independently developed resources sharing the same name is a major problem in open modular systems. For example, in Oberon, modules (with the same name) can only be loaded once, restricting only one of many modules with the same name being active in the system.

There are two solutions to this problem: a name conflict can be removed by renaming resources at the destination or source. If renaming resources occurs only when a conflict occurs (at the destination), we might have to modify parts of an application. In Gadgets, the connection from user interface to application module must also be guaranteed under command renaming. As a solution based on renaming just defers the naming process from the developer to the module loader, we can achieve the same effect by naming modules uniquely in the first place (at the source). This can be as easy as prefixing a developer name to module names and other resources (like in Java [Mic95]). In the long run, the enlargement of the module name space in Oberon has thus to be foreseen, and is essentially a question of a minor system modification to reduce the restriction on a module name length.

A more serious problem with the portable documents, as sketched above, is that new document classes cannot be distributed inside resources packed in documents having that class. This is because the document generator is called to create a document instance, which in turn must unpack the resources. If the document generator is not available yet, we cannot create the document to unpack its own module. This hen-and-egg problem is solved by modifying the document loader. The modification involves the document loader reading the meta-data and sending the prepare signal to the resource object itself, instead of the document as is usually the case. The result is that control over meta-data is removed from the document's responsibilities.

In the author's opinion, the preferred solution is based on a matter of taste. The current implementation keeps control with the documents themselves, which is more in the spirit of object-orientation, but has this restriction. A compromise is possible by having the document loader usurp control from documents *only* in the case where the document generator is unavailable, as detected when reported by the module loader on executing the document generator.

7.2.3 Module Transport

In principle, module resources of portable documents can be transported as source code installed by invocation of the compiler. Its disadvantages include the lack of investment protection and required compilation time. As advantages, optimal run-time efficiency is obtained in comparison with interpretive approaches based on distributing byte-codes, as in Java [Mic95], and the document is completely portable. As re-engineering systems from low level abstractions like machine code and byte-codes is just a matter of time and money, protecting the development investment was not an overriding concern in our implementation of portable documents for Oberon; a simple but not trivial module representation instead of pure source is deemed to be sufficient.

The number of ways modules can be transported, counting out source code and machine dependent object code, depends on how many ways we can cut up the Oberon compiler and still obtain a platform-independent module representation. The natural choices include a token stream or a parse tree representation (for a two-pass compiler). These involve splitting the compiler between the token scanner and the parser, and the intermediate parse tree representation and the code

Storage Technique	*Relative Size*
Source Code (bytes)	100%
PowerPC Object File	88%
Intel Object File	76%
MC680x0 Object File	70%
Compressed Source	33%
OMI Object File	26%
Compressed Tokens	25%

Table 7.1: A size comparison of module representations

generator [Cre90, Wir96]. The further the selected presentation is in the compilation process, the less work has to be done to install the module resource, and the faster it takes place. Currently, both token stream and parse tree representations are used for transporting modules in portable documents. The reason is more pragmatic than anything else. A parse tree representation of modules, based on the Oberon Module Interchange (OMI) [Fra94b, Fra94a], requires an adapted Oberon system, which was not available for all Oberon platforms at the time of writing. A token stream representation of modules is simpler to implement, as it only involves inserting an interface between the scanner and parser for reading and writing tokens.

The token stream representation of a module is created by scanning the tokens of a module, and numbering identifiers as they are encountered. The resulting token stream, consisting of the token numbers, identifier numbers and the identifier table, is further compressed using the common LZSS compression algorithm. The decompression of the token stream representation can be imagined as a pipeline that reads the resource data, decompresses it, and reconstructs the original scanner token stream for feeding into the compiler's parser.

There are even further possibilities for compacting the token stream representation. In [Zel], unexported identifiers—which cannot influence the module symbol file—are omitted and substituted with numbers. This transformation increases the difficulty in re-engineering the module contents. This improvement has not been incorporated in our implementation. A comparison of file sizes of modules using a few selected representations is given in Table 7.1. The Oberon Internet tools (including source comments) consisting of 19 modules are the basis of these measurements. For comparison, the size of native object files on a few platforms are also listed.

What is interesting is that the compressed token representation and OMI object files are approximately of the same size. This is because OMI object files use a similar predictive way of encoding symbols as LZSS. The distinguishing factor is the installation speed; OMI-based modules can be unpacked unmodified and loaded immediately, whereas the token stream representation must first be unpacked and then compiled. Once the token stream is compiled, the cost of loading a natively compiled module is lower than loading an OMI module; the latter does the decompressing and compilation of the parse tree each time the module is loaded. A speed comparison of OMI and the token stream representation is highly dependent on the platform, hosted operating system, disk transfer rate, and Oberon compiler back-end for that platform. The current implementation of OMI for Intel processors, with a two pass compiler back-end, cannot be directly compared with a PowerPC and MC680x0 implementation, that uses a one pass compiler back-end. Also the relatively slow disk sub-system of MacOS in comparison to Intel-based PCs skews numbers across platform comparison. Objective benchmarks show that the two pass Intel back-end is currently the bottleneck for OMI on Intel: in comparison to PowerPC implementations where OMI is not significantly slower than native object files, OMI object files on Intel have 4 to 8 times longer loading time than native Intel object files, depending on machine characteristics. Subjectively, the loading times involved are so small that one or the other technique makes little difference. For example, the complete Internet tool suite of 19 modules loads with OMI in approximately 4 seconds on an Intel 120 MHZ processor.

7.3 System Protection and Security

Single user operating systems are especially sensitive to badly written, virus and Trojan horse infected software, which can crash the system, access sensitive data, destroy software or even the whole installation, and bar further access. In Oberon, the situation is not different. Any software obtained can exploit the security and protection deficiencies of the system. In document-based systems like Oberon, the perceived problem seems to be larger than usual, because even supposedly and traditional harmless entities like documents can harbour unwanted executable content, and simply the act of opening a document has unknown side-effects. The problem is aggravated by high connectivity of the Oberon system, which allows remote documents to be accessed as if they were local. In fact, the problem is only perceived to be larger: it is an inherent problem of all software, that is unrelated to documents or the network.

These issues and the protection of software systems are traditionally the domain of the military, who want to restrict the flow of information, and multi-user systems [AGS83], which are interesting targets as so many people are both dependent on the system and so much "interesting" information is collected in one place. Information security, as practiced by the military, has three separate but interrelated objectives [San93]:

1. *confidentiality*, related to the disclosure of information,
2. *integrity*, related to the modification of information and software,
3. *availability*, related to denial of access to information and software.

Unfortunately, the military have specific interests related to security classifications of people and not programs. This is an important point. The security mechanisms incorporated in multi-user systems like UNIX build independent user spaces restricting the influence, access, information flow of one user over and to another, are of little use in a single-user environment. A single user of a UNIX operating system is just as unhappy when the program that he or she just downloaded over the net wrote over private files. The security mechanism designed for this environment is to protect the system, and not its clients.

This example illustrates that a security level determined by the user identity is too coarse-grained; security attributes are to be associated with single applications. In the case of Oberon, the "identity" of modules and documents need to be determined, which in turn determines what they can or cannot do. For effective system protection in a single-user environment, two problems must be solved. The first problem is that of *authentication*. Applications are not resistant to tampering, as the large number of viruses can attest. A trusted application can be turned into a havoc creating one by the adjustment of a few minor bytes. A digital signature testifying its authenticity is required. A second and much more difficult problem is to decide how and where to restrict access for authenticated applications.

Various approaches are possible. An incomplete list of (non mutual exclusive) approaches, sorted according to decreasing user paranoia, might be the following:

- *Prevention*, which restricts all untrusted applications as much as possible, but with enough freedom so that at least something useful can be done. This approach assumes that all programs are bad. It is for example the approach followed in Java. Java restricts access of untrusted applications to resources in a single file directory. This approach is an anti-thesis to an open system—why bother when no clients are trusted to use these features. Prevention is expensive to implement and involves a game of wits between attackers and defenders.

- *Confirmation*, which verifies each potentially "dangerous" operation and asks the user if it can be completed or not [Dät96, Han90]. This approach assumes that there might be bad programs, and is deficient because the user quickly looses his or her patience answering questions that might be above the user's level of understanding.

- *Authenticated trust*, which involves only using application software that has been digitally signed by its manufacturer. This approach assumes that nobody wants to be identified as the culprit. It is dependent on trusting somebody who your lawyer can call if your machine is dead after installing the software (as long as you can prove it). This is often enough motivation except when in a state of war.

- *Trust* in your fellow man or woman. The current state of affairs in the software world.

The island approach. The principal difficulty in introducing strong security and protection mechanisms in Oberon is the high cost. The system was not designed with these features in mind and has numerous unsafe module interfaces. As a redesign of the system is out of the scope of this thesis, we investigated cheaper alternatives, like authenticated trust which only involves verification of digital signatures of modules.

Two alternatives come to mind. A reasonably expensive approach is to verify digital signatures each time a module is loaded and thus protecting the module against tampering. A cheaper approach is to verify digital signatures at the time of installation. We assume (perhaps mistakenly) that installed (and thus trusted) applications do not modify modules of other applications. This measure alone cannot protect an Oberon system; once a module has passed the immigration gate of the "island" it has full access (even when later modified).

The island approach was selected as a means to protect an Oberon system against unwelcome portable documents. It is an open question if the island approach is sufficient for protecting the Oberon system. It is clearly a minimal approach that, according to the author is sufficient for low security applications. It can also form the basis of a more refined and selective model should the Oberon module interfaces be adapted. In the latter regard, Oberon has a large potential because it has a strong foundation in type safety (cf. 7.3.2).

7.3.1 Authentic Portable Documents

Public key cryptography. Digital signatures, authentication and public key cryptography are dependent on mathematical *one-way functions*. One-way functions are easy to compute in one direction but difficult to compute in the other direction. A *trap-door one-way function* is a special type of one-way function that is invertible with the help of some secret information. They are used to transmit coded messages. The message sender (often referred to as Alice) uses the one-way function to create an encrypted version of a message, which the receiver (often referred to as Bob) can decrypt by inverting the function using the secret information. If only the receiver knows the secret information, then he or she is the only person that can decipher it. The trap-door one way function can be shared by many communicating parties by parameterizing it with public and secret information. The *public key* of Bob is the information he publishes for somebody to send a message to him. He uses his *secret key* to read the messages encrypted with his public key. The security lies in the fact that the private key cannot easily be deduced from the public key—the "interesting" functions from the cryptographers viewpoint are those that can be made arbitrarily difficult to invert. The advantage of *a-symmetrical public key cryptography* is communicating parties need not agree on a shared key to communicate (as in conventional symmetric cryptography).

A digital signature is based on the above technique, except that it is applied in reverse. For Alice to sign a message, she encrypts it with her private key to form the signature. To verify that the message comes from Alice, Bob uses her public key to decrypt the message; if it is readable, he is sure that it must come from Alice. The protocol has the following characteristics [Sch94a]:

1. The signature is unforgeable; only Alice knows her private key.
2. The signature is authentic; when Bob verifies the message with Alice's public key, he knows that she signed it.
3. The signature is not reusable; the signature is a function of the message and cannot be transferred to any other document.
4. The signed document is unalterable; if there is any alteration to the message, it can no longer be verified with Alice's public key.

5. The signature cannot be repudiated; Bob does not need Alice's help to verify her signature. Anybody that knows Alice's public key can verify that the message came from her.

RSA digital signatures. The digital signature mechanism used for portable documents is called RSA [RSA78], one of the easiest to implement and secure protocols. The signing of a document with the RSA algorithm requires a large number of computations, which makes it unfeasible for signing the whole text with the one-way function. Instead, Alice only signs a one-way hash of the document. A one-way hash function creates a characteristic number from the document contents that is much shorter, typically 128 bits, which is called a *message digest*. As the chance of two documents hashing to the same value is only one in 2^{128}, we can safely equate a signature of the hash with a signature of the document. The one-way hash function used for portable documents is called MD5 [Riv92].

The RSA algorithm will now be sketched; its theoretical underpinnings and a general description of its use are found in [RSA78, Sch94a]. Alice generates a private and public key pair by picking two large prime numbers p and q (typically 256 or more bits long each). Compute the product:

$$n = p \times q$$

Pick an encryption key, e, so that

$$GCD(e, (p-1) \times (q-1)) = 1$$

Use the extended Euclid algorithm to compute the decryption key, d, so that

$$e \times d \equiv 1 \pmod{(p-1) \times (q-1)}$$

The numbers e and n are Alice's public key. The number d is the private key. The two primes p and q are no longer needed and should be destroyed or never revealed.

To sign a message digest m, where m is a numeric representation of the message digest and $m < n$, compute

$$s = m^d \pmod{n}$$

To verify the signature compute

$$m = s^e \pmod n$$

and compare it with the message digest of the received message. A match indicates that the document was signed by the corresponding private key.

It is conjectured that the security of RSA depends on the factoring of large numbers. The larger n, the more difficult the problem becomes. With current factoring mechanisms it is agreed that a modulus of 1024 bits (308 decimal digits) is sufficient for long-term secrets for the next ten years. Assuming one computer can form a million factoring steps per second and a million computers can work on the task, then it will take 10^{10} years to factor a 1024 bit modulus.

Key distribution and key rings. Although years of investigation have increased the confidence in the RSA algorithm itself, the signature protocol can be comprised in a very simple way. The protocol assumes that the public key of Alice really belongs to her. An imposter, Mallet, can generate an own public and private key pair, and publish the public key as if it belonged to Alice. The signature protocol is thus highly reliant on the public keys being authentic. There are several approaches to ensure this:

- Alice and Bob can meet face to face, verify each other's identity, and exchange public keys.
- Alice can publish her public key in a database or newspaper. As long as her public key is easily accessible, the chances of Mallet introducing a fake key is reduced.
- If no direct verification of Alice's public key is possible, a central authority can sign her public key, which can be verified using the authorities' public key. A digital signature on a public key is called a key *certificate*.

Public key certificates have the advantage that communicating parties only need to know the public key of the authority, which is highly published, perhaps even burned into the operating system ROM. In the case of portable documents, each developer would have his or her public key signed at the authority, and so obtain a valid identity. Portable documents signed by a key without a certificate from the authority might have been compromised and must be rejected.

In our implementation of digital signatures for portable documents, the public and private keys are visual key gadgets (Figure 7.2). As gadgets, keys can be

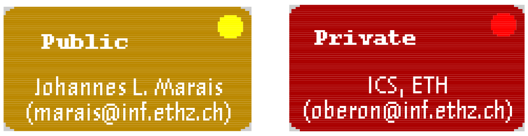

Figure 7.2: Public and private key gadgets

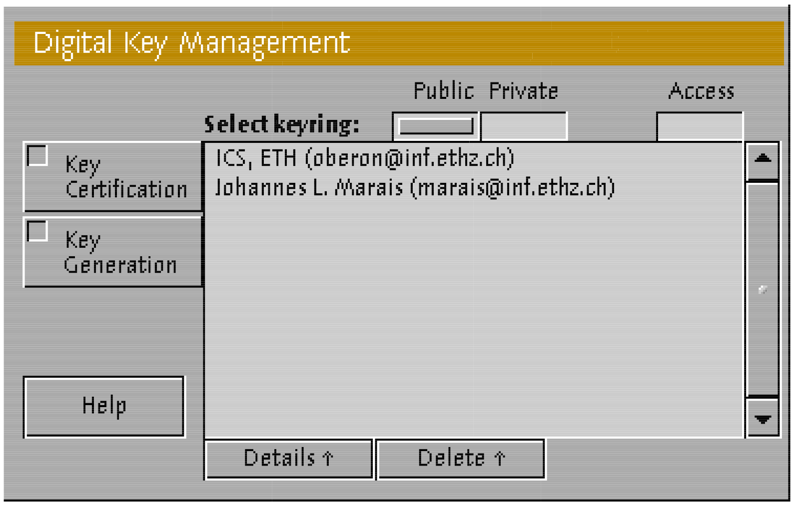

Figure 7.3: A panel for managing keys

transported inside documents and electronically mailed from one user to another. Keys are collected on *key rings*. We define key rings containing the public keys of acquaintances (the public ring), a key ring for the user's secret keys (the private ring), and a key ring identifying developers whose portable documents have access to the machine (the access ring). A panel allows the user to drag and drop keys among rings, generate new key pairs, certify keys, and verify certificates (Figure 7.3). As the security of the protocol is compromised when a private key is stolen, we encrypt private keys with a personal identification phrase, which must be entered before the private key is unlocked for use. The encryption algorithm we use is for private keys is IDEA [LM91], with the cipher key the MD5 hash of the identification phrase.

To reduce the number of keys on the access ring, we also allow access to documents that are signed by a person whose key is certified by one of the keys on the access ring. The idea is that when allowing access to an institution like a software house, we automatically allow access to all developers belonging to that institution. As authentication only provides traceability, and not defense or protection, we keep track in a session log of resources installed through portable documents and of the keys that were involved.

The current implementation is incomplete in the sense that the log is not protected from tampering and no standard revoking mechanism is foreseen for revoking compromised private keys. The latter would not be too difficult to add to the implementation. The first involves the resource manager signing the log after a modification with a private key embedded in the software. This however has the danger of Trojan horses fishing for the key if they can circumvent the system's type safety.

A cryptographic toolbox. A cryptographic toolbox comprising about 1200 Oberon statements was implemented for the authentication protocol. Its scope includes mathematical operations on arbitrary precision integers, the IDEA symmetric cryptographic algorithm, the MD5 message digest algorithm, the fast generation of large prime numbers, public and private key calculations based on RSA, key certificates, signing of message digests, key ring management, and password prompting. This toolbox covers many algorithms used in public key cryptography.

An extract of the *Crypt* module is presented below. The definitions are restricted to multi-precision *registers*, keys, certificates, key rings and a few selected operations.

```
DEFINITION Crypt;

  TYPE
    RegisterPtr = POINTER TO Register; (* Multi-precision Integers *)
    Register = ARRAY OF LONGINT;

    Certificate = POINTER TO CertificateDesc;
    CertificateDesc = RECORD
      authority: ARRAY 128 OF CHAR; (* Certifying authority. *)
      signature: RegisterPtr; (* Signature of key. *)
      next: Certificate;
    END;

    Key = POINTER TO KeyDesc;
    KeyDesc = RECORD
      name: ARRAY 128 OF CHAR; (* Owner. *)
      private: BOOLEAN; (* Is this a private key? Then exponent is encrypted. *)
      size: INTEGER; (* Size in bits of the modulus *)
      exponent, modulus: RegisterPtr;
      check: LONGINT; (* Least significant LONGINT of un-encrypted exponent. *)
      time, date: LONGINT;
```

```
    certificates: Certificate;
  END;

  Enumerator = PROCEDURE(K: Key);

  Keyring = POINTER TO KeyringDesc;
  KeyringDesc = RECORD
    name: ARRAY 64 OF CHAR;
  END;

(* Key generation, encryption and decryption *)
PROCEDURE MakeKeys(VAR primep, primeq: Register; name, password: ARRAY OF CHAR;
      public, private: Key);
PROCEDURE RSAEncrypt(publickey: Key; VAR msg, result: Register);
PROCEDURE RSADecrypt(privatekey: Key; password: ARRAY OF CHAR;
      VAR msg, result: Register; VAR res: INTEGER);

(* Key ring management *)
PROCEDURE OpenKeyring(name: ARRAY OF CHAR): Keyring;
PROCEDURE EnumKeys(R: Keyring; enum: Enumerator);
PROCEDURE InsertKey(R: Keyring; K: Key);
PROCEDURE DeleteKey(R: Keyring; K: Key);
PROCEDURE FindKey(R: Keyring; name: ARRAY OF CHAR; VAR K: Key);
PROCEDURE CloseKeyring(R: Keyring);

END Crypt.
```

A complete description of the algorithms used is beyond the scope of this thesis; the interested reader is referred to [Sch94a] for an overview. Instead, I will present a few general comments about the difficulty of implementing this module in Oberon.

Open arrays were invaluable for operations on multi-precision integers. The user selects the security level for digital signatures by varying the size of the modulus, which determines the size of the multi-precision integers. Open arrays are the only Oberon-2 feature used in the Gadgets system (and only in this module).

We regard the LONGINTs of the multi-precision registers as unsigned integers, and so require code procedures for unsigned compares and long multiplication with a 64 bit result. Although unsigned compare and long multiplication can be coded in Oberon, their portable implementation is too slow for cryptographic applications. The low-level features of module *SYSTEM*, especially the bit manipulation operations based on SETs, are heavily used in the IDEA and MD5 implementations. Until bit manipulations are directly allowed on integers, this code remains unportable.

Some time was spent tuning the algorithms. This often involves eliminating unnecessary procedure calls, which is tedious and error prone for algorithms like MD5 that make lots of bit manipulation; the wish for inline expansion of proce-

Operation	*512 bit*	*1024 bit*
Signature	109	605
Verification	7	45

Table 7.2: Digital signature speed (ms)

dures was ever present. A first implementation of RSA was disappointingly slow. In contrast to RSA encryption, which typically involves modular exponentiation with a small exponent, RSA decryption involves modular exponentiation with a very large exponent. The effect is that signing a document is much slower than signature verification. Investigation showed that the bottleneck was a slow divide routine. Two further algorithms boosted the performance. The *Chinese remainder theorem* reduces the calculations for the modular exponentiation by exploiting the primes p and q, which now become part of the encrypted private key. A further speedup involves a special internal representation for numbers that avoids division by n [Mon85].

Table 7.2 compares signing and verification of a MD5 message digest on a 120 MHZ Intel Pentium-based PC. The signature operation includes the verification of the user identification phrase that protects the private key. In comparison with highly tuned Intel assembly code the Oberon RSA algorithm is about 7 times slower [Dai]. We did not improve the implementation any further as signature verification is done adequately fast for our purposes.

7.3.2 Perspectives

With the growing popularity of open operating systems and Internet connectivity, the distribution and verification of software components and network aware applets becomes an increasingly larger problem. As the island approach as applied to portable documents only provides enough protection for low security applications, we still have to investigate further mechanisms for protecting single user operating systems.

In this regard, Oberon's type safety is a second level of defence. A third level of defence involves the modification of the system itself to take protection mechanisms into account. Although these approaches are not addressed in this thesis, a short overview of the situation is interesting.

Type Safety. In principle, Oberon's type safety can protect sensitive parts of system against tampering, as it prevents modules from accessing things they are not allowed to access. The verification of a module's type safety is with the compiler, which is trusted to detect unsafe operations on data. By disallowing the import of module SYSTEM, which circumvents the type system, we can ensure that a module can only access the features that other modules provide to it (this restriction is also enforced for portable documents).

In practice, even though Oberon is type-safe for general programming practises, it still has a few typing holes that provide a convenient backdoor for hackers:

- Oberon modules can be freed regardless of PROCEDURE variables still pointing to procedures in the module. Calling a "dangling" PROCEDURE variable often results in a run-time exception; it can however be exploited to access restricted areas of the system. A poor man's solution prevents unloading of modules; a more refined strategy is to tag PROCEDURE variables to detect to which module they point.

- The Oberon WITH statement applied to POINTERs can be used to circumvent the type system, according to the following scheme:

```
TYPE
  T = POINTER TO TDesc;
  TDesc = RECORD END;
  T1 = POINTER TO T1Desc;
  T1Desc = RECORD (TDesc)
    a: INTEGER;
  END;

VAR t: T;

PROCEDURE P;
BEGIN NEW(t);
END P;

PROCEDURE Do*;
BEGIN
  WITH t: T1 DO
    P;
    t.a := 0; (* Boom ! *)
  END
END Do;
```

- As all types are compatible with procedure parameters of SYSTEM.BYTE, we can misuse variable parameters of this type to generate fake pointers.

> This is especially problematic with *Files.ReadBytes*, which allows the creation of fake POINTERs from any data written in a file. In this case we have an unsafe interface to module *Files* that can be used without importing module *SYSTEM*.

Protected interfaces. A larger drawback than the type holes in the Oberon language (which can be fixed by adding more restrictions and checks) is that the Oberon module interfaces have no concept of access rights or security; it was a design decision to keep the system as simple as possible. A type safe module interface does not necessarily guarantee that the module's operations are safe—an explicit or implicit security token must be passed to the callee procedure to verify if the caller has the right to access a resource or perform an operation.

An *explicit security token* approach can be supported by type safety. The token is a pointer to an opaque type implemented by the system kernel, which is passed as an extra parameter to the callee. The combination of type safety and security tokens can provide a very efficient protection mechanism, as illustrated by its use in the SPIN operating system [BSP$^+$95] based on type safe Modula-3 [Nel91a].

An *implicit security token* approach determines the momentary access rights from the call trace from procedure to procedure; should an untrusted procedure be found on the execution stack, the operation is disallowed. A similar approach is implemented in [Dät96] in an extended portable document implementation for Oberon System 3.

Sometimes the modularization of the Oberon system requires the export of "private" objects from low level modules for the exclusive use of a select few high level modules. This is often the solution to a bootstrapping problem or a convenient decomposition of large sub-systems. Prime examples include modules *Kernel* and *Modules*, which are a combination of safe procedures and low level access routines. Restricting untrusted modules to only importing a sub-set of "safe" modules is thus of little value until the Oberon system interfaces are clearly divided into safe and unsafe sub-sets.

7.4 Summary

The current interest in Internet-based applications, as popularized by Java, prompts the question if portable documents can be embedded in HTML pages too—the answer is yes. In an experiment we extended the Oberon HTML browser with our own markup tags to include portable documents. The implementation turned out to be quite simple. The new tag just needs to contain the name of the portable document to be included, and the current document version number. The latter number is stored in a local database that keeps track when to re-fetch an already fetched portable document over the net. Portable documents embedded in this way in an HTML page are called *gadlets*. The major disadvantage of this approach is that gadlets can only be used inside the Oberon system and not with other HTML browsers, which makes them of little interest to the larger world.

Chapter 8

Summary and Conclusions

8.1 Summary

Modern software engineering involves the complementary processes of *decomposing* complex systems into constituent parts, and *composing* a software product from those parts. The reasoning behind this approach is threefold: making the software system easier to comprehend, independent construction of system components, and the possibility of reusing constructed components to amortize their development cost. Until recently, *building* components using object-oriented techniques received the most attention, whereas the composition process was viewed as a side-effect that follows from the application of those techniques. Unfortunately, it has been observed [JF88] that component composition and software reuse do not happen by accident, even with object-oriented languages.

The problem is that reuse is dependent on components being *designed* for composition. System-wide programming conventions and communications protocols are required before any composition can take place. These conventions and protocols, manifested in the component interfaces, have a tremendous impact on the stability and usability of a software system. Interface adaptions, corrections and improvements necessarily lead to the invalidation of clients of that interface, which in turn requires adjusting client code or writing "adapters". This is referred to as high "plumbing cost". In systems with well-defined scope it is often possible to define in advance interfaces with a high degree of stability, whereas, in systems that can be independently extended, it is difficult to impossible to define generally applicable interfaces.

With this as background, the author's premise is that if we require components to be as easily composable as Lego building blocks, we have to investigate more flexible and extensible ways of inter-component communication. Thus we set about building a system where:

- components are first class citizens,
- components can be composed interactively,
- and the set of components is independently extensible at run-time.

These features are a good exercise for pluggable component. We expect first class components to be reusable across application boundaries. Interactive composition requires a very flexible "glue" for connecting components at run-time. Independent extension means that we cannot force a mutually consistent view of components and their interfaces—we expect independent parties to *define* or *extend* communication protocols and components for specialized purposes, without invalidating existing clients. This thesis describes the design and implementation of the Gadgets system, a system designed to experiment with the goals stated above. The following paragraphs give a short synopsis of the main features and design of this system.

Components in the Gadgets system are called *gadgets*. A classification hierarchy divides gadgets, amongst others, into classes of visual components for the construction of graphical user interfaces, and non-visual components for connecting user interfaces and applications. Visual components are used and edited in place both during system development and in the delivered product. This makes it possible for the end-user to customize and reorganize applications at any time, as if editing a document. Drag-and-drop allows the movement of gadgets between container gadgets, and the construction of complicated components from simpler ones. As an important part of the thesis, a large set of typical user interface dialog elements were realized as gadgets.

Gadgets composition is an interactive process involving nesting visual gadgets in each other, and connecting components with each other using communication links. A generic attribute editor allows the inspection and configuration of gadgets on the fly. Actions are associated with gadgets by specifying the Oberon commands executed when activated. A scripting facility allows the reuse of prefabricated commands, and the Oberon language is used for constructing the latter.

In-place editing blurs the distinction between applications and documents, allowing us to unify these two concepts in *documents*. A document might be a prefabricated panel for controlling an application, or a graphic document consisting of numerous embedded gadgets. If required, documents can be embedded in other documents. Documents are identified using URL notation and remote documents can be accessed as if they were local. Documents classes for Internet-based documents like Web pages and anonymous FTP are tightly integrated into the document model. Gadgets can be transparently integrated into Web pages.

The Gadgets system addresses consistent document distribution by providing a means of attaching resources like component code and data files to documents. This ensures that a recipient of a document can view it without installing additional software in a separate step. The embedding of active content in documents is a high security risk as untrusted code can easily make improper use of the system where the document is viewed. To soften the problem and to introduce *accountability*, document authentication with digital signatures based on public key cryptography is used.

The design of the Gadget system is based on a set of hierarchies. The module hierarchy is the principle structure of the system; its purpose is code reuse. The object type hierarchy defines the internal structure of components, and corresponds closely to the traditional class hierarchy. The message type hierarchy defines the message protocols components understand: in comparison to other systems, this is the most unique hierarchy in the system. The display hierarchy forms the run-time organization of components, for example what appears on the display. The persistence hierarchy determines how components persist from one Oberon session to another.

Of these hierarchies, the message type hierarchy plays a very important role in glueing components together. Component connections are realized through *open message interfaces* that provide a universal message interface for components. An open message interface can accept any message, even if it is defined after the initial design of a component. The possibility of manipulating messages in a generic way allows the construction of sophisticated message handling mechanisms. An additional advantage is that components can be upgraded with additional functionality without invalidating their interfaces and thus clients. An important part of the thesis work involves the definition of compact message protocols for the use and construction of document-based user interfaces, and the realization that a set of well-thought out protocols can create very complicated behaviors.

8.2 What has been achieved?

The ultimate test of a component-based system is how it stands up in practice regarding the number of components, the application classes that can be built with it, the level of reuse achieved, and the speed in which new applications can be constructed.

In this regard, the Gadgets system is in a fortunate position in respect to other research prototypes. Approximately forty gadgets are distributed with the standard distribution, and several applications add a dozen more. Many gadgets are based on standard dialogue components found in typical user interfaces, although more complicated gadgets for editing and viewing diagrams, bitmaps, and three dimensional worlds have been constructed. The gadget palette is a representative mix of generally usable components.

Regarding the application classes that can be constructed with the Gadgets system, many interesting applications ranging from graphics, databases, to simulations have been experimented with. There are however application classes that are not suitable for the Gadgets system, like high speed animations. The main limitation is the performance of the message broadcasting system when using many fine-grained gadgets.

The level of component reuse is quite high amongst Oberon applications. Most applications use the set of prefabricated user interface elements provided by the system. Components are seldom extended; most new gadgets are completely new instead of being based on an existing gadget. A notable exception are the panels, which are often extended with new features.

The speed in which new applications are build is highly dependent on the programmer itself, as is typical for complicated systems. Our experience from student projects however shows that most students grasp the main concepts within the first few days of a project, and often produce a first new gadget soon after. This has resulted in a large number of successful student projects with highly visual user interfaces. The quick results is a major motivation for the students. Also, course work about Oberon System 3 and Gadgets has shown that it is possible to teach a person knowledgeable in the Pascal family of programming language in approximately a week's time the basic principles of the Oberon language, the Oberon message mechanism, the persistency model, and all the levels of Gadgets programming, except for a cursory overview of the last level (programming an own container gadget). This corresponds approximately to a single semester course.

8.3 What should still be done?

In retrospect, I am satisfied with the current design and scope of Oberon System 3 and Gadgets. There are however some further aspects of the system that needs investigation.

Although open message interfaces provided the muscle for building components, they are unsatisfactory from a software engineering viewpoint as the message protocols a component understands is not explicitly known. The current suggestion is to send a request to a component to confirm that it speaks a certain protocol. Language extensions could make supporting this operation more convenient, and could potentially check if a component completely implements a desired protocol. I believe that the lack of such facilities tends to introduce implicit assumptions about components that do not necessarily hold.

Currently the set of model gadgets is rather small. It would be challenging to enlarge them to more general purpose components. We already have more specific model gadgets, like for examples ones that continuously update statistics about the system state and can be connected to oscilloscope and histogram gadgets. With the recent development of Columbus [Sal96], a tool that can compose non-visual gadgets "off-screen", there are potentially many more possibilities for development in this area.

Another area that I perceive as a problem is the abrupt transition from user interface composition to programming commands. The current scripting facility based on macro substitution is simple and easy to understand, but assumes that a large set of useful commands can be created. As scripts are so highly application dependent, it is often difficult to think of general command sets. Thoughts about a general command set eventually lead to the need for control structures, variables, procedures and so on. It is however unlikely that on this high level of abstraction the full power of the Oberon language is required or even necessary, and thus we have to think seriously about adding an "intermediary" scripting language between the current macro scripts and the Oberon programming language. This intermediate level could additionally ease the transition from end-user to interface builder.

8.4 Conclusion

The Gadgets system illustrates that it is possible to build fine-grained document-based user interfaces when we rethink how components are integrated with each other. Although interfaces and message protocols have traditionally played an important role in software development, the applications of these techniques to *independently* developed software components create a new challenge of trying to reconcile several different views of how components should look and behave. This necessitates a re-evaluation of component interconnects, for example, how to make interface more open, flexible and extensible. These issues are becoming increasingly important with the increasing industrialisation of software construction.

Appendix

To give an impression of the size of Oberon System 3 and Gadgets, the following tables list the size of the most often used modules of the Spirit of Oberon distribution hosted on Microsoft Windows. The size is the number of source statements as measured by the Oberon code *Analyzer*. Remember that in a native implementation of Oberon the driver modules of the base system are much larger due to the fact that Windows functions like display primitives need to be implemented in Oberon too.

Table 8.2 does not contain all the modules belonging to the Gadgets system, but only those modules that implement gadgets. These modules are the standard components delivered with a Gadgets distribution. Several applications tightly integrated with the system are left out, notably the modules related to authentication and portable documents.

Module	Implements	# Statements
Bitmaps	Windows bitmap interface	44
Display	Display driver	297
Fonts	Font sub-system	560
Input	Mouse and keyboard input	15
Math	Mathematical functions	24
MathL	Mathematical functions	18
Oberon	Event dispatcher	285
Objects	Object manager	413
System	System manager	786
Win32	Interface to Windows	887
WinPrinter	Windows printer driver	942
Configuration	Startup configuration control	138
Console	Debug output	166
FileDir	Directory support	670
Files	File system	620
Kernel	Memory manager	608
Modules	Module loader	646
Reals	Real number conversion	145
Registry	System configuration	77
Texts	Text abstract data type	1113
Viewers	Viewer manager	246
Total		8700

Table 8.1: Module size of the Oberon base system

Module	Implements	# Statements
Attributes	Attribute and macro handling	757
BasicFigures	Line, Circle, Rectangle and Spline gadgets	1055
BasicGadgets	Boolean, Integer, String, Real, Button, Checkbox and Slider gadgets	1159
ColorTools	ColorPicker gadget	329
Complex	Complex number gadget	58
Desktops	Desktop manager	1748
Display3	Display driver with clipping	1250
Documents	Document manager	561
Effects	Special effects and unportable code	493
Gadgets	Default handlers	1336
Icons	Icon and Iconizer gadgets	1361
Inspectors	Inspector gadget	767
Links	Link handling	144
Lists	List gadgets	1076
NamePlates	Document nameplate gadgets	512
Navigators	Desktop navigator gadget	196
NoteBooks	NoteBook gadgets	528
Organizers	Constraint-based panel organizer	200
PanelDocs	PanelDoc gadgets	201
Panels	Panel gadgets	1600
Printer3	Printer driver with clipping	820
TextDocs	TextDoc gadgets and text commands	687
TextFields	TextField and Caption gadgets	1083
TextGadgets	Text editor gadget	1509
TextGadgets0	Text editor utilities	2124
Views	Camera-view gadget	650
Total		22204

Table 8.2: Module size of the Gadget system

Bibliography

[AGS83] S. R. Ames, M. Gasser, and R. G. Schell. Security Kernel Design and Implementation: An Introduction. *Computer*, pages 14–22, July 1983.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[Bah92] Hicham Bahi. Archives: Eine E-Mail System für Aktive Objekte. Master's thesis, Institut für Computersysteme, ETH Zürich, October 1992.

[BCFT92] M. Brandis, R. Crelier, M. Franz, and J. Templ. The Oberon System Family. Technical Report 174, Departement Informatik, ETH Zürich, April 1992.

[Bie91] E.A. Bier. EmbeddedButtons: Documents as User Interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 45–53. ACM, 1991.

[Ble95] M. Bleichenbacher. Design & Implementierung eines Komponentensystems für eine Klassenbibliothek. Master's thesis, Institut für Informationssysteme, ETH Zürich, March 1995.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications*. Benjamin/Cummings, 1994.

[Brä93] Marcus A. Brädle. Grafische Programmierung. Master's thesis, Institut für Computersysteme, ETH Zürich, September 1993.

[Bro93] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1993.

[BSP$^+$95] N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Fifteenth ACM Symposium on Operating System Principles*. ACM, 1995.

[Car86] Luca Cardelli. Building User Interfaces by Direct Manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, volume 20, pages 233–240. ACM, 1986.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[Com90] Apple Computer. Speed your software development with MacApp. *Develop—The Apple Technical Journal*, pages 155–171, April 1990.

[Com94] Apple Computer. *Dylan Interim Reference Manual*. Apple Computer, Inc., June 1994.

[Cor93] Microsoft Corporation. *Visual Basic 3.0: Programmer's Guide*. Microsoft Corporation, 1993.

[Côt96] Raymond Ga Côté. OpenDoc: Small is Beautiful. *BYTE*, pages 167–168, February 1996.

[Cox90] Brad J. Cox. Planning the Software Industrial Revolution. *IEEE Software*, November 1990.

[CP95] Sean Cotter and Mike Potel. *Inside Taligent Technology*. Addison-Wesley, 1995.

[Cre90] Régis Crelier. OP2: A Portable Oberon Compiler. Technical Report 125, Departement Informatik, ETH Zürich, February 1990.

[Cre94] R.B.J. Crelier. *Separate Compilation and Module Extension*. PhD thesis, ETH Zürich, 1994.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), December 1985.

[Dai] Wei Dai. Crypto++ 2.0 class library. http://www.eskimo.com/ weidai/Crypto.html.

[Dät96] Markus Dätwyler. Executable Content in Compound Documents. Master's thesis, Institut für Computersysteme, ETH Zürich, March 1996.

[fC] ETH Institut für Computersysteme. Module *types*. Part of the Oberon V4 distribution.

[Fra93] M. Franz. Emulating an Operating System on Top of Another. *Software—Practice and Experience*, 23(6), June 1993.

[Fra94a] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zürich, 1994. ISBN 3 7281 2115 0.

[Fra94b] M. Franz. Technological Steps toward a Software Component Industry. In *Springer Lecture Notes in Computer Science*, volume 782, pages 259–281. Springer Verlag, March 1994.

[FvDFH91] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics—Principles and Practice*. Addison-Wesley, 1991.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1995.

[Gib94] W.W. Gibbs. Software's Chronic Crisis. *Scientific American*, September 1994.

[Gla95] Brett Glass. How to Build an Internet App. *BYTE*, pages 211–212, December 1995.

[GM93] Simon L. Garfinkel and Michael K. Mahoney. *NeXTSTEP Programming*. Springer-Verlag, 1993.

[GR93] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1993.

[Gri91] Robert Griesemer. On the Linearization of Graphs and Writing Symbol Files. Technical Report 156, Departement Informatik, ETH Zürich, March 1991.

[Gut94a] Jürg Gutknecht. Oberon — Perspectives of Evolution. In Schulthess [Sch94b], pages 395–406.

[Gut94b] Jürg Gutknecht. Oberon System 3: Vision of a Future Software Technology. *Software — Concepts and Tools*, 15:26–33, 1994.

[Han90] W. J. Hansen. Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit. In *Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages*, pages 23–32. IEEE Computer Society Press, March 1990.

[HH96] G. Dan Huthcheson and Jerry D. Hutcheson. Technology and Economics in the Semiconductor Industry. *Scientific American*, pages 54–62, January 1996.

[Hil92] R. D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *Proceedings of the CHI Conference*, pages 335–342. ACM, 1992.

[HR93] J. Harris and I. Ruben. *Bento Specification, Revision 1.0d5*. Apple Computer, Inc., July 1993.

[HS96] T. R. Halfhill and S. Salamone. Components Everywhere. *BYTE*, pages 97–106, January 1996.

[Ich83] J.D. Ichbiah. *Reference Manual for the Ada Programming Language*. United States Department of Defense, 1983. ANSI/MIL-STD-1815 A.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[Kee89] S. E. Keene. Design Considerations for CLOS. *Journal of Object-Oriented Programming*, pages 68–70, September/October 1989.

[Kno89] Nancy T. Knolle. Variations of Model-View-Controller. *Journal of Object-Oriented Programming*, pages 42–46, September/October 1989.

[KP88] G.E. Krasner and S.T. Pope. A Cookbook for using the Model-Viewer-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *Proceedings of OOPSLA 1986*, pages 214–223, September 1986.

[Lin96] D. S. Linthicum. Integration, Not Perspiration. *BYTE*, pages 83–96, January 1996.

[LM91] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In *Lecture Notes in Computer Science 473*, pages 389–404. Springer-Verlag, 1991.

[Lun89] C. P. Lunau. Separation of Hierarchies in Duo-talk. *Journal of Object-Oriented Programming*, pages 20–25, July/August 1989.

[Mar91] Johannes L. Marais. The GADGETS User Interface Management System. *Structured Programming*, 12:75–89, 1991.

[Mar94a] Johannes L. Marais. Oberon System 3. *Dr. Dobb's Journal*, pages 42–50, October 1994.

[Mar94b] Johannes L. Marais. Towards End-User Objects: The Gadgets User Interface System. In Schulthess [Sch94b], pages 407–420.

[McI76] M.D. McIlroy. Mass Produced Software Components. In P. Naur, B. Randall, and J.N. Buxton, editors, *Software Engineering—Concepts and Techniques*, pages 88–95. Petrocelli/Charter, 1976.

[Mei93] M. Meier. ObjectBase: Objekorientierte Datenbank für Oberon V3. Master's thesis, Institut für Computersysteme, ETH Zürich, May 1993.

[MGD+90] B. A. Myers, D. Giuse, R.B. Dannenberg, B. Vander Zanden, B. Kosbie, D.Pervin, E. Mickish, and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, pages 71–85, November 1990.

[Mic95] Sun Microsystems. *The Java Language Specification*, May 1995.

[MMPN93] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Mös93] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.

[Nel91a] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[Nel91b] M. L. Nelson. An Object-Oriented Tower of Babel. *OOPS Messenger*, 2(3), July 1991.

[Nel91c] Mark Nelson. *The Data Compression Book*. Prentice Hall, 1991.

[Neu91] T. Neuendorffer. ADEW: A Multimedia Interface Builder for Andrew. In *Proceedings of the Multi-Media Communications, Applications and Technology Workshop, Sydney*, July 1991.

[NeX92] NeXT Computer. *The Objective C Language, Release 3.0*, 1992.

[NO90] A. Nye and O'Reilly. *X Toolkut Intrinsics Programming Manual, 2 Ed.* O'Reilly and Associates, 1990.

[NRB69] P. Naur, B. Rendel, and J.N. Buxton. *Software Engineering—Proceedings of the NATO Conferences*. Petrocelli/Charter, 1969.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PHK+88] A. Palay, W. Hansen, M. Kazar, M. Sherman, and M. Wadlow. The Andrew Toolkit: an Overview. In *Proceedings of the USENIX Technical Conference, Dallas*, February 1988.

[Riv92] R. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sal95] Patrick Saladin. Watson—A Smart Browsing Tool. Semesterarbeit, Institut für Computersysteme, ETH Zürich, 1995.

[Sal96] Patrick Saladin. Columbus—Die Entwicklung eines neuen Object-Inspectors für Oberon System 3 und Gadgets. Master's thesis, Institut für Computersysteme, ETH Zürich, March 1996.

[San93] Ravi S. Sandhu. Lattice-Based Access Control Models. *Computer*, pages 9–19, November 1993.

[Sch94a] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., 1994.

[Sch94b] Peter Schulthess, editor. *Advances in Modular Languages*, volume 1 of *Technology transfer series*, Benzstrasse 12, Postfach 4204, D-89032 Ulm/Donau, September 1994. Universität Ulm, Universitätsverlag Ulm.

[Smi87] Randall B. Smith. Experiences With The Alternate Reality Kit, An Example of the Tension Between Literalism and Magic. In *Proceedings of the CHI + GI Conference*, pages 61–67. ACM, 1987.

[Som94] Ralph Sommerer. Script. Online Documentation *ScriptGuide.Text* in the Oberon System 3 distribution, May 1994.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[SUC91] R.B. Smith, D. Ungar, and B. Chang. The Use-Mention Perspective on Programming for the Interface. In *Languages for Developing User Interfaces*, pages 79–89. Jones and Bartlett Publishers, 1991.

[Szy92a] C.A. Szyperski. Write-ing Applications: Designing an Extensible Text Editor as an Application Framework. *Proceedings TOOLS'92, Dortmund*, March 1992.

[Szy92b] Clemens A. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems*. PhD thesis, ETH Zürich, 1992.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[Tea93] The OpenDoc Design Team. *OpenDoc Technical Summary*. Apple Computer, Inc., October 1993.

[Tem94] J. Templ. *Metaprogramming in Oberon*. PhD thesis, ETH Zürich, 1994.

[TGP89] D. Taenzer, M. Ganti, and S. Podar. Object-Oriented Software Reuse: The Yoyo Problem. *Journal of Object-Oriented Programming*, September/October 1989.

[Ude94] Jon Udell. ComponentWare. *BYTE*, pages 46–56, May 1994.

[US87] D. Ungar and R.B. Smith. Self: the power of simplicity. *Proceedings of OOPSLA 1987*, pages 227–242, October 1987.

[VL89] J. M. Vlissides and M. A. Linton. A Framework for Building Domain Specific Graphic Editors. Technical Report CSL-TR-89-380, Computer Systems Laboratory, Stanford University, July 1989.

[Wal92] Roger Waldner. Digitale Simulation unter dem Gadgets UIMS. Semesterarbeit, Institut für Computersysteme, ETH Zürich, 1992.

[Weg90] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1), August 1990.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[WG94] Andre Weinand and Erich Gamma. ET++ – a Portable, Homogenous Class Library and Application Framework. *Proceedings of the UBILAB 1994 Conference, Zürich*, pages 66–92, 1994.

[Wir83] N. Wirth. *Programming in MODULA-2*. Springer-Verlag, 1983.

[Wir88a] N. Wirth. The Oberon System. *Software—Practice and Experience*, 19(9), September 1988.

[Wir88b] N. Wirth. The Programming Language Oberon. *Software—Practice and Experience*, 18(7):671–690, July 1988.

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996.

[Zel] E.J. Zeller. SourceCoder. Private communication.

[Zel93] Emil Zeller. Data Compression Techniques. Semesterarbeit, Institut für Computersysteme, ETH Zürich, January 1993.

Curriculum Vitae

Johannes Leon Marais

July 21, 1967 born in Cape Town, citizen of the Republic of South Africa, son of Marius and Tharina Marais

1985 Matriculated Linden High School, Johannesburg

1986–1988 B.Sc Mathematical Sciences
Randse Afrikaanse Universiteit, Johannesburg

1989 B.Sc Honors Informatics
Randse Afrikaanse Universiteit, Johannesburg

1990 M.Sc Computer Science
Randse Afrikaanse Universiteit, Johannesburg

1991–1995 research and teaching assistant at the Institute for Computer Systems, Swiss Federal Institute of Technology (ETH), Zurich, in the research group of Prof. Dr. J. Gutknecht

1996– Member of the research staff,
Systems Research Center, Digital Equipment Corporation, Palo Alto, California