A FLEXIBLE OBJECT ARCHITECTURE FOR COMPONENT SOFTWARE

by

Angelika Leeb

Diplom-Ingenieur, Computer Science (1992)
University of Technology Vienna

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Signature of Author_____

Department of Electrical Engineering and Computer Science

May 8, 1996

Certified by _____

John V. Guttag

Professor of Computer Science and Associate Department Head

Thesis Supervisor

Accepted by_ ____

F. R. Morgenthaler

Chairman, Department Committee on Graduate Students

A FLEXIBLE OBJECT ARCHITECTURE FOR COMPONENT SOFTWARE

by

Angelika Leeb

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1996, in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

ABSTRACT

This thesis presents a software architecture designed to help end-users to build small applications efficiently by reusing existing software components. In my approach, software is structured into *flexible components*, executable objects that can be modified and combined at runtime. Flexible components are based on existing software interoperability standards, such as COM/OLE or SOM/OpenDoc. They extend conventional component software by extending the component architecture and the functionality of the component engine that manages component interoperation. Flexible components have a component-specific programming interface that allows not only users but other components as well to modify and extend the functionality of a component dynamically. Furthermore, the flexible component engine provides services for addressing, relating, grouping, and composing components in a flexible way. With these characteristics, flexible components combine the flexibility and simplicity of dynamic object environments with the reusability and universality of component software. They promote a new component-centric development style that unifies multiple languages and paradigms within the same application environment and supports incremental development of components.

With the implementation of a prototype, Alego, that simulates a flexible component environment, I have provided some evidence for the feasibility and usefulness of flexible components.

Thesis Supervisor: John V. Guttag

Title: Professor of Computer Science and Engineering

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my thesis supervisor, professor John Guttag, for his enthusiasm, patience, and encouragement. His comments were always so much to the point that I often wondered afterwards how I could have missed them. His critique was always constructive and motivating.

Furthermore, I would like to thank my husband Gunter Leeb for his support in stressful situations and for helping me out with constructive comments when I needed it most.

I'm grateful to the members of my family that have made it possible for me to start this project in the first place. And I'm grateful to professor Michael Loui from the University of Illinois at Urbana/Champaign, since he motivated me to come to the MIT.

Finally, I thank all the members of my research group, especially David Evans, for their useful comments and encouragement.

Contents

1. Introduction

With the spread of cheap PCs and the World Wide Web, computers are rapidly invading new areas in entertainment and education. Untrained end-users are starting to use computers for creative tasks, such as inventing games, creating educational software, writing interactive books, or configuring heterogeneous networks in the home.

The main goal of this work is to make it easier for end-users with little or no computer training to program in the small. Our approach makes the following assumptions: First, users would like to be able to build small applications fast and easily, with no substantial preliminary learning. As a rule of thumb, the learning effort required before one can start building an application should stay in the range of hours. Nevertheless, users expect their applications to be comparable in look and feel to commercial software. Second, we assume that, within the context of small-scale, exploratory programming, flexibility, i.e., the capacity of software to be modified at runtime, has priority over reliability, robustness, and performance. Third, we assume that applications are often built and used by the same person.

1.1 Current Situation: The Developer-User Gap

Traditionally, the software industry has focused on delivering high quality, special purpose application software to the end-user and high-quality, general purpose development software to the professional developer. End-users are not expected to develop their own applications. Professional developers produce streamlined applications, and users work with them efficiently.

Efficient use of people's time is the central concern on both sides. Much effort has been invested in making software development more efficient for professional developers by providing huge collections of pre-fabricated software components. Much effort has been invested in increasing the utility of application software by providing ergonomic, customizable user interfaces.

The desire of end-users to create original applications or adapt software to individual purposes has been largely neglected. On one hand, most development tools are too complex for end-users. The few that are simple enough to attract non-programmers limit the user to toy applications that can't be combined with other software. Applications have to be built from scratch, they do not scale, and, therefore, they are of limited practical use. On the other hand, commercial application software offers little modifiability beyond customization of the user interface. In particular, applications do not let users modify their behavior, or reuse their functionality within other applications.

The bottom line is that end-users are excluded from the software development process and depend entirely on professionals for new features, new behavior, new possibilities. Whether this may or may not be intended by marketing strategies, it clearly hinders creativity and innovation and causes frustration for many users.

1.2 Thesis Content

We address the described problem starting from two premises:

1. The only way to produce high-quality applications with little skill and effort is by taking advantage of functionality developed by others, and
2. The best way to achieve flexible integration of third party software is by breaking applications into small functional components that can be reused independently and recombined efficiently.

The first assumption indicates that, rather than a conventional programming language to *implement* functionality, end-users need a glue to *combine* existing pieces of functionality. The second statement suggests how to organize these pieces in order to be able to combine them in a flexible and modular way.

In this thesis, we present a software architecture that will enable end-users to build applications by modifying and combining existing software components. In our approach, software is structured into *flexible components*. Flexible components are executable pieces of software that can be developed, distributed, and reused independently. They constitute flexible building blocks for applications by being programmable, interoperable, and combinable.

By programmable we refer to the capacity of modifying and extending the functionality of a component dynamically. For that, each component provides a programming interface that can consist of anything from simple point-and-click mechanisms to a full-featured programming language.

Interoperability refers to the capacity of components to communicate with each other. Components interoperate through standardized interfaces, managed by a system component called the *component engine*.

Finally, flexible components can be combined dynamically to build applications or compound components. They cooperate by communicating with each other. They may form hierarchy or grouping relations. They can share parts. They can also be composed to form new components. The component engine provides functionality for coordination, relation, and composition of components.

Flexible components simplify programming for several reasons. First, since components are modifiable, they can be reused for different purposes. Therefore, when building an application, most of the functionality can be taken from existing components that can be adapted to the specific goal in mind. Second, the programming interface of each component can be simple and efficient, since it is tailored to the specific purpose of the component. Third, components can be modified, extended and combined *in-place*, i.e., within their application environment. This allows users to observe the effects of modifications immediately, and it eliminates the need for mastering a specific or even several different software development packages in order to build applications.

Furthermore, flexible components can be developed incrementally, which enables users of varying experience to contribute. For instance, professional programmers may implement a generic component using an efficient object-oriented language, the amateur programmer may extend the component's functionality using, say, its built-in Visual Basic interface, and the non-programmer may further customize the component's behavior using a simple dialog box.

Finally, the flexible component environment is open-ended, and applications built out of flexible components are easily extensible. The environment can be extended to support new programming languages and paradigms by simply introducing new components. Applications can be extended by modifying integrated components and incorporating new ones. In addition, different applications can be combined by relating their components. In fact, the boundary between applications becomes as malleable as are their components, and the distinction between applications becomes purely conceptual.

We have implemented a prototype, Alego, that simulates a flexible component environment. Like flexible components, Alego objects include a programming interface. The flexible component engine is simulated by the Alego runtime environment. Alego introduces several new features, e.g., a dynamic multi-cast addressing scheme and a grouping relation.

The flexible architecture of Alego objects has proven useful for small, exploratory applications. For instance, simple games can be extended and modified while playing them, which makes them more variable and simplifies their development. However, we have only experimented with small applications, developed and used by a single person, using the same programming language for all objects. Multiple users and multiple languages will certainly introduce a multitude of problems related to complexity, performance, and reliability that have yet to be explored.

The following section discusses related work. Chapter two introduces flexible components. Chapter three presents the prototype environment Alego. In chapter four, we evaluate the presented work by pointing out advantages and limitations of flexible components and reporting on our experience with Alego. Finally, we conclude with a look at work that remains to be done.

1.3 Related Work

In this section, we discuss two approaches that tackle different aspects of the described problem. Together, they possess the advantages that we consider as essential for user-friendly software development: Dynamic object environments support a flexible and interactive programming style; component software promotes the reuse of available functionality.

1.3.1 Dynamic Object Environments

Dynamic object environments represent the traditional approach to user-friendly programming. Figure 1 situates dynamic object environments within the context of programming tools.

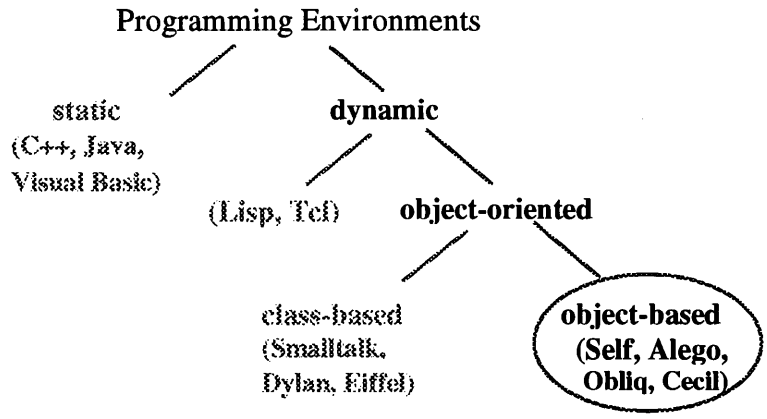

Figure 1. Classification of programming environments. The circle marks dynamic object environments.

Dynamic programming environments (Lisp [Ste90], Tcl [Ost90, Bor94]) are based on an interpreted (or incrementally compiled) language and merge runtime and development time of an application. They support a flexible, interactive programming style: Any modification of code has an immediate effect on the behavior of the running application.

Object-oriented dynamic environments add structure and uniformity to the flexibility. Programs are structured into objects that are treated in a uniform way. Object environments can be classified roughly into class-based and object-based, depending on whether the main programming activity revolves around classes or concrete objects (instances) [Mal95, Lie86, Weg87].

In class-based environments, objects are created as instances of previously defined abstract descriptions called classes. Most object-oriented programming languages, such as C++ or Java [Com96, Dec95, Sun95], are class-based. Dynamic class-based programming environments, such as Smalltalk [Kho95, Sut95], Eiffel [Mey92], and Dylan [App95b, Dum95] aim at increasing the programmer's productivity while preserving the powerful abstraction mechanisms of the class-based paradigm. In general, class-based programming is considered as difficult to learn, and the target audience of these dynamic development tools is professional developers of large-scale production quality software [And95, App95b, Bae94, Joh95, Raw95].

Here, we are interested in object-based (also called *prototype-based*) environments which are targeted to the explorative and casual programmer. We refer to these environments as *dynamic object environments*. In dynamic object environments, such as Self [Hoe93, Smi95, Ung91, Ung92], Obliq [Car94], and Cecil [Cha93, Cha95], objects, called *prototypes*, exist on their own (i.e., they are not instances of a class), and new objects are created by cloning existing ones. A prototype is a collection of data or method members called *slots*. Prototypes usually offer a user interface that supports direct manipulation of their structure (i.e., the number, names and types of their slots), and behavior (i.e., the functionality of their method slots).

Dynamic object environments have been developed in order to simplify programming under the assumption that programming is easier to learn and more straightforward when manipulating

concrete objects instead of abstract class definitions [Mal95, Smi95]. In fact, environments such as Self allow one to build small, exploratory applications relatively fast and easily. The user-friendly interface, in which objects and their relationships are represented visually, promotes intuitive understanding for the programmer and a fast start for the novice.

However, the benefits of simplicity, uniformity and directness of dynamic object environments are valid only as long as applications are small.

First, because of the inferior abstraction mechanisms, prototype-based applications are less scaleable than class-based ones. With increasing number of objects, applications become difficult to look through since all structure is dynamic. In order to understand the role of a particular object in a program, it is necessary to examine all its slots and follow all reference pointers to other objects.

Second, prototype-based systems inherently offer little support for structuring collections of objects with common characteristics. When many objects share slots on a permanent basis, it becomes desirable to represent the shared parts separately and in a centralized way, so that space can be saved and updates will be shared automatically.

Delegation has been introduced in prototype-based systems as an alternative to class-based inheritance (the primary sharing mechanism in class-based systems). Delegation establishes a parent-child hierarchy among objects. The child can delegate messages that refer to some unknown slot to its parent. The parent, if owner of the missing slot, will handle the message on behalf of the child; otherwise, it will pass the message further up in the hierarchy.

Besides delegation, new kinds of objects are introduced that represent repositories for shared information. *Traits* are externalized groups of methods that can be shared among several objects. *Maps* are elements that contain the structural information of an object. As discussed in [Mal95], the externalization of shared object parts leads to a break-down of the concreteness principle in prototype-based environments by introducing abstract objects that do not behave like other objects. Also, with increased sharing of information, interdependencies in a dynamic system become complex, and the basic advantage of concreteness being simpler to understand than abstractness vanishes.

An even bigger disadvantage of dynamic object environments is that they represent closed worlds in the sense that they offer little or no interoperability with other applications. Objects rely heavily on environment-specific features, and, in general, they cannot be reused in a different context. Nor is it possible to integrate objects created by some other development tool. Hence, even if the programming language is simple enough, the learning effort often doesn't pay off because very soon the limit of what can be done is reached, and users have to switch to another tool. And, because of the lack of interoperability between these environments, much work has to be redone. Next, we describe an approach that solves this problem by making applications interoperable.

1.3.2 Component Software

Component software is an object-based software model aimed at efficient and incremental software development. The main idea is to break monolithic applications into reusable, binary components that can be developed, distributed and upgraded independently. In essence, this

comes down to providing standard mechanisms for interoperability between applications and components. If components can interoperate, they can be combined to build larger applications in a flexible and incremental way.

A component is a binary (i.e., compiled) module with standardized interface. That is, the component interface is specified separately in an Interface Description Language (IDL), and its binary representation conforms to the underlying component software standard. Component interoperation is managed by a component engine that uses the interface specification and the binary layout of interfaces to resolve method calls at runtime.

There are two major competing standards for component software, Microsoft's COM/OLE [Box95, Bro94, Bro95] and IBM/Apple's SOM/OpenDoc [App95, CIL94a, CIL94b, IBM94, Pie94]. While OpenDoc's design seems to be more consistent, OLE technology is more advanced in terms of available components. OLE Controls [Mic95, Muk96] arguably constitute the most advanced components of today. Typically, OLE Controls represent user interface objects such as buttons or list boxes. They expose properties that can be set and methods that can be called from within the containing application. In addition, they generate notification messages on certain predefined events (e.g., mouse click).

The main goal of existing component software standards is to speed up the development of application software by enabling developers to reuse ready-to-run, pre-tested components. Ideally, 70% to 90% of the functionality of a new application would be provided in generic fashion by pre-built modules. Only a small amount of additional code (10% to 30%) would be required for fine-tuning and custom-tailoring [CIL94a].

With respect to reusability, component software has two key advantages over traditional object-oriented programming: First, components interoperate at runtime, and can therefore be integrated in applications in a flexible and dynamic way. Second, interface descriptions are separated from the actual implementation of a component, which makes it possible to develop and update components independently. To implement code that refers to a component, only the interface specification of that component is needed.

Professional developers benefit from component software in many ways. They can create new components or derive them from existing ones; they can distribute them independently and integrate third-party components with their applications. The structure and granularity of components makes them particularly efficient as building blocks of large-scale integrated business solutions with standard functionality and user interfaces customized to the specific configuration of a company.

However, from the end-users point of view, existing component software approaches fail to exploit their potential of reusability. In fact, existing approaches are based on the assumption that the development of components and their integration in applications is done by professionals, and end-users will just use these applications in a well-defined, anticipated way. Accordingly, very little has been done to enable end-users to build components or applications based on component software.

First, developing a component is a notoriously complex task and usually requires several months, perhaps years, of background training, even for professional programmers [Bro95, Muk96].

Second, components provide little support for dynamic modification. They typically have a specific functionality that cannot be modified or extended. Also, their interfaces are determined at compile time and immutable.[1]

Finally, in order to build applications that use components, one has to use a conventional programming tool. Even the more user-friendly tools that support component integration, e.g., Visual Basic, require at least several weeks of learning and are therefore too complex for the average end-user. Furthermore, users depend entirely on developers for component functionality. For one, in the growing market of components it becomes hard to find out what's available. On the other hand, because available components will usually not fit exactly the requirements of a specific application, a lot of messy programming is often required to make their behavior appear slightly different. Finally, new versions of the development platform may not support older components, because the component standards are constantly evolving.

To let end-users exploit the advantages of component software, we propose a *flexible* component software model. Our approach extends conventional component software two ways: First, it extends the component architecture in order to support dynamic modification and extension of individual components. Second, it extends the functionality of the component engine in order to support dynamic coordination and combination of components.

[1] [Har93] and [Oss95] describe a class-based approach, called subject-oriented programming, to augment the flexibility of components in order to increase their reusability.

2. Flexible Components

In this chapter we present the concept of flexible components and outline the design of a simple flexible component software model.

2.1 Overview

Flexible components are intended to combine the best attributes of component software and dynamic object environments. The goal is to build components that are

1. Autonomous. By this we mean that they can be used in a variety of environments, i.e., they depend only upon a limited set of global services that are independent of the application making use of the component.
2. Flexible. By this we mean that components can be modified in a variety of ways at runtime by both users and other components. Methods can be added, deleted, or modified. Furthermore the structure of the state can be changed.

The next two subsections outline our approach to achieving these goals.

2.1.1 Autonomy

We achieve autonomy in a similar way as existing component software approaches: Components are executable objects that communicate with each other at runtime, managed by a component engine.

In order to be able to interoperate, the binary format of components is standardized. For instance, the interface of a component has to be represented in a specific format so that it can be accessed in a uniform way by the component engine and other components. Flexible components can be based on any of the existing binary standards for component software.

Component interoperation is managed by a component engine that runs as a separate process. In COM, the component engine loads components and establishes connections between them. For flexible components, the component engine provides additional services for coordinating components dynamically. Later in this chapter, we will describe the functionality of the flexible component engine.

2.1.2 Flexibility

We achieve flexibility by providing each component with a programming interface.

The programming interface offers the possibility to *modify* the functionality and the interface of a component in addition to *executing* its functions. This generalization of the component interface has two important consequences. First, each component can be programmed in a custom-tailored, component-specific way. Second, programming an object by simply sending it messages can be done by other components as well as by the user. This opens a new dimension to object interoperability: Objects can modify the behavior of other objects.

As an example, suppose we have a database component that offers an SQL interface. When sending an SQL *select* statement to this component, it returns the corresponding data records. Through its programming interface, we can also send the component a message that replaces

the SQL interface by a dbase interface. The modified component will understand dbase commands instead of SQL statements.

The programming interface might offer simple scripting or point-and-click mechanisms to add or remove methods and state variables. Adding state variables creates storage to store useful information in a component. Adding methods extends the functionality of a component.

As an example, imagine a simple edit box component that contains text to be edited. By adding a method *save* we can extend the component's functionality to save text to a file. The method *save* might open a dialog that prompts the user to type in a filename. By adding a state variable *filename* to the component, we can automate the save operation to use the stored value instead of typing in a filename each time *save* is invoked.

For modifying or implementing methods, the programming interface might provide a simple command interface to link method names to available functions. Or, it might offer a full programming language in which methods can be implemented.

As an example, consider a new method *find* that one might add to the edit box component. Suppose, a *find* function is available that takes a window handle,[2] opens a dialog to type in the expression to search for, searches the text in the window and moves the cursor to the corresponding place. A simple programming interface could let us add a new method and link it to the existing *find* function by specifying the corresponding parameters. Alternatively, the component might offer a C programming interface, in which case we could implement the method in C.

Finally, the programming interface might offer multiple languages, adaptable to a variety of preferences. For instance, a component could offer Basic as an easy-to-use language and C for performance-critical methods. Method sources could be translated automatically to the language of choice (as far as possible).

Flexible components are more flexible than conventional OLE or OpenDoc components since they support dynamic modification of structure and functionality. They are also more flexible than objects in dynamic programming environments in the sense that the programming interface of each component can be designed according to its specific purpose and functionality.

2.1.3 Application Fields

Flexible components can be assembled quickly to build small, highly dynamic experimental applications. Some application fields that would especially benefit of this flexibility are education, research and experimental computing, interactive books, combinatorial and thinking games, and configuration tools for home systems.

For instance, games could be made more exciting and variable by making it possible to add and modify rules, or combine different games. Generic components for game boards, pieces, letters, words or numbers, arithmetic operators and timers, with special interfaces to easily define rules could provide a base for creating modifiable games.

---

[2] In a window-based graphical user interface environment, window handles are reference pointers to windows.

2.2 Design Principles for Flexible Components

A component software system should be universal enough to satisfy the needs of many users and enable the production of widely reusable components. It should also be user-friendly enough to support component development by end-users, including non-programmers.

2.2.1 Universality

The component software environment should accommodate a diversity of applications. In particular, it should support

- **Universal Functionality**. Components of variable degrees of flexibility should fit into the standard. It should be possible to develop robust, specialized components as well as flexible, generic components with reasonable performance.
- **Universal Development**. Professional as well as non-programmers should be able to develop components. It should be possible to develop components incrementally, using a variety of programming languages and methods. In particular, it should be possible to extend and combine components in-place, i.e., without having to enter a separated development environment.
- **Universal Connectivity**. Components should be mobile. They should be able to interoperate across address spaces and, possibly, platform boundaries, e.g., over the World Wide Web.

For maintaining universality in the future, the component software standard should be extensible. In addition, the component architecture should allow dynamic tuning of structural and performance parameters. For instance, when a component grows in complexity, the user should be able to consolidate its structure in order to gain performance at the cost of flexibility.

2.2.2 User Friendliness

The system should be simple to learn and easy to use, and it should encourage spontaneous, carefree experimenting with objects. The following aspects are important:

- **Documentation**. Each component should provide precise and concise documentation of its behavior and interface. When modifying a component, its documentation should be invalidated or, if possible, updated automatically. For instance, when deleting data or method members, the corresponding documentation could be deleted. Or, when adding a new method, documentation that includes the number and types of its parameters could be added automatically.
- **Fast Start**. A newcomer without programming experience should be to build a simple component within, say, 20 minutes from seeing the system for the first time. Few basic principles should be needed to manipulate objects, or start an application. Ideally, users should be able to find out the basic techniques by trial-and-error, without a tutorial or help system.
- **Smooth and Scaleable Progress**. Learning should proceed smoothly, i.e., in small steps that lead to visible progress. One effective way to learn is by picking an example

component and modifying it little by little. The system should provide a pool of sample components of different complexity.

The more experienced a user becomes, the more possibilities the system should offer to him. In other words, the learning effort/reward cycle should remain as constant as possible.

- **Spontaneous Action.** In order to motivate users to experiment, the system should be liberal, robust, and responsive. It should impose a minimum of rules and restrictions. It should provide backup strategies to let users resume at a safe state in case of errors. Finally, the system should provide informative feedback on demand or in critical situations, without interrupting the flow of action more than necessary.

2.3 A Flexible Component Software Model

We distinguish two main agents in the flexible component software model: The flexible component engine that manages component interoperation and the individual components. The model specifies the

- Architecture of flexible components
- Functionality of the flexible component engine (FCE)
- Mechanics of component interoperation

The component architecture should ensure that components can be developed, reused, and upgraded independently and that they can be composed, and possibly decomposed, to form applications efficiently. The underlying component engine (FCE) should support efficient manipulation, coordination and interoperation of components.

We base our flexible component software model on an established component software standard (COM/OLE or SOM/OpenDoc) in order to increase its practical impact and take advantage of existing functionality. Some ramifications of this are discussed in section 2.3.3.

2.3.1 Component Architecture

A flexible component is a binary object. Its functionality is encapsulated in methods and data members. Its interface permits both controlling the component's behavior by executing methods and modifying its functionality by adding, deleting, or modifying methods or data members. In the following, we describe the parts of a component:

- **Identity.** Each object receives a system-wide unique, immutable identity at creation time. This identity is known only to the FCE which can pass it to other components when necessary. When an object is moved to another address space, it is destroyed and a clone with a new identity is created at the destination site in order to preserve the uniqueness of the identity.
- **Name.** Each object has a name that is mutable, i.e., it can be changed by users or other components, and need not be unique.
- **Visual Representation.** Each component has a (mutable) visual representation. The visual representation displays the status of a component that can be inactive or active. In inactive status, a component might be represented by an icon. In this state, the interface is usually

restricted, for instance, a component might not be able to react to certain messages. When activated, the component displays its characteristic visual features, e.g., user interface elements such as windows and gadgets. Not all components need to have visual characteristics. Still, invisible components should be recognizable by some alternative means, e.g., a symbol that can be hidden or displayed. In some cases, the combination of components might result in a unification of their visual representations (e.g., when merging components).

- **Data and Methods**. A component's data and methods represent its functionality. Some of them might be modifiable or removable, others not, depending on the implementation of a specific component. Note that in contrast to the traditional, class-based object model (as well as the conventional component software model), methods now may form part of the state of a component. Moreover, even (part of) the *structure* of methods and data members belongs to the state, since it can be modified dynamically.

- **Interface**. From a functional point of view, we distinguish a command interface to *access* a component's functionality and a programming interface to *modify* it. The command interface consists of the object's methods and data at a certain time. It is used to execute methods and set or retrieve data values.

  The programming interface determines the flexibility of a component which may vary among components. For instance, some components may allow users to modify the code of certain methods, others not. The programming mechanisms offered by different interfaces will also vary. For instance, some components may offer a simple point-and-click interface, others a scripting language, or even a full-featured programming language that offers the option to compile methods. In general, each component will expose a programming interface that suits its intended purpose.

  From a structural point of view, we distinguish a static and a dynamic part of the component's interface. The static part is the part of the interface that is determined at compile time, i.e., by the initial programmer of a components, and cannot be modified. The static interface contains the functionality that all components have in common. For instance, the basic functionality used for component interoperation is static. This functionality should be static since the component engine as well as other components rely on it. Some component-specific features might also be determined statically if the initial programmer of a component decides that this features should remain unchanged.

  The dynamic part corresponds to the component-specific functionality that might be modified or extended by the users of a component. It can be viewed as an interpreter that interprets incoming messages. Some messages will result in the execution of a method, others will effect changes of the component's structure or functionality. Changing the functionality of the component will often change its interface, for instance, if new methods are added or existing methods are deleted.

- **Documentation**. A component provides a documentation interface for both the user and other components. The documentation can be modified and should consistently reflect a component's capacities and its interface.

2.3.2 Functionality of the Flexible Component Engine

The flexible component engine (FCE) consists of system components that provide global services to manage component interaction.

In existing component software standards, the component engine knows nothing about the semantics of component interoperation. Relational functionality, such as hierarchy or sharing relations, is implemented by individual components and their interfaces. This makes component software more portable and extensible, since new relations can be introduced by simply defining new interfaces. However, components tend to become complex, and, as the number of interoperating components grows, the performance goes down because there are no mechanisms for consolidating groups of cooperating components. Performance problems are even more critical for flexible components, since a message exchange will require at least one more level of indirection. It becomes desirable to extend the functionality of the component engine to support more efficient management of component groups.

Beyond the basic services that can be found in existing component software standards, the FCE provides extended functionality for manipulating, coordinating, and combining components dynamically. In particular, it offers functionality for

- **Creation and Destruction of Components.** New components are created by cloning, i.e., duplicating, existing ones. The FCE provides the necessary mechanisms to initiate a cloning operation, and create a new object with a new identity. Similarly, it provides functionality for deleting components. Before deleting a component, it can be saved to a persistent form. Then, the runtime object is unloaded and cleanup operations have to be performed to maintain the consistency of components affected by the deletion. For instance, if a component is deleted that is vital for the proper functioning of other components, the FCE might need to shut down these other components too.

- **Message Passing and Addressing.** Components interact by exchanging messages. This message exchange may occur within the same process, or between separate process spaces or even between different machines. The FCE is responsible for translating addresses transparently across process and network boundaries.

  Addressing flexible components is done in a different way than in existing component software standards. Since most of the behavior and interaction of flexible components is programmed at runtime, they require a more dynamic and user-friendly addressing scheme. In particular, users have to be able to refer to other components, or to component groups, in an intuitive way. Accordingly, flexible components are addressed by their name, not by their identity. Furthermore, in order to be able to address groups of components efficiently, we use a multi-cast addressing scheme. An address consists of a list of names, a scope identifier, and a list of descriptors. The scope depends on the component's place within the global hierarchy relation (see below). A descriptor expresses conditions that narrow down the list of addressed components. For instance, you can address all components with a certain name within a certain scope that have an attribute *color* with value *blue*. This extended dynamic multi-cast addressing scheme is described in the next chapter.

  The FCE provides the necessary functionality for the naming scheme. It has to establish connections between components and translate addresses into component identities. It

depends on the implementation whether this translation is done for each message or the FCE establishes a connection between components so that they can exchange consecutive messages directly (as long as no changes are made that affect the respective addresses).

Note that an address might correspond to different components at different times, depending on their actual name and scope. The naming consistency, i.e., the correct mapping from names or addresses to identities, must be ensured by the implementation of the message passing algorithm.

- **Establishing Relations between Components**. Several relations can be established between components. A parent-child relation establishes a component hierarchy. The place within this hierarchy determines the context or *scope* of an object. When forming a parent-child relation, the state or behavior of the involved components might be influenced by scope-dependent characteristics. For instance, the parent component could determine that all its child components share their *color* attribute. A new child that has such an attribute will then automatically adapt its color to the color of its siblings. We call this adaptation of characteristics *adaptive inheritance*.

  Sharing is another relation. Flexible components can share any part of their state. Since modifiable methods form part of the state of a flexible component, components may share behavior. While the parent-child relation is based on a component's identity, i.e., it is invariant to a change of name, the sharing relation might be based either on the identity of a component, or on its name, scope or other mutable characteristics. For instance, one might specify that all components named "number" share a method *add*. Alternatively, one might want to create several components that will permanently share some method, regardless of their name or other attributes.

  Finally, we briefly mention a grouping relation that groups components to indexed arrays. Grouping components is useful for processing them as arrays or lists. The next chapter describes one possibility to define these three relations.

  The FCE provides the functionality for establishing and removing relations. Furthermore, the FCE might also manage and maintain some relations or aspects of them. The advantages of managing relations globally are that it simplifies individual components and ensures a global semantics of relations. The disadvantage is that central management creates a bottleneck at the FCE and is less flexible in terms of innovations.

- **Component Composition**. The FCE provides operations to compose components. Component composition merges several components into a single one that unifies some of their characteristics. This unification consolidates the structure and functionality of a group of closely related components. It simplifies clusters of cooperating components and enhances their performance by eliminating the need for intercommunication and maintaining relations. Component composition operations can be defined in a similar way as in [Oss95].

2.3.3 Component Interoperation

The mechanics of flexible component interoperation are based on an existing interoperability standard. This has some consequences for the implementation of flexible components. For instance, existing standards assume statically defined component interfaces. Consequently, the

interface of a flexible component has to be implemented as static interface that conforms to the underlying interoperability standard. The dynamic features of a component have to be implemented on a higher level, and they can only be accessed through its static interface.

Flexible components have a standardized binary format and a separate interface specification. Both the binary format and the specification language are defined by the underlying component software standard, e.g., COM. The interface specification describes the static part of the interface, i.e., the part that is determined at compile time. It is used by the component engine to identify the component's communication entry points. The dynamic interface of a flexible component, i.e., its flexible data and method members, is accessed through a special static method that checks for the corresponding member and passes the message to it. For more information on how components interoperate in COM or OpenDoc, see [Bro95] and [App95].

3. The Alego Environment

Alego is a prototype-based dynamic object environment. It is designed for building small, simple, very flexible, exploratory applications such as simple board games or mathematical models (e.g., state machines). It can also be used for ordering thoughts and ideas, or creating interactive text books.

Our main intention was to set up an experimental environment fast and easily in order to be able to explore the potential and study the limitations of flexible components, which are simulated by Alego objects. By implementing Alego as object environment, we have avoided the complexity of implementing, e.g., OLE components, and we have preserved the independence from any component software standard.

3.1 User Interface

Alego's user interface provides only a small set of features, mainly controllable by the mouse. It also offers an extensible tutoring facility that assists user activities with comments and suggestions. All other functionality is associated with individual objects. Each Alego object exposes its data, methods, documentation, programming language, and command interface within its information window.

Alego is a single mode environment that does not distinguish between development and use of an application. The user can modify any aspect of an object. Objects are dynamic entities that behave as programmed, and immediately readjust their behavior on changes.

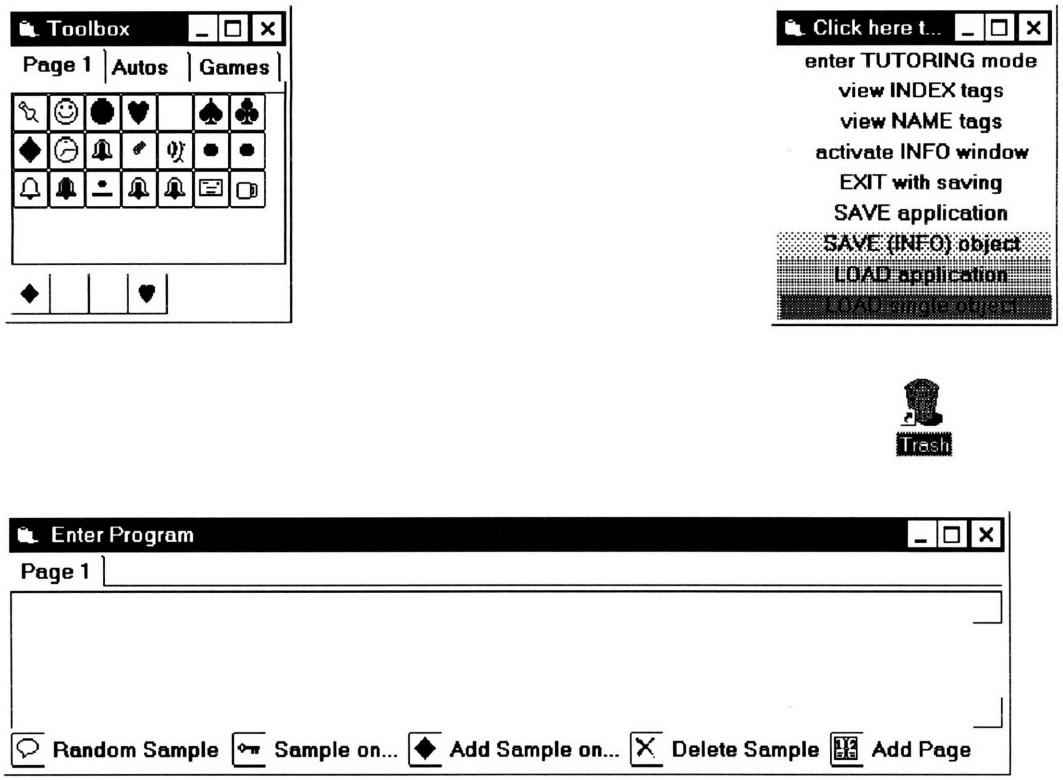

Figure 2. The Alego User Interface. When starting Alego, several windows appear: The Toolbox contains available objects to load; the Status window offers a series of commands and options; the Trash serves to delete objects; the Command window provides a general interface to create and program objects.

The user interface consists of several windows, some of them organized as books. A book is a window with multiple tabbed pages. These windows are:

- The *Toolbox* is a repository of all available prototypes. Each prototype is represented as a small image. Users can instantiate objects by double-clicking them or dragging them onto a page. Users can also drag any object to the toolbox in order to produce a reproducible prototype of it.

  In Figure 2, the toolbox contains several basic objects such as the basic window object, picture and text page objects, page-tabs objects to organize several pages on a window, a timer object, and several simple general purpose block objects. These prototypes or *tools* form the basic building blocks of an application.

- The *Trash* has the expected functionality of deleting objects that are dropped into it.

- The *Status window* consists of a menu with general functionality such as save, load, exit. It has some switches for enabling or disabling the tutor, or showing name tags with the objects.

- The *Programming window*, which is also organized as a book, serves as a global command and programming interface. It has several facilities that assist users in programming objects. For instance, the user can fetch example programs or store additional samples. Furthermore, there is a facility that suggests possible continuations while the user is typing a command.
- As users instantiate objects, an *Information window* that exposes the object's interface pops up corresponding to the active object. One object in the system is always the active one. The active object determines the default scope and default address.

3.2 Brief Example: The 15-Puzzle

In order to provide an overall feel of Alego, we present a small example before going into details. The 15-puzzle consists of a 4 × 4 board with 15 numbered tiles and an empty place to which one of the adjacent tiles can be moved. The goal of the game is to bring the tiles in the correct order from 1 to 15.

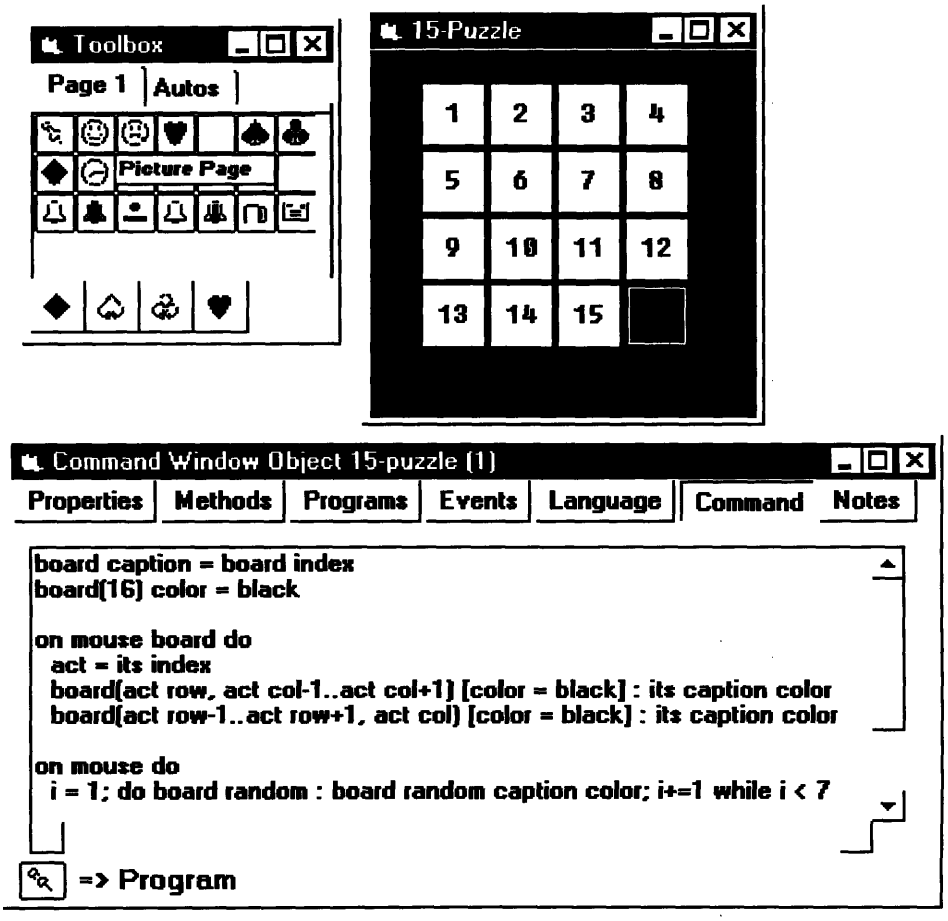

Figure 3. The 15-Puzzle. The command window shows all commands and programs needed apart from the visual construction of the board.

Figure 3 shows an example implementation of the 15 Puzzle in Alego. To create the *15-Puzzle* object (upper right corner in the picture), we instantiate a new *Picture Page* object from the toolbox. We also instantiate a group of objects that form the board, using the built-in group instantiation feature. We call the new group object, a two-dimensional array of simple squares, *board.*

The command window of the *15-Puzzle* object contains all code needed to initialize and play the game. The first line is a multi-cast command which sets the caption of each tile on the board to its index. The second line sets the color of tile 16 to black, which makes this tile appear like an empty space on the board.

The next four lines constitute a program that implements the rule of the game, namely, moving a tile to the adjacent empty space. More specifically, it states that when the user clicks on some tile, we swap its color and caption with the (adjacent) empty space. Swapping color and caption has the same apparent effect as moving the tile to the empty place.

This simple program illustrates the expressiveness of our addressing scheme. The test for adjacency has been buried in the two-dimensional addressing expression

```
board(act row, act col-1..act col+1) [color = black]
```

that simultaneously addresses the tiles in between the left and the right neighbor of the one that has been clicked. The attached test expression in square brackets checks for each of these tiles whether its color is black (i.e., it is the empty space). The tiles that do not satisfy this condition are discarded. If one of these tiles is indeed the empty space, it swaps (":" is the swap operator) its color and caption with the selected tile (which is addressed using the shortcut "its"); otherwise, nothing happens. Furthermore, the addressing engine automatically discards references to rows or columns outside the board boundaries so that the same expression works for border tiles.

Finally, we define a program for mixing up the board before starting the game (last line in command window). When clicking anywhere in the *15-Puzzle* page (i.e., within the area that surrounds the board), it performs six random inversions (an even number of inversions will always yield a solvable instance of the puzzle). Note that the capacity to create properties that can hold data eliminates the need for variables in our language. For instance, the property *i* has been added to the *15-puzzle* object in order to store a counter variable for the *do-while* loop.

All commands and programs are addressed to the *15-Puzzle* object, since they have been typed in its command window. While commands (the first two lines) are executed immediately, programs will be stored as new methods of the *15-Puzzle* object. After that, whenever the object receives a message "mouse" (mouse-click event) or "mouse board" (mouse-click event directed to a subobject named *board*), the corresponding program will be executed.

3.3 Object Model

Alego objects have a state and an interface through which the state can be accessed and modified. The state of an object describes its mutable characteristics. The state of an Alego object includes the following:

- *Properties* are data members whose values can be viewed and modified. Properties can have different access permissions, such as "read-only" (the value cannot be modified), or "non-removable" (this property can't be deleted).
- *Methods* are compiled operations that cannot be modified. There are several types of methods: Some methods are functions with no side-effects and can be added to any object. Others depend on specific properties or internal characteristics of an object and cannot be transferred to other objects. Some methods can be removed, others not. Finally, some methods can be delegated: An object can execute a method on behalf of another object. This mechanism will be explained later in the context of the hierarchy relation.
- *Programs* are modifiable operations, i.e., "methods" for which the source code is available and can be modified. In Alego, all objects use the same programming language for their programs, a simple scripting language called *Lago*. Programs are typically used to determine an object's reaction to certain events (e.g., a mouse-click event).
- *Documentation* gives the user information about an object and its features and can be edited by the user.
- The *graphical representation* displays the characteristic features of an object. All Alego objects represent user-interface elements and have a graphical representation, e.g., a window or a command button. The appearance of an object can be modified through some of its properties (e.g., a *color* property).
- The *Toolbox image* of an object is a small bitmap that symbolizes an object within the toolbox.

The distinction between methods and programs represents a simplification of the flexible component architecture. A flexible component can have methods that expose their code and others that can be invoked but not modified. Or, there could be methods that cannot be modified arbitrarily but can be composed out of various pieces of code that can be rearranged in different ways. Or, the same method could be modifiable by some clients and have a fixed behavior for others. In Alego, we have chosen to look at the two poles of fixed and arbitrarily programmable methods.

The interface of an Alego object defines the set of all messages it understands. Strictly speaking, this interface forms part of the state, since it is modifiable. For instance, adding a method extends the set of meaningful messages for an object. This set of messages can be divided into:

- *Lago* expressions. *Lago* is the scripting language in which Alego objects are controlled and programmed. A *Lago* expression may contain references to an object's properties, methods, or programs. It may also contain references to other objects, using the extended addressing scheme of Alego (described later). Such a message may affect the state of an object in many ways, for instance, by modifying a property value, or by adding a new program that determines the future behavior of the object. Users can pass expressions to an object through the command interface of its information window, or the global command interface of the Alego system. Other objects can pass them through the message passing system of the Alego runtime engine (referred to as the *default* message passing system).

- *Events*. Event messages are constants that represent a user action, such as pressing a key or a mouse button, or an action of an object, such as generating a timeout. User action events are typically passed to the object through its graphical representation. They can also be simulated by sending the corresponding constant instead. Other events are passed to an object by the default message passing system.
- *Point-and-click* Messages. These are user actions that access the object through its information window. While events typically serve to *control* the object, point-and-click messages are used to *program* it (i.e., to modify its structure or behavior). For instance, the property page of the information window offers point-and-click mechanisms to add a new property to an object. Note that while *Lago* expressions and events might be generated by other objects, point-and-click messages can only be generated by the user.

According to the types of messages and the different ways in which messages can access an object, we identify two parts of an object interface:

- *Visual* interface. The visual interface consists of the graphical representation of an object and its information window.
- *Textual* interface. This interface consists of the programming language *Lago*, as well as an object's methods, properties and programs.

The textual interface of an object can be viewed as an interpreter that interprets incoming messages. If an expression cannot be parsed by the interpreter, the rest of the message is simply discarded. Changes that have been made already based on the part of a message that has been parsed are *not* undone. Wrongly typed expressions (e.g., assigning a value of the wrong type to a property) are also discarded. However, because of the polymorphism in *Lago* and the liberal type conversion, it seldom occurs that an expression does not type check.

3.3.1 Addressing Alego Objects

Designing a compact yet precise naming scheme for objects is one of the most challenging tasks in a dynamic object environment. Most systems provide a hierarchical naming convention, where the parent's name is a prefix of the child's name. Sometimes the user can use relative addressing as a shortcut. We propose an *extended dynamic multi-cast* addressing scheme that supports the simultaneous addressing of groups of objects, and the selective addressing of objects depending on their dynamic characteristics.

An address consists of an optional scope identifier, a list of names, and a list of test conditions. This address is resolved into a set of zero or more matching identities. The corresponding command is then executed for each object in that set. For instance, "all square circle color = green" states that all existing object instances in the system named "square" or "circle" should be colored in green. Note that the same address may correspond to different objects at different times. Furthermore, Alego supports a one and two dimensional array addressing.

A scope specifies a subgraph of the hierarchy tree. For instance, "page text share color" states that all objects named "text" that are direct or indirect subobjects of the active page (i.e., the currently visible page of the active window) should share their *color* property. The default scope corresponds to direct child objects of an object.

The name of an Alego object is a non-removable property. When an object is created by cloning a prototype, its name will be the same as the name of the prototype. The name can be changed by the user at any time. In case of a name change, all references to the former name in existing programs will not affect an object anymore. Note that the name of an object is totally unrelated to its identity, which is system-wide unique and immutable.

In addition to name and scope, the user can specify test conditions that narrow further down the set of addressed objects. For instance, "button [color = red and left > 100]" refers to all red objects with name "button" whose x-position is > 100 relative to the local coordinate system. Test conditions simplify programming by eliminating control structures, e.g., nested loops and conditionals.

Alego supports grouping of objects by overlaid indices. For addressing specific elements of such an ordered array of objects, there are a variety of array specifiers that can be included in an address after the name. For instance, "board (1,2)" designates the element in the first row, second column of an array named "board" (supposedly arranged in a matrix shape). A more sophisticated example is "board (1 3, 2..4)" which refers to the second, third, and fourth element of rows one and three. Array boundaries are checked automatically. Only one and two-dimensional arrays are supported.

3.3.2 Relations

Three different relations can be established between Alego objects: hierarchy, sharing, and grouping. All these relations are dynamic, i.e., they can be removed and re-established between any two objects at any time.

In Alego, the *hierarchy* or parent-child relation is used to combine objects to compound objects. When a parent-child relation is established between a parent P and a child C, several things happen:

1. As a consequence of the dynamic addressing scheme, P and C will from now on be affected by any references to their name within the new scope. For instance, suppose the name of C is "sheep," and P has a program that states: on mouse-click on any child named "sheep" this child should beep. Then, C will behave accordingly from there on. Similarly, C could have a program that refers to its parent. In that case, the parent's behavior would be affected.
2. As a consequence of the different types of methods, some methods of C might now be available for P via delegation. For instance, suppose, C has a method *add* that adds two numbers and returns the result. Then, if P gets a message "add(3,4)" from some other object, it can delegate it to the new child. For the sender of the message, it looks exactly as if P would have computed the result. Note that in Alego, delegation works in the reverse direction than in other systems, e.g., Self. In these systems, child objects delegate messages to their parents.
3. When moving or deleting P, C will be moved or deleted automatically.
4. The visual representation of C will only be visible within the boundaries of P's visual representation.

For as long as the parent-child relation is maintained, these consequences will hold. We refer to the first two consequences as *adaptive inheritance*. Like other dynamic inheritance mechanisms, adaptive inheritance is an ideal vehicle for automating object transformations. For instance, an object can specify that all its child objects have a certain behavior. A new child will automatically acquire this behavior.

Adaptive inheritance also benefits from the flexible *sharing* relation of Alego. In Alego, sharing is done on a per slot basis. That is, two or more objects can share any subset of their properties, methods, or programs. Each object has its own copy of the shared slot. A group of objects that share a particular slot is specified by an address. Therefore, sharing is not associated to object identities but to their names and other characteristics determined by this address. There can be several disjoint groups of objects sharing a slot with the same name, each group with its own value.

The advantages of this form of sharing are that, first, it does not introduce dependencies between objects, and, second, it is more flexible than the traditional sharing by inheritance or delegation. Since each object has its own copy of the shared information, objects can be removed without affecting other objects' shared slots. Since any addressable group of objects can share any combination of individual slots, many more configurations are possible than for sharing by inheritance, in which the child inherits *all* methods and properties of the respective parent. First, not all parent characteristics have to be inherited by a new child, and, second, an object can "inherit" from any other object, not just its parent.

The disadvantages are that, first, it costs extra memory, and, second, the updating process can slow down the system when there are many objects to update. In Alego, we update shared information whenever possible on a per need basis in order to minimize the effects of accumulated updates.

Finally, Alego has a *grouping* relation to form ordered groups (or arrays) of objects. Grouping is used for processing objects as arrays or lists. Any addressable set of objects that have the same parent can form a *group*. A group has its own name, and can be addressed in the same way as individual objects. When building a group, each member receives an index within this group. Objects can form part of more than one group at the same time.

Alego offers many features to handle groups efficiently. First, most object operations such as instantiating, moving, deleting, addressing objects can be performed for groups as well as individual objects. Furthermore, there are operations for, e.g., extending, shrinking, or combining groups, or for laying out the (visual representation of) group members in a specific shape (e.g., a circle or matrix). Finally, the addressing scheme provides features for addressing group members, including one- and two-dimensional array addressing and pointer addressing for list processing.

The grouping feature, together with array addressing, has proved to be very useful when building board games.

3.4 Language

In Alego, all objects are programmed in a simple scripting language, called *Lago*. *Lago* is an interpreted language and can be used for executing commands directly or to define programs that can be triggered by messages or events.

Our main goal in designing this language was to express object behavior in a simple, flexible, fast and concise way. Besides a rich set of flexible, polymorphic operators, *Lago* has very few constructs. It offers conventional *if-then-elseif-else* and a *do-while* control structures, and an *on-do* statement for associating commands to events. Furthermore, *Lago* eliminates explicit declarations, typing, and scoping problems for the programmer, since it does not have variables. Instead of variables, object properties are used to store data.

3.4.1 Operators

Lago provides a rich set of operators. Besides the usual arithmetic, logical, and relational operators, there are several combinatorial operators, and operators for special operations, e.g., swapping or shifting properties over lists of objects, or recursively evaluating expressions.

The introductory example of the *15-Puzzle* demonstrates the extreme conciseness provided by our extended addressing scheme in combination with a powerful operator such as swap (denoted ":"). The simple program for moving the tile would require quite a complicated piece of code in a conventional language such as Pascal, C or Basic. This conciseness due to heavy use of operators characterizes many functional languages, such as APL [Bea95, McI95], FP [Bir88], or ML [Pau91].

In the *Pattern* example we will encounter another interesting operator, the unary operator @, which recursively evaluates an expression until it finds a fixed point, i.e., the expression remains stable.

Operators have the advantage of the inherent elegance and expressiveness of mathematical phrasing. Also, since we are used to associate a much more uniform syntax with mathematical concepts (e.g., binary operators) than with natural language constructs, it can be expected that certain syntax errors (e.g., wrong word ordering) will occur less frequently. Finally, the power of operators can be further increased by composition. It might sometimes take a while to understand how a complex composition operation works, but once this is understood, it offers a concise way to express a precise transformation.

3.4.2 Polymorphism

The semantics of operators may vary depending on the types of the operands and the context. Many operators can be applied to several types. An example of the varying semantics of the plus operator in combination with assignment (+=) is given in Figure 4.

Expression	Meaning
A += 3	Add 3 more elements to the array A (by cloning the first array element)
A actual += 3	Increment the pointer named "actual" pointer (A is a group object) by 3
A += B	Build the union (defined as for set operations) of the arrays A and B, and assign it to A
O text += 3	If the value of the *text* property of object O evaluates to a number, add three to this number and assign the result to O *text*. Otherwise, concatenate O *text* with "3".
O color += red	Sets the bits corresponding to the red component in the *color* attribute of O. For instance, if O's color is blue, after this operation it will be magenta.

Figure 4. Semantics of the += operator

In Alego, types are inferred automatically. Typing is very liberal in *Lago*: Mixed type expressions are tolerated as long as there is a possible interpretation. This liberal, dynamic typing system is not robust. Throughout the Alego environment, we have opted for liberalism at the cost of safety. The reason is that the simple and exploratory programming Alego has been intended for is not error critical. We felt that in this application field, much more damage could be done by overly restrictive and complicated rules than by occasionally surprising results. However, for the future, mechanisms to detect and report possible errors and to increase the overall reliability, would be a welcome improvement.

3.4.3 *Lago* Programs

A *Lago* program is a sequence of (possibly nested) statements. Typically, a program consists of a statement of the form: "*on* EVENT *do* COMMANDs, " where EVENT specifies a mouse or keyboard event, a timeout of a timer object, or another, user-defined event that can be triggered at any time, and COMMANDs consists of a few simple statements.

Lago is a very high-level language and therefore not suitable for time-critical or system programming. The language is still under development and has not been tested with non-programming users. It is expected, that *Lago* will be very easy to learn and use even by inexperienced users. However, some work needs to be done to improve the ease of understanding *Lago* programs written by other users, which sometimes look complex when they contain many unusual operators (a similar problem is known from APL or C code).

3.5 Detailed Example: Pattern

In this section, we present a typical application for Alego. In the following, we demonstrate step by step how a simple game named *Pattern* can be implemented, and later modified.

The main purpose of this demonstration is not to show that Alego supports object-oriented programming but that it supports a straightforward style of developing, using and modifying simple and dynamic applications. Therefore, we focus on simplicity rather than modularity.

Pattern is a simple paper and pencil game for two or more players [Gar70]. Each player has his own $n \times n$ board. There are k different symbols that can be placed on a field. One player, the designer, designs a pattern on his own board using the available symbols. He does not show his board to the other players. The remaining players have to guess the pattern by inductive inference.

In guessing the pattern, each player can either ask the designer for the content of a specific field, or he can tentatively place a symbol into a field. Each player only sees his own board and cannot draw information from the other players' queries. Players need not fill up all field in the board. When the guessing process is finished, the players hand in their boards and their results are evaluated: For each correct guess, the player gets one point. For an incorrect guess he gets *-1*. For a query, he gets *0*. The designer gets twice the difference between the best and the worst player. This encourages the designer to choose the pattern intelligently: It should be regular enough to be guessed reasonably well by at least one player. And it should be complex enough to be missed by at least one player.

3.5.1 Implementing Pattern in Alego

In our implementation, the designer is represented by the computer, and there is only one player (the user) that has to guess the pattern of the board.

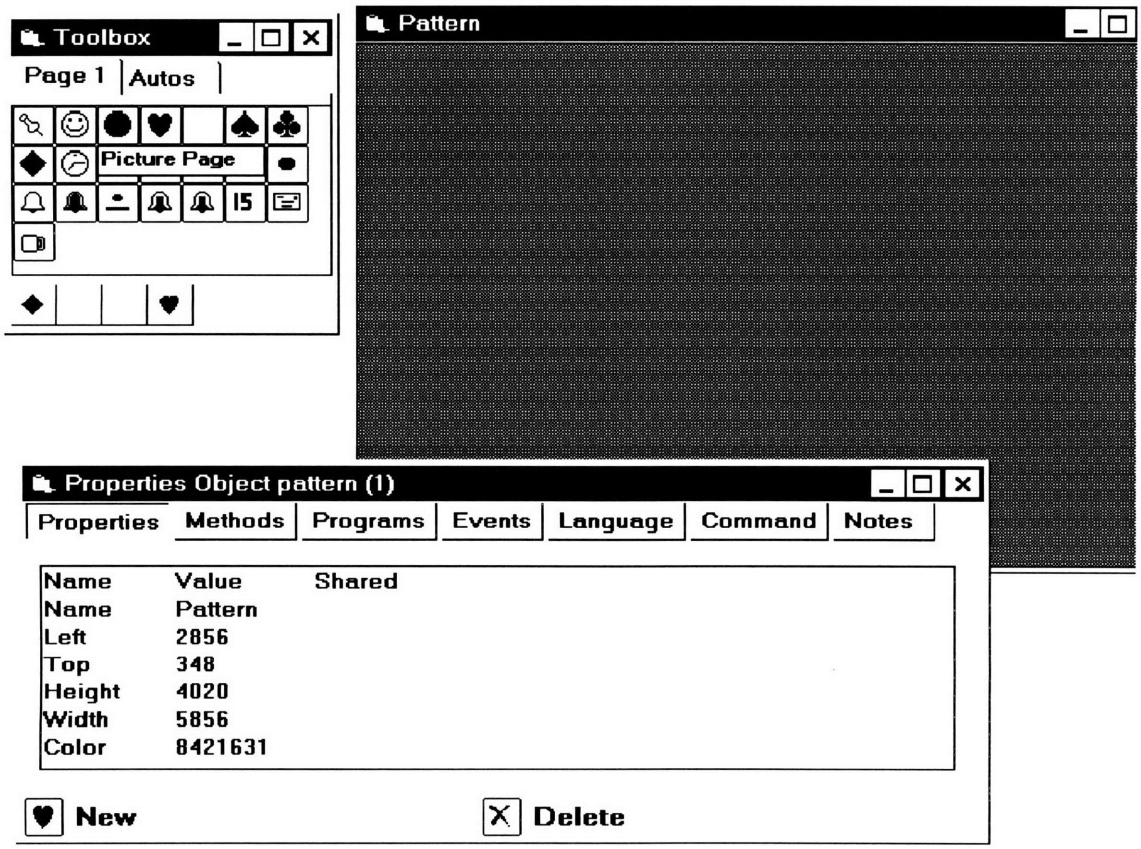

Figure 5. Implementation of the game "Pattern", first step. New Page with its information window.

First, we instantiate a page object by double-clicking on the corresponding prototype in the toolbox. When an object is created, its information window pops up automatically and displays its property page.

Next, we invoke the group instantiation dialog box (not shown) for instantiating a two-dimensional array of "basic game objects" that will represent fields of the board. This is done by holding down a control key while dragging the desired object to the page. The dialog box allows one to choose the size, name and layout of the array. We chose the name *board* for obvious reasons. In the same way, we instantiate another array, *palette*, containing three possible symbols and a button marked with '?' used to query a field.

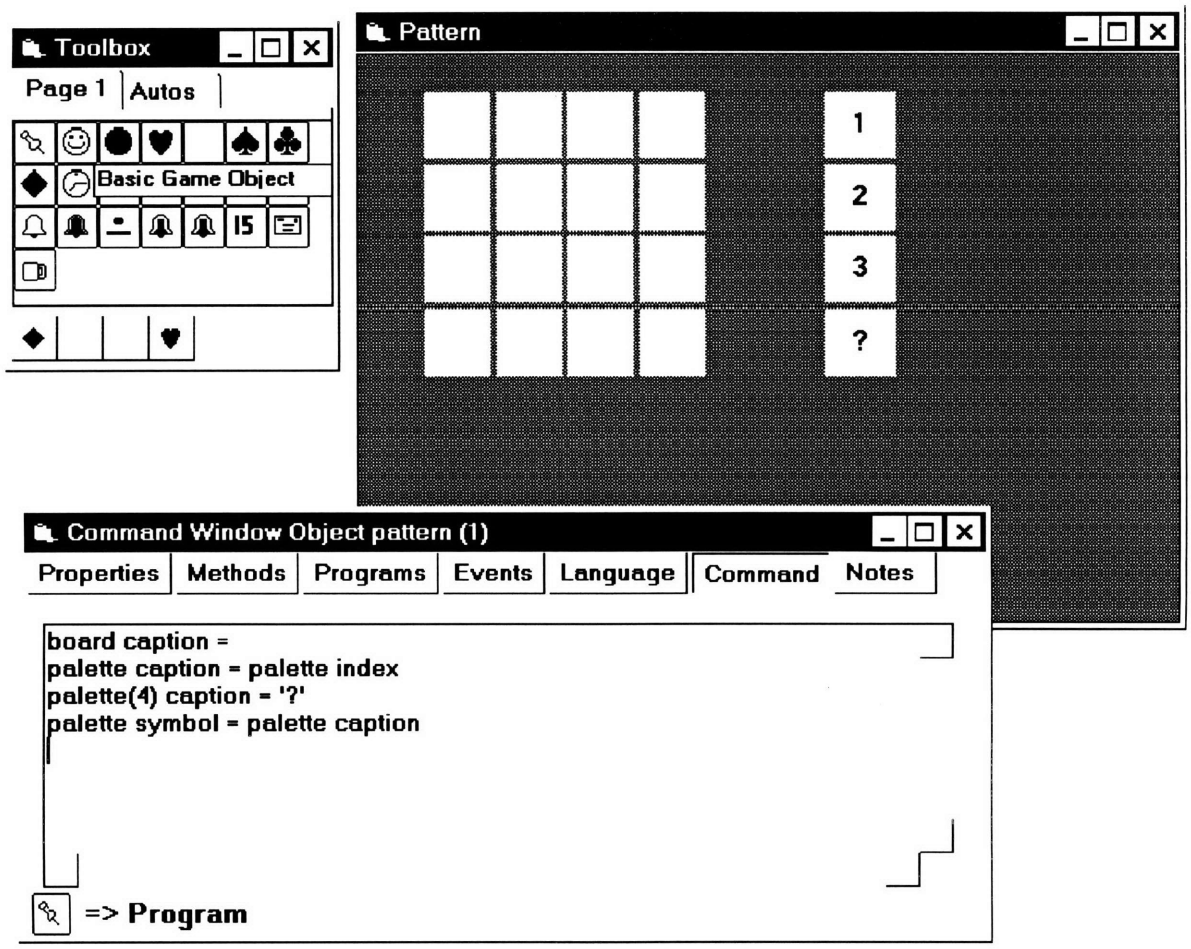

Figure 6. Implementation of the game "Pattern", second step. We have instantiated two group objects, named board ***and*** palette. ***The information window displays the command page of the pattern object, to which we direct several commands that clear the*** board, ***and set the*** caption ***and*** symbol ***properties of*** palette.

The first two commands in the command window (Figure 6) illustrate multi-cast addressing. First, the caption of all subobjects of *pattern* named *board* are set to nothing. Then, each *palette* object's caption is set to its index within the array (the assignment operator "=" works element-wise). We reset the caption of the query button (palette(4)) to '?'. Finally, we store in the *symbol* property of each *palette* object the symbol it represents.

Until now, we have only modified the appearance of objects. All commands we have entered so far were direct commands sent to the *pattern* object, the parent of the two group objects *board* and *palette*. Next, we would like to program the behavior of the game.

The mechanism of guessing the pattern should be as follows: The user clicks on one of the palette buttons to fetch a symbol. Then he clicks on one or more fields in the board to either guess a symbol or, if the symbol he has fetched is '?', to query the specified field. The caption of the corresponding field should be set to the guessed symbol or, on query, to its own symbol.

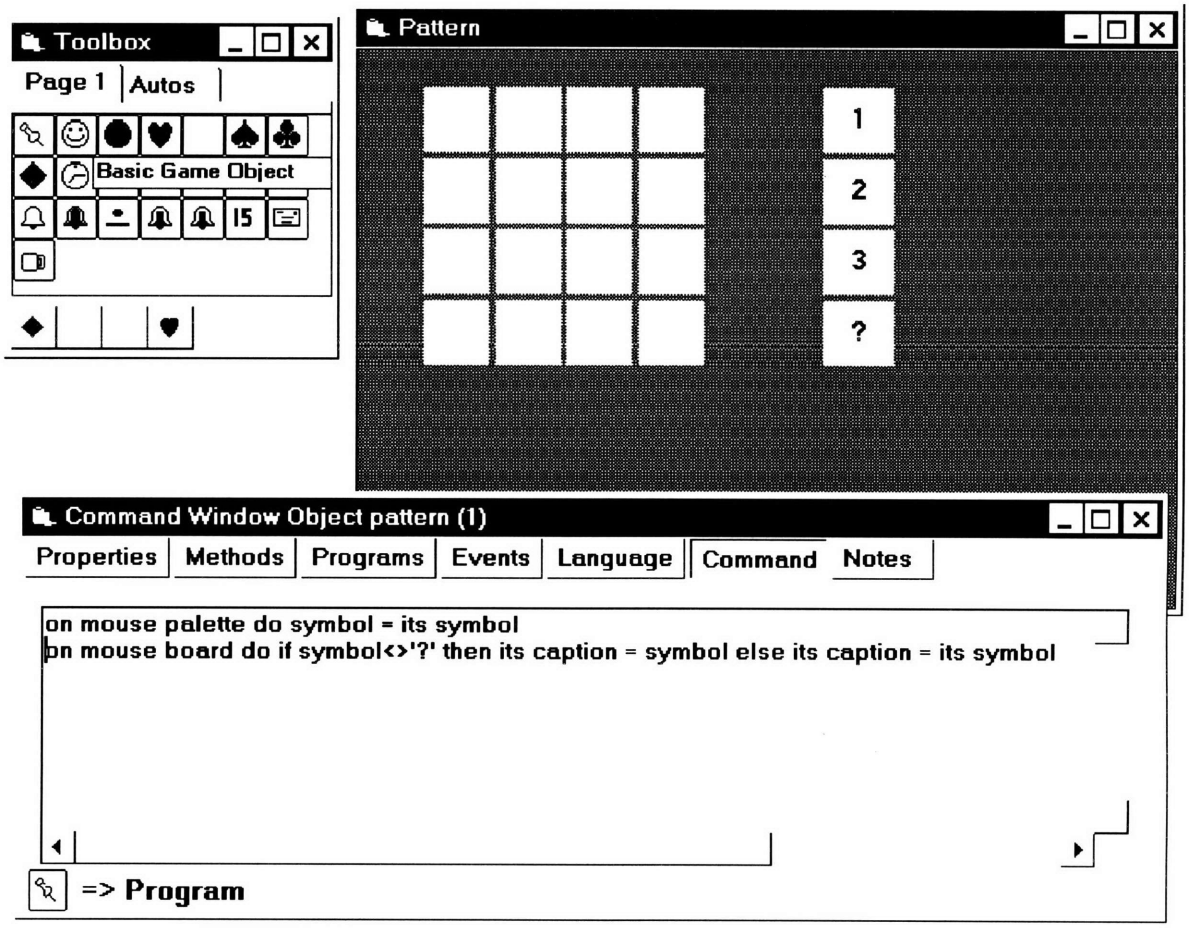

Figure 7. Implementation of the game "Pattern", third step. We have added the two main programs that specify the game mechanism. Clicking on a palette button lets the user fetch a symbol. Clicking on a board field results in a guess or a query depending on symbol that has been fetched.

The two one-liners (see Figure 7) that determine the described behavior are typical Lago programs. Both programs are pretty self-explaining once the addressing mechanism is understood. For instance, the first program "on mouse palette do symbol = its symbol" determines that when the user clicks the mouse (left button is the default) on any *palette* object, its *symbol* is copied to the property *symbol* of the *pattern* object (the program owner). Recall that the addressing keyword "its" refers to the object that has got the event.

In the next program (second line), we assume that the original pattern is stored in the *symbol* property of each *board* object. For now, we haven't set this property. We will now add the necessary functionality for defining a pattern and calculating the score at the end of a game.

In our computer version of pattern, only the player that guesses the pattern is evaluated at the end of a game. For each correct guess he will get *1* point, for an incorrect guess, *-1*, and for a query, *0* points.

In order to compute the score, we have added to the *board* objects a property *score* that holds the individual score for each field. We slightly extend the guessing program: To each field that has been guessed we give a score of *1*, while queried fields get *0*. The most recent action on a particular field overrides previous actions, yielding a correct score value at any time (however, this does not prevent the player from cheating by first querying a field and then "guessing" the correct symbol) .

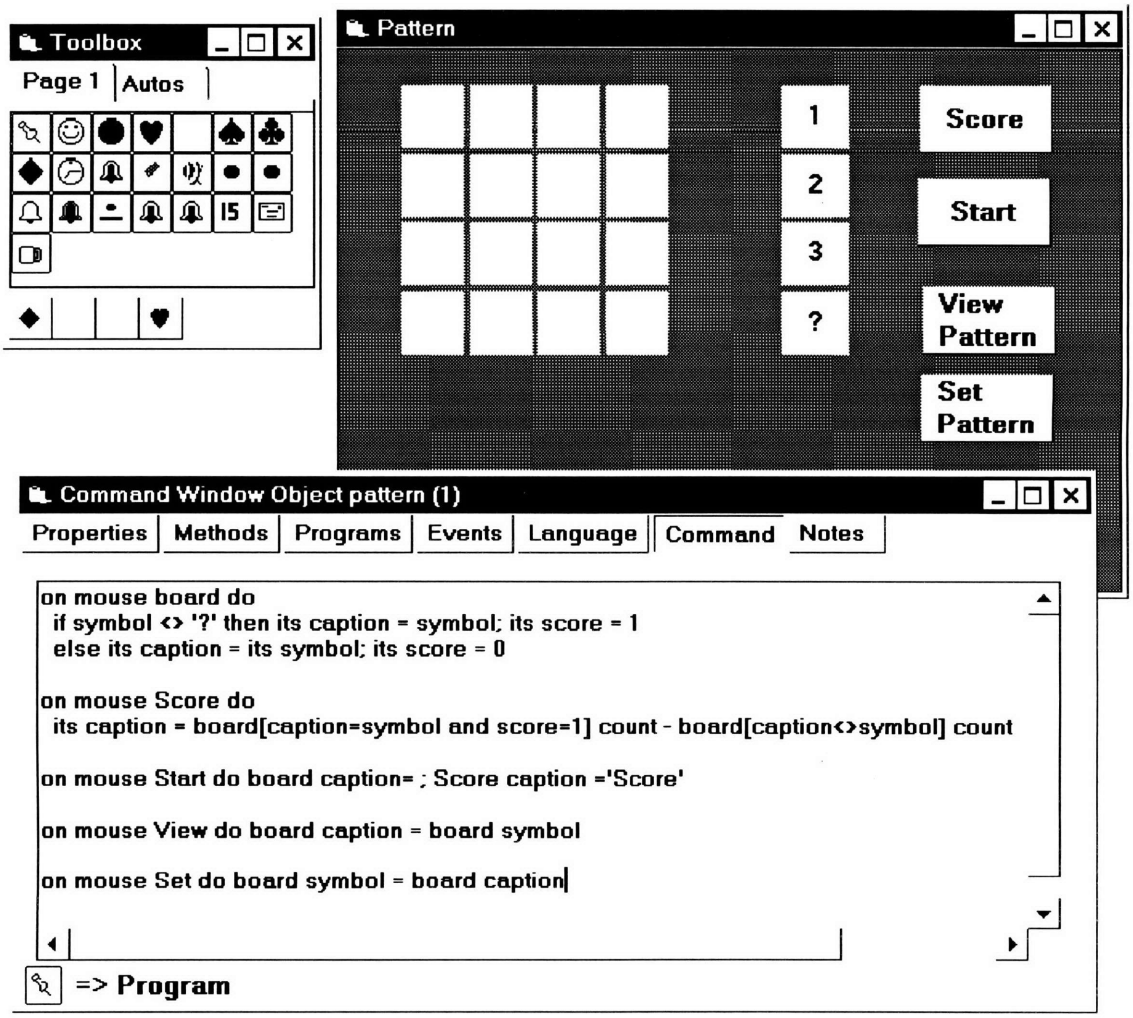

Figure 8. Implementation of the game "Pattern", step four. For computing the score of a game, we have slightly modified the guessing program. We have also added four short programs that determine the behavior of the four control buttons Score, Start, View, and Set.

The program that computes the score (second program in the command window of Figure 8) is executed when users press the button named *score*. Here, we have an example of the usefulness of test conditions to filter out the addresses we are interested in. The simple expression

```
board [caption = symbol and score = 1] count - board [caption <> symbol] count
```

effectively subtracts the number of *board* objects with wrongly guessed symbol (i.e., *caption* <> *symbol*) from the number of correct guesses (i.e., elements with correct caption that have been *guessed*, not *queried*). The resulting number is displayed on the *score* button. The three programs for the *start*, *view*, and *set* buttons clear the board, display the pattern, and store a (new) pattern, respectively.

This completes the implementation of pattern in a single-player version on computer. Six small programs, one for each button, one for the board, and one for the symbol palette are all we need, in addition to setting some properties and visually composing the user interface. The application can be tested at every stage of its construction since running and developing is done in the same mode. Furthermore, the entire game or any of its parts can be saved as prototypes for later reuse. In the next section, we describe some modifications in order to demonstrate the advantages of a flexible object environment.

3.5.2 Variants of Pattern

The user might think of numerous variants of the game. For instance, one might modify the board size or the way scores are computed, or one might want to add a timer that constraints the total time of a game. Such a timer could then be used to reveal random fields successively so that players have to guess fast in order to prevent loosing too many points by fields that are revealed. All those modifications could be incorporated in *Pattern* very easily.

Our first modification changes the symbol palette available to the user. Instead of having just three symbols, we want to take all integers. Since all integers obviously don't fit into the page, we slightly modify the rule for guessing: A symbol 0..9 is concatenated to the actual board caption. A new "blank" symbol erases a field. The question mark is used for guessing, as before.

Modifying the size and layout of an array is easily done by using the group dialog box. The first command, `palette =`, invokes the dialog box for *palette*.

We then adjust the *symbol* and *caption* properties of *palette* to the new layout. Lines two to six in the command window on Figure 9 show the corresponding commands.

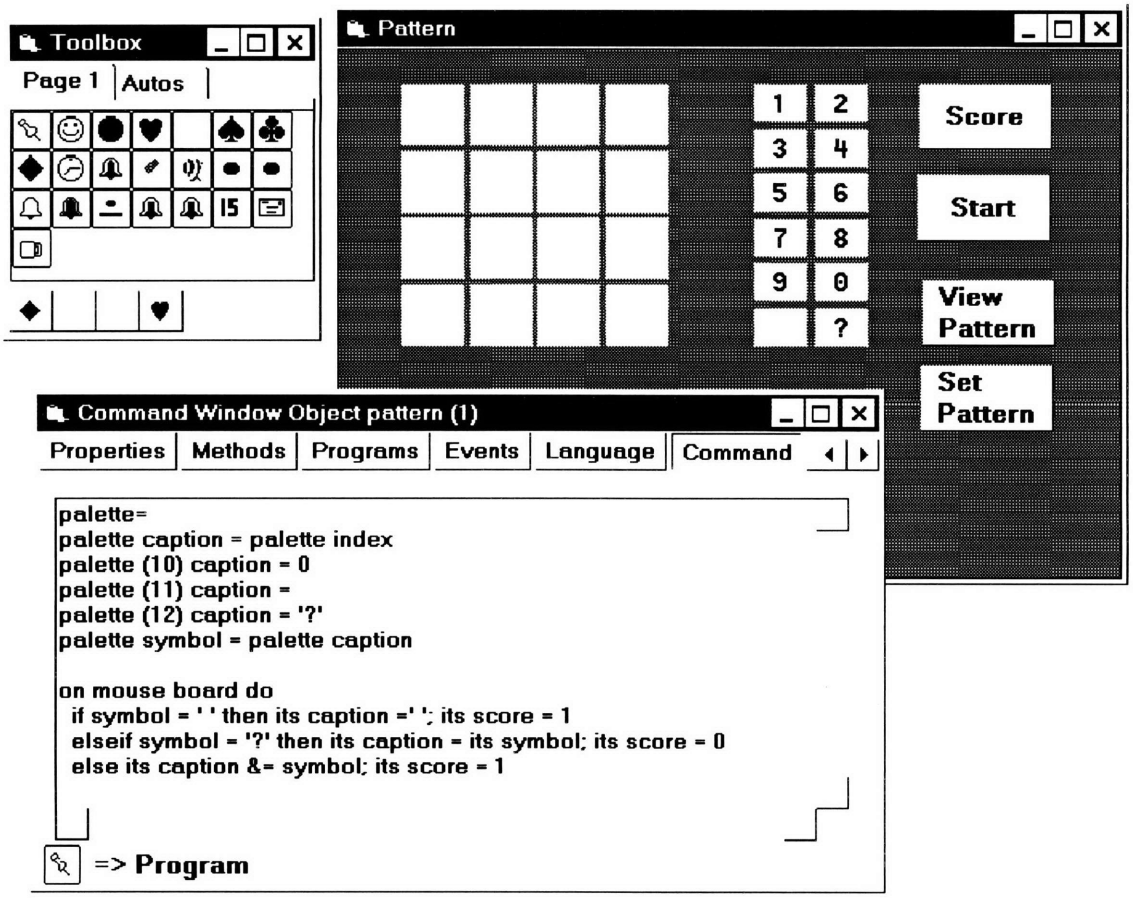

Figure 9. Modifying Pattern. The symbol palette has been extended.

Next, we modify the program for guessing according to the new rule. We add an if clause for the blank symbol, which clears the board field and sets the *score* property to *1* as for a guess. We modify the clause for symbols *0* to *9* so to concatenate the new symbol to the caption (using the operator &=) instead of just assigning it.

That's it. The programming effort for this modification has been minimal. However, finding the right place to change the behavior - in this case, the guessing program - might be a challenge for a different user from the one that has programmed the application. For the moment, Alego does not offer assistance in analyzing an application. Providing such orientation facilities is probably one of the most important tasks for the future development of Alego.

As a further extension that makes the game more interesting we propose to add a mechanism through which the user can guess the pattern in terms of a general mathematical formula that applies to the board elements, for instance, as a function of their index.

For that, we drag an editing object *textbox* to the page. As the user enters an expression and clicks on *textbox*, the board should be filled out according to that expression. For instance, the expression "board index" would just assign each field its own index as caption.

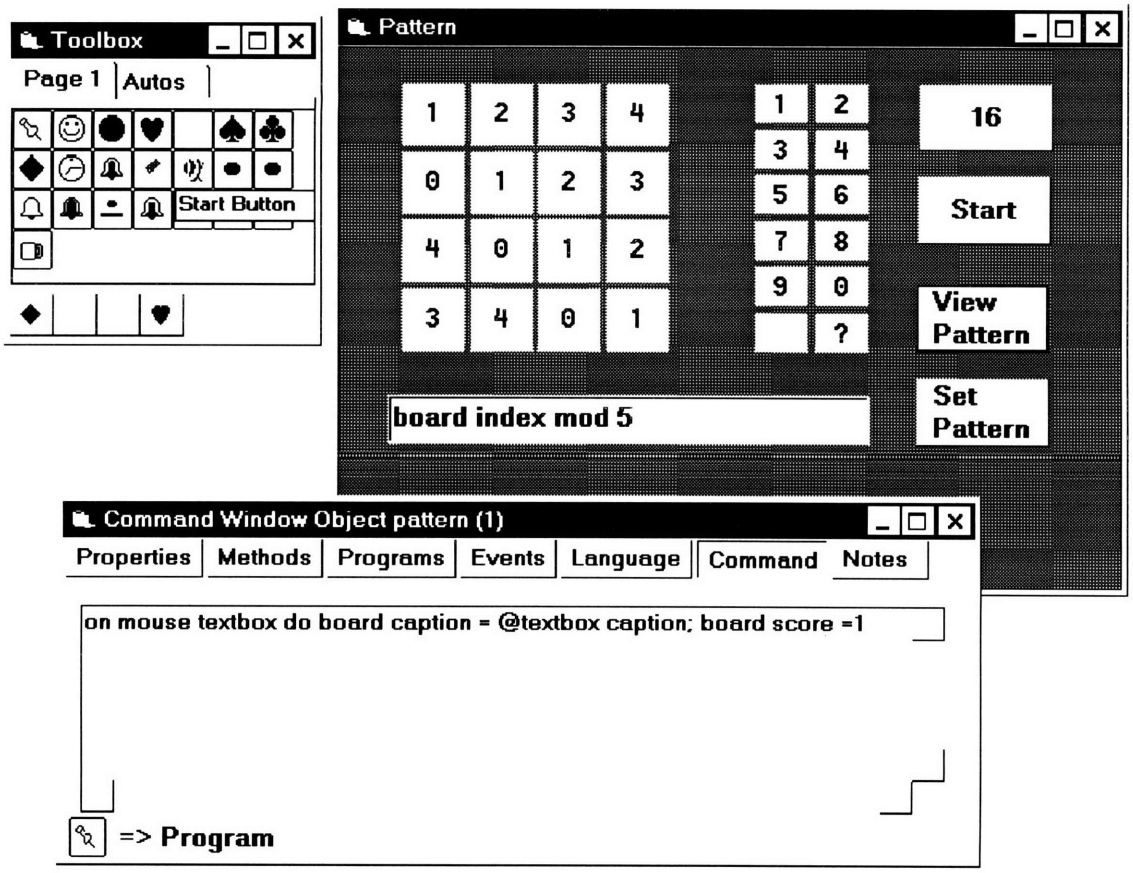

Figure 10. Modifying Pattern. We have added an object textbox that lets the user guess the pattern by giving a general mathematical expression for board fields.

The program in the command window (Figure 10) defines the new functionality. We have already mentioned the recursive evaluation operator @. It takes an expression and returns the result of evaluating the expression repeatedly until a fixed point is reached, i.e., the result equals the input expression. This is how the expression "@textbox caption" is evaluated: Evaluation of "textbox caption" yields the new expression "board index mod 5." This expression is evaluated again, yielding the list of indices of the board elements, each index modulo 5, i.e., "1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1." This result is a fixed point: Evaluating it will result in the same list of integers. Hence, the result of "@textbox caption" is the list of board indices taken modulo *5*. The multi-cast assignment "board caption = @text caption" assigns these integers one by one to the *caption* property of the board elements. The difference between the @ operator and a simple "eval" operator as can be found in many languages is that @ continues evaluating the resulting expression as long as the result differs from the input expression. For instance, the expression "eval(textbox caption)" would have returned the string "board index mod 5." Operators like @ or "eval" add flexibility to the language by enabling users to execute "meta"-commands.

4. Evaluation

In this chapter, we evaluate the concept of flexible components, and we reflect on what has been achieved so far and what remains to be done.

4.1 Advantages of Flexible Components

Flexible components combine advantages of highly dynamic object environment with those of component software. Because of their flexibility, they can be reused in many different contexts, and they can be modified and extended more easily than existing component software. Because of their autonomy, they can interoperate with independently developed other components, facilitating incremental and distributed software development. And, since components are compiled binary objects they can be pre-tested and tuned to high performance.

With these characteristics, flexible components promote a shift from language-centric to the more comfortable and direct component-centric software development.

4.1.1 Problems of Language-Centric Development

Depending on purpose and users, the best way to program an object can vary widely. Traditional development environments do not take this into account because within them all objects are programmed using a single environment-specific language. In order to cover a broad range of applications, this language is usually a general-purpose programming language. In order to support object development in a natural way, it is usually object-oriented. These two characteristics make this language fairly complex and inappropriate for the non-programmer.

Furthermore, integrated objects, e.g., OLE Controls, usually have to be separated from their application environment in order to be modified. This hinders simultaneous testing within the target environment.

4.1.2 Advantages of Component-Centric Development

Flexible components promote a shift to component-centric development, where each component can be further developed on the spot according to its purpose and needs, using a language tuned to the individual component's characteristics. Flexible components support this approach by incorporating their programming interface.

With component-centric development we achieve both simplicity, because programming interfaces are application specific, and universality, because a multitude of languages and interfaces can coexist within a single application environment.

Furthermore, because the development process is incremental, end-users do not need to bother with the complex basic functionality of a component. They can concentrate on the component-specific behavior and modify it through a simple interface. For more complex functionality, available pre-tested modules can be attached to the component. By using a custom-tailored interface, end-users can achieve sophisticated results that until now have been reserved for professionals with elaborate programming skills.

Alego presents some of these advantages. Alego objects are specialized for building simple interactive applications. Some of these objects have quite a complex functionality that an average end-user would not be able to implement. However, their programming interface is very simple and gives users exactly what they need to build the applications they are intended for.

4.2 Problems and Limitations of Flexible Components

One problem common to component software approaches in general is standardization. The success of component software depends on how well a standard can be established. Without a standard, the distributed effort of many developers cannot be combined. To establish a standard is hard and depends on many factors other than technology. Furthermore, standards are inflexible and often stifle innovation.

Other problems and critical aspects are introduced by flexible components in particular:

Complexity. Flexibility often increases the complexity of a component. The more flexible a component is, the more options it presents to the user. If a component as complex as a full-featured word processor presents too much flexibility, the user will be unable to keep track of its features and functionality. On the other hand, developers might have a hard time deciding on how flexible to make the interface or whether to provide several interfaces. More options mean more decisions have to be made.

Performance. The addition of several levels of indirection for the interpretation of methods and messages will certainly decrease performance. In general, message passing might present a bottleneck because of the dynamic and centralized naming system. Merging components instead of passing messages between them might reduce the overhead. However, efficient yet uncomplicated merging operations remain to be defined.

Reliability. Documentation of components is optional and cannot be enforced. Components with multiple interfaces that can be modified in multiple ways by multiple users clearly create multiple sources of problems that have yet to be dealt with. How to prevent users from damaging a component? How to preserve the reliability of applications when single components are exchanged or updated? How to inform the user about changes induced by modifying or exchanging a component of an application?

4.3 Experience with Alego

The strength of Alego lies in its flexible and autonomous object architecture that can be easily adapted for implementing flexible components. In particular, the comprehensive object presentation, the simple language, the powerful naming scheme, and the grouping relation are notable features that have not been found in similar form in other systems. Within its range of application, Alego represents an original and useful tool. In particular, our experience with Alego has confirmed the usefulness of flexible components for creating and modifying a class of simple but highly flexible game-like applications.

On the other hand, Alego represents an extreme simplification of a flexible component environment. It does not offer means for exploring the universality of flexible components. In

particular, we haven't had the opportunity to experiment with component interoperability, or with multiple interfaces and diverse application areas.

Our experience so far is limited to very small applications. As applications grow larger and more complex, with more and more interoperating objects, Alego's performance deteriorates rapidly. In fact, poor performance and limited robustness are major limitations of Alego in its current stage. Both problems are partly rooted in the fast implementation. Apart from this, we are able to evaluate Alego in terms of how far it has helped in achieving the intended goals concerning user-friendliness.

4.3.1 Is Alego Simple?

Simple to Learn? Not yet. The few experience we have had on this point show that, in fact, Alego is not as simple to understand for newcomers as we had expected. Mixing development and run time, we have provided several unusual mouse and keyboard combinations in order to let the user move and resize, activate, and get information on an object at the same time. For people used to the Microsoft Windows environment, some of these combinations were difficult to remember.

Another confusion has been caused by the lack of browsing and cross referencing tools that explain how objects are related with each other. The problem of code scattered in little pieces over many objects, which makes it hard to find out what's going on in the system, is common to object-oriented or message-based systems in general.

Although Alego has not proved simple enough to be understood immediately by novices, we are optimistic about its general design. We attribute many problems to the fact that until now very little effort has been put into making the start easy, e.g., by providing help features.

Simple to Use? Yes, in its domain. Basic object manipulation is simple: Creating, destroying, activating, moving, replicating objects, and even arrays of objects can be done by point-and-click. Programming objects is simple: Sending commands to any object or programming objects can be done by using the object's command interface that pops up automatically together with complete information of the object's interface plus documentation. Addressing objects or groups of objects is simple and efficient. All together, building applications such as the ones we have presented in the example section can be done in a few minutes.

Changing applications is usually simple. However, it can be hard to find out what has to be changed or where behavior has been defined once applications grow larger and more complex, since there are no tools to examine object interoperation. Furthermore, in its present state, Alego does not offer any mechanisms to undo changes.

4.3.2 Is Alego Flexible?

Nearly any aspect of an object or an application can be modified. Users can add and remove properties and methods, they can program the behavior of objects, and relate it to a variety of events. Alego objects and applications are as flexible as in other dynamic single-mode object environments such as Self.

However, the way objects can be modified is uniform throughout the system and cannot be changed. Alego does not support modification of the flexibility of objects, client specific

restrictions, multiple or hierarchical interfaces, or even different programming languages. In those aspects, it is by far not as flexible as flexible component software is meant to be.

4.3.3 Is Alego Effective?

Yes and No. Alego is very effective in terms of how fast an experienced user can develop an application. Simple games such as Xattack, Patterns, or Magic Squares can be developed from scratch (i.e., using the basic toolbox objects) in a few minutes. We haven't found another system comparable to Alego in this respect.

We attribute the effectiveness of Alego to several reasons. First, by merging runtime and development time, Alego offers an interactive programming style that allows users to make changes at any time, with immediate feedback. Second, since objects are represented visually and can be manipulated directly, the user can see the results of even small modifications, and there is less abstraction involved in the programming process. Third, programming in Alego is straightforward because the of the simplicity of the language. The user is not distracted by details of syntax or semantics, and can instead concentrate on the problem of interest. Finally, the Alego toolbox provides a collection of specialized components that constitute ideal basic building blocks for the class of applications Alego is intended for. This collection can be extended by the user, who can produce new prototypes from components at any stage during development.

As a result of its flexibility and effectiveness, Alego makes both playing and programming a game more attractive. Many simple games become boring after a certain time. With Alego, users can make them more challenging as their skills improve, and can maintain the excitement by introducing new variants. Programming is more fun because there is no need for long preparation or planning before users can start an application, and users can try out alternatives immediately while programming.

However, Alego is not as effective for analyzing or debugging applications. It does not offer tools to locate pieces of information or trace the control flow of object interaction.

In summary, the overall interaction style of Alego seems to be very promising but many features and details need to be added before testing Alego on a broader public. Only then, we will be able to definitely judge whether Alego meets our expectations.

4.4 Future Work

For a successful implementation of flexible component software, several aspects need to be further investigated:

- **Performance and Scalability**. Achieving an acceptable performance will be much more difficult for flexible components than for conventional component software. Dynamic modifiability of components is usually costly because much code is interpreted. Also, the flexible component engine represents a potential bottleneck by offering extended services such as address translation. Furthermore, considerable communication overhead is involved when compound components are implemented as clusters of independent, interoperating components (the usual implementation in existing component software approaches). To reduce the overhead for intercommunication and message interpretation within component

groups, composition techniques should be developed to consolidate the structure of such groups. [Ste87] discusses the problem of gradual consolidation of growing objects to more static structures from a theoretical point of view.

- **Robustness and Reliability**. A flexible component environment is prone to errors and inconsistencies that arise from incoherent manipulation of components. Since it is so easy to modify components, it is also easier to introduce unintended behavior and inconsistencies. Furthermore, the distributed system architecture makes it much harder to check inconsistencies and to test applications. Much of the functionality that is traditionally provided by integrated development environments has to be implemented in a distributed fashion.
- **User Orientation**. One of the most important aspects of any system targeted to end-users is to facilitate efficient information. When heavy reuse of other peoples products is desired, this becomes even more critical. For one, users have to be able to find out what is available and how to use it. Second, they need to rapidly orient themselves in programs that involve many objects and complex interdependencies. Third, they often need to browse through not only space but also time, trace applications, or observe actions and state of many objects at the same time. Information has to be revealed and highlighted but also hidden at the right moment. Interesting research is going on to study the optimal use of parallelism in user interfaces [Van96].
- **Automation**. Many operations and actions in a distributed system with autonomous, interoperating objects are amenable to automation:
  1. Help and Documentation. Assistance could be automated by generating useful comments whenever the system can detect that the user needs help. Another field of investigation is the partial automation of documentation updates when an object is modified.
  2. Search and Classification. Tools that search for objects or functionality that would be useful within a specific context, or that classify objects automatically according to their purpose and functionality would be useful.
  3. Object Composition. When combining or composing objects, new relations could be established automatically and properties or methods that are induced by the nature of the composition operation could be generated automatically.

5. Conclusion

In this thesis, we discussed the problem of software development in terms of satisfaction of the end-user. In particular, we observed that the traditional strict separation of application development, done by professionals, and the use of applications by end-users does not support new requirements of computing in homes and schools. In these contexts, end-users should have more liberty in modifying and combining applications, and they should be able to incorporate available functionality efficiently in self-made applications.

We presented flexible component software as a promising solution to satisfy the new requirements. Flexible component software generalizes existing component software approaches by allowing a component to provide its own programming interface. The flexible component architecture promotes reuse of available functionality. Furthermore, flexible components can be combined and composed efficiently using diverse relations and operations.

With these characteristics, flexible components support a new, component-centered style of software development. In a sense, component-centered software development generalizes the compound document technology. As compound documents unify the editing of a diversity of formats, flexible components overcome programming language boundaries. Component-centric development unifies different languages and programming interfaces within a single platform of interoperable components. Similar to the compound document's in-place editing of embedded components, flexible components support in-place programming of components integrated in an application, using each component's individual programming interface.

Component-centered software development offers a comfortable way to modify components within their application environment, at runtime of the application. It also gives the average end-user of a component additional power because the component's programming interface will, as a rule, be much simpler than its original source language.

Experiments with Alego have demonstrated the usefulness of flexible components for small, exploratory applications, such as the creation of modifiable board games. Moreover, we have seen that this simplicity can be achieved without sacrificing the universality of component software. On the contrary, flexible components even add to the generality of component software by increasing the diversity of components.

Our main conclusion is that application-specific diversification of components in terms of their flexibility and their language interface is highly desirable. Complex, highly specialized components primarily need to be reliable and easy to understand, but they do not need to be very flexible. For small, experimental applications in which components are typically small, and users like to combine them in many different ways, flexibility and programmability are important.

5.1 Outlook

The concept of flexible components is a powerful one: It combines divergent approaches and antagonistic advantages. It unifies flexibility and stability, diversity and compatibility, simplicity and power. Accordingly, we have to deal with the combined problems that come with these

advantages. Furthermore, many tradeoff decisions have to be made for combining advantages conveniently.

Flexibility has its costs and should therefore by used carefully. It is a difficult task for the initial developer of a flexible component to determine how flexible the component should be, which aspects are likely to be modified over time, and what should be fixed in order to achieve acceptable performance and efficiency. The problem how this decision process can be assisted has to be investigated.

Next, we should experiment with flexible component with multiple interfaces, multiple languages, and multiple users. We should investigate the interoperability of components with divergent characteristics. One interesting research direction is the gradual consolidation of components. Another challenging problem is improving the reliability and robustness of flexible components and ensuring their consistency. Furthermore, flexible component software has great potential for automating some activities, which could significantly improve its user-friendliness. Also, the area of error handling, feedback, and the treatment of emergency situations has to be investigated. Finally, we have to deal with performance and scalability issues.

Flexible component software is only at its beginning. It promises much, but many details have yet to be figured out. We have left much of the work needed for an efficient implementation of flexible components for future research. What we do hope to provide with this work is a critical overview of the main problems of component software and traditional software development approaches with respect to the end-user. More importantly, we hope to convey a vision of a more flexible, universal, and creative approach to software development, in which users of different background will be able to contribute their unique talents and expertise.

Literature

[Adl95] Adler, R. M., "Emerging Standards for Component Software", IEEE Computer, March 1995.

[Age94] Agesen, O. and Ungar, D., "Sifting Out the Gold", Proceedings OOPSLA '94, October 94.

[And95] Andersen Consulting, "Smalltalk: A Safe Choice for Building Business Systems", White Paper, 1995.

[App95] Apple Computer Inc., "OpenDoc: Required Reading Packet", Preliminary Draft, February 1995.

[App95b] Apple Computer Inc., "Dylan Reference Manual", Draft, September 1995.

[Bae94] Baer, A. J., "Smalltalk", discussion panel, Proceedings OOPSLA '94, October 94.

[Bea95] Beaumont, N., "Is APL2 a Good Programming Language?", Proceedings of the APL '95 Conference, San Antonio, Texas, p1-10, 1995.

[Bir88] Bird, R. and Wadler, P., "Introduction to Functional Programming", Prentice Hall, 1988.

[Bor94] Borenstein, N. S., "E-Mail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail", submitted to ULPAA '94.

[Box95] Box, D., "Building C++ Components Using OLE2", C++ Report, March-April 1995.

[Bro94] Brockschmidt, K., "Introducing OLE 2.0, Part 1: Windows Objects and the Component Object Model", Microsoft Systems Journal, August 1994.

[Bro95] Brockschmidt, K., "Inside OLE", Microsoft Press, Redmont, Washington, 1995.

[Car94] Cardelli, L., "Obliq: A Language with Distributed Scope", Technical Report, 1994.

[Cha91a] Chambers, C. and Ungar, D. and Bay-Wei, C. and Hoelzle, U., "Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF", LISP and Symbolic Computation: An International Journal, March 91.

[Cha91b] Chambers, C. and Ungar, D., "Making Pure Object-Oriented Languages Practical", Proceedings OOPSLA '91, October 91.

[Cha93] Chambers, C., "The Cecil Language: Specification and Rationale", Technical Report 93-03-05, Univeristy of Washington, Seattle, March 93.

[Cha95] Chambers, C., "Overview of the Cecil/Vortex Project", May 95.

[CIL94a] CI Labs Meta Group, "Component Software", White Paper, December 1994.

[CIL94b] CI Labs, "OpenDoc: Making Software Work Together", Background Information, August 1994.

[Com96] Comaford, C., "The Brave New World of Java", PCWeek Online, January 1996.

[Cot95] Cote, R. C., "Good News from Delphi", BYTE, May 95

[DeT93] DeTreville, J., "The Graph VBT Interface for Programming Algorithm Animations", In Proc. 1993 IEEE Symposium on Visual Languages, p26-31, August 93.

[Dec95] December, J., "Presenting Java: An Introduction to Java and HotJava", Sams.net Publishing, 1995.

[Dob95] Dobson, R., "RADical Databases", BYTE, July 95.

[Don92] Dony, C. and Malenfant, J. and Cointe, P., "Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation", Proc. OOPSLA '92, Sigplan Not. 27/10, 201-217, October 92.

[DSo94] D'Souza, D., "Training Professionals in object technology", discussion panel, Proceedings OOPSLA '94, October 94.

[Dum95] Dumas, J. and Parsons, P., "Discovering the Way Programmers Think About a New Development Environment", Communications of the ACM, Vol.38 #6, June 1995.

[Fug92] Fugini, M., Nierstrasz, O., and Pernici, B., ``Application Development Through Reuse: The ITHACA Tools Environment," SIGOIS Bulletin, vol. 13, no. 2, Aug. 1992, pp. 38-47.

[Gar70] Gardner, M., "Mathematical Circus", Penguin Books: New York, 1970.

[Har93] Harrison, W. and Ossher, H., "Subject-oriented Programming: A Critique of Pure Objects", Proc. OOPSLA '93, pages 411-428, 1993.

[Hoe93] Hoelzle, U., "Integrating Independently-Developed Components in Object-Oriented Languages", Proceedings ECOOP '93.

[Hoe94] Hoelzle, U., and Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", Proceedings of the ACM SIGPLAN `94 Conference on Programming Language Design and Implementation, Orlando, FL, June 1994.

[Hoe95] Hoelzle, U., "Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming", Ph.D. thesis, Computer Science Department, Stanford University. Published as Stanford CSD Technical Report STAN-CS-TR-94-1520, 1995.

[IBM94] IBM Corp. Objects Technology Products Group, "The System Object Model (SOM) and the Component Object Model (COM): A Comparison of Technologies from a Developers Perspective", White Paper, Mat 1994.

[Ind95] Indermaur, K., "Baby Steps", BYTE, March 1995.

[Joh95] Johnson, R., "Smalltalk Summer School", University of Illinois at Urbana-Champaign, http://st-www.cs.uiuc.edu/users/johnson/summerschool.html, 95.

[Jon95] Jones, C., "Patterns of Large Software Systems: Failure and Success", IEEE Computer, March 95.

[Kho86] Khoshafian, S. N. and Copeland, G. P., "Object", Proc. OOPSLA '86, ACM, pp406-416, September 1986.

[Kho95] Khor, K., and Lovett, S. M., "IBM Smalltalk Tutorial", North Carolina University, http://www2.ncsu.edu/eos/info/ece480_info/project/spring95/proj 39/www/index.html, 1995.

[Lea94] Leake, G., "The Benefits of Component Software: How OLE Can Help Move Software from the Object-Oriented Age to the Component Era", Microsoft Developer Network News, November 1994.

[Lie86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", Proc. of OOPSLA '86, Sigplan Not. 21/11, 214-223, November 1986.

[Lin95a] Linthicum, D. S., "The End of Programming", BYTE, August 95.

[Lin95b] Linthicum, D. S., "Foxy Move to Client/Server", BYTE, August 95.

[Ste87] Stein, L. A., "Delegation Is Inheritance", Proc. OOPSLA '87, ACM, pp138-146, October 1987.

[Mad93] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K., "Object-Oriented Programming in the BETA Programming Language", Addison-Wesley and ACM Press, 1993.

[Mad94] Madsen, O. L., "The Mjølner BETA System: BETA Language Introduction", Mjølner Informatics Report, MIA 94-26(1.0), http://www.daimi.aau.dk/~beta/Tutorials /BETAintroduction/BETAintroduction.html, August 1994.

[McI95] McIntyre, D. B., "The Role of Composition in Computer Programming", Proceedings of the APL '95 Conference, San Antonio, Texas, p116-133, 1995.

[Mey92] Meyer, B., "Eiffel: The Language", Prentice Hall, 1992.

[Mic94] Microsoft Co., "The Component Object Model: Technical Overview", White Paper, October 1994.

[Mic95] Microsoft Co., "OLE Control and Control Container Guidelines, Version 1.1", http://www.microsoft.com/msdn/library/technote/ole11.htm, November 95.

[Muk96] Mukhi, V., "The Makings of an OCX Container", http://www.neca.com/~vmis/final1.html, 1996.

[Mye95] Myers, W., "Taligent's CommonPoint: The Promise of Objects", IEEE Computer Journal, March 1995.

[Orf95] Orfali, R. and Harkey, D., "Building a SOM OpenDoc Part", Reprint from Dr. Dobb's Journal, March 1995.

[Oss95] Ossher, H., Kaplan, M., Harrison, W., Katz, A., and Kruskal, V., "Subject-oriented Composition Rules", Proc. OOPSLA '95, pp235-250, ACM, Austin, Texas, October 95.

[Ost90] Ousterhout, J. K., "Tcl: An Embeddable Command Language", Winter USENIX Conference Proceedings, 1990.

[Pae93] Paepke, A. (Editor), "Object-Oriented Programming: The CLOS Perspective", MIT Press, 1993.

[Pau91] Paulson, L. C., "ML for the working programmer", Cambridge University Press, 1991.

[Pie94] Piersol, K., "A Close-Up of OpenDoc", reprint from BYTE, March 1994.

[Pou85] Poundstone, W., "The Recursive Universe", William Morrow and Co. Inc., New York, 1985.

[Raw95] Rawles , R., "Object-oriented languages are getting more dynamic", The Newsweekly for MacIntosh Managers, Volume 9 Number 11, 1995.

[Rob94] Robertson, S. P., Carroll, J. M., Mack, R. L., Rosson, M. B., Alpert, S. R., and Koenemann-Belliveau, J., "ODE: A Sefl-Guided, Scenario-Based Learning Environment for Object-Oriented Design Principles", Proc. OOPSLA '94, ACM, Portland, Oregon, 1994.

[Smi95a] Smith, R. B., Maloney, J., and Ungar, D., "The Self 4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility", Proc. OOPSLA '95, ACM, Austin, Texas, 1995.

[Smi95] Smith, R. B., and Ungar, D., "Programming as an Experience: The Inspiration for Self", ECOOP '95 Conference Proceedings, Aarhus, Denmark, August 1995.

[Sta95] Stata, R., and Guttag, J. V., "Modular Reasoning in the Presence of Subclassing", OOPSLA '95 Conference Proceedings, pp200-214, ACM, Austin, Texas, October 1995.

[Ste90] Steele, G. L., "Common Lisp the Language", http://fas.sfu.ca/projects/ ElectronicLibrary/Collections/CMPT/csbooks/cltl.html, 2d. edition, Thinking Machines Inc., 1990.

[Sun95] Sun Labs, "The Java Language: A White Paper", http://java.sun.com/ 1.0alpha3/doc/ overview/java/index.html, 1995.

[Sut95] Sutherland, J., "Smalltalk, C++, and OO COBOL: The Good, the Bad, and the Ugly", Object Magazine, http://www.tiac.net/users/jsuth/papers/ oocobol.html, 1995.

[Ung91] Ungar, D. and Smith, R., "Self: The Power of Simplicity", Lisp and Symbolic Computation, Vol. 4, 1991.

[Ung92] Ungar, D., Smith, R., Chambers, C., and Hoelzle, U., "Object, Message, and Performance: How They Coexist in Self", Computer, 25(10), October, pp. 53-64, 1992.

[Van96] Vandevoorde, M., "Parallel User Interfaces for Parallel Applications", Submitted to HPDC '96.

[Vio94] Vion-Dury, J. and Santana, M., "Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems", Proc. OOPSLA '94, ACM, Portland, Oregon, 1994.

[Wat95] Watters, A. R., "Python Online Tutorial", The McGraw-Hill Companies, Inc., http:// www.wcmh.com/uworld/archives/95/tutorial/005.html, 1995.

[Way95] Wayner, P., "Free Agents", BYTE, March 1995.

[Weg87] Wegner, P., "Dimensions of Object Oriented Language Design", Proc. OOPSLA '87, Sigplan Not. 22/12, 168-182, December 87.