UC Irvine
ICS Technical Reports

Title
The programming language Lagoona : a fresh look at object-orientation

Permalink
https://escholarship.org/uc/item/3pw2p1wq

Author
Franz, Michael

Publication Date
1996-09-11

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SL BAR
Z
699
C3
no. 96-40

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

The Programming Language Lag*oo*na: A Fresh Look at Object-Orientation

Michael Franz

Technical Report 96-40
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

11th September 1996

The Programming Language Lag*oo*na: A Fresh Look at Object-Orientation

Michael Franz
Department of Information and Computer Science
University of California
Irvine, CA 92697-3425

Abstract

Lagoona is a strongly-typed object-oriented programming language based on Oberon. Lacking the *class* construct found in traditional object-oriented languages, Lagoona separates many of the concerns usually subsumed by classes, such as *encapsulation*, *structure*, and *behavior*, turning them into independent language constructs. A rich data model is provided that can express the difference between *specialization* and *combination*, two relations that are otherwise often both mapped onto subtyping.

In contrast to most strongly-typed object-oriented languages, Lagoona's *messages* are not subordinate to classes, but are stand-alone entities that can be combined with arbitrary objects. A delegation mechanism is provided, by which objects can forward received messages even if they do not "understand" their contents. Lagoona's particular message-send semantics make the construction of extensible systems simple and elegant.

Keywords: programming languages, extensible programming, object-orientation, specialization vs. combination, stand-alone messages, Lagoona, Oberon

1 Introduction

The programming language Lagoona is a general-purpose programming language with capabilities for extensible object-oriented programming. It is a direct descendant of the language *Oberon* [Wir88a] and continues a family line that has led from *Algol* [Nau60] to *Pascal* [Wir71b], onwards to *Modula-2* [Wir82] and then to *Oberon*. At first sight, Lagoona doesn't differ much from Oberon, just as Oberon didn't appear to differ much from Modula-2 when it first arrived. Beneath the surface, however, Lagoona provides

powerful new abstraction mechanisms that, hopefully, will make the construction of large software systems easier and safer.

When compared to other object-oriented programming languages, the most conspicuous characteristic of Lagoona is that it lacks a *class* mechanism. Instead, it offers two separate mechanisms for classifying objects, the *type* construct and the *category* construct. Categories are less concrete than types, as they specify only a *protocol of interaction* that has to be followed by clients, rather than prescribing an actual implementation. Types, on the other hand, reify data abstractions to the level of explicit storage layouts and statement-by-statement specifications of algorithms. By offering a choice between these two structuring mechanisms, Lagoona permits the decoupling of sub-systems (with category-based interfaces between them) while simultaneously providing for the efficient integration (using type-based interfaces) of the components that comprise such a sub-system.

Categories are similar to *protocols* in Objective-C [Nex92] and *interfaces* in Java [AG96]. However, like most object-oriented programming languages, Objective-C and Java both use *classes* as the primary structuring mechanism, providing *encapsulation* and defining both an object's *structure* and its *behavior*. Lagoona's approach is very different, in that it models each of these concerns by a separate language element. In Lagoona, encapsulation is the domain of the *module* mechanism, structure is supplied by a *strong type system* that supports *type extension* [Wir88b], while behavior is provided by dynamically-bound procedures (*methods*) that are executed in reaction to *messages* sent to *objects*. Categories add a further dimension to this matrix, allowing to statically express a constraint relationship involving otherwise self-sufficient messages, types, and variables.

Lagoona's *messages* are wholly self-contained entities that are not associated with any particular object, type, or category by default. Instead, any message can be sent to any arbitrary object. This has an effect only if the receiver object has declared a corresponding method. Static typing in Lagoona can be used to guarantee that a particular receiver object *will* understand a certain message, but it cannot exclude that the object will also understand further messages not statically visible at compilation time. It is exactly this extensibility that is lacking in most statically-typed object-oriented languages, in which the definitive message protocol must be specified already for the abstract receiver type.

The remainder of this paper is organized as follows: The next section starts out by discussing two different uses of inheritance in object-oriented languages that are mutually incompatible. This incompatibility provided the initial motivation for the category concept in Lagoona. Subsequent sections of the paper then introduce Lagoona's two key concepts of *coexisting types and categories* and *stand-alone messages*, before presenting a more systematic overview of the language's features. Some familiarity with the general concepts of Oberon, particularly the *type extension* concept, is assumed. The language

overview is followed by a discussion centered around a large example. A review of related programming languages and a short section on implementation conclude our paper.

2 Specialization vs. Combination

Object-oriented programming languages such as *C++* [Str87] use *inheritance* to express structural and behavioral object commonalities. Two mutually incompatible kinds of relationships among objects can be expressed by inheritance in this manner:

- *Specialization*: an object of a specialized type has *fewer degrees of freedom* than objects of the base (parent) type. Hence, such an object can be used wherever the specification mentions only the base type; this is the principle of *substitutability* [WZ88]. The relationship between the specialized type and its parent is also called the "is-a" relationship; any object of the specialized type is simultaneously *also* an object of the base type. For example, the type *RedCar* can be modeled as a specialization of the type *Car*: every red car is a car, but not every car is red.

  Specialization is typically associated with a *top-down* design activity. The well-known technique of *stepwise refinement* [Wir71a] naturally starts off with general descriptions of data structures and keeps getting more specific as it makes progress towards the actual implementation.

- *Combination*: an object of a combination class has *more degrees of freedom* than objects of the component (parent) classes. Combination assembles new classes by *inheriting traits* from parent classes; an object of a class that has been constructed in this manner can perform every service of each of its parent classes. For example, the class *RedCarClass* could combine the traits of *CarClass* and those of *RedObjectClass*. Red cars share the "being red" characteristic with other red objects, and simultaneously also share the characteristics of cars – hence, objects of the class *RedCarClass* have *more* functionality than those of either *CarClass* or *RedObjectClass*.

  Combination is a *bottom-up* activity; before one can combine parts, one needs to construct the parts. Combination is typically used in conjunction with *application frameworks*. These provide a cast of ready-made building blocks from which programmers assemble their own. The less overlap there is among the ready-made components, the more beneficial will the effect of combination be – combining independent features is simpler than dealing with hidden dependencies.

The difference between specialization and combination is sometimes misunderstood, and often ignored due to the fact that most current languages cannot express the distinction.

Concurrent use of the two conflicting meanings of inheritance in the same program may jeopardize maintainability or even lead to errors. Some language designers avoid this dilemma by ruling out multiple inheritance and thereby also combination.

The programming language Lagoona employs two different language constructs in order to keep specialization and combination independent of each other. Specialization is modeled by *type extension* and combination by *category inclusion*, and the type and category hierarchies are separate. Our treatment of specialization and combination above almost adopts Lagoona's nomenclature already, except that the term "class" needs to be replaced by "category".

Note that *aggregation*, also called the "has-a" relationship, is a concept disjoint from both specialization and combination. In Lagoona, it is modeled by a reference from the owning to the owned object. We mention this because the "has-a" relationship is sometimes given a name similar to "combination". When we speak of combination in this paper, we mean the integration of inherited traits from multiple parent categories into an independent new category.

3 Coexisting Types and Categories

In Lagoona, types and categories coexist, providing a richer data model than is offered by other languages. Every object in Lagoona has a *type*, which may take part in a type-extension hierarchy satisfying the substitutability relationship (just like in Oberon). In this type extension hierarchy, every type has *at most one* direct base (parent) type, from which it inherits its structure. No type has more than one direct parent.

On the other side, we have the category inclusion hierarchy. Categories describe minimal sets of traits that member objects are guaranteed to provide. Inheritance of category traits is *additive* and objects can belong to several categories simultaneously. They must then offer every service mandated by every category they belong to, which can be verified statically by the compiler. Typically, categories describe relatively *small* service protocols; the expressive power of the category construct stems from the combination of *orthogonal* traits.

The separation of the category and type hierarchies affords a polymorphism that cannot be expressed in Lagoona's ancestor language, Oberon. Lagoona allows the declaration of variables not only by their *type*, but alternatively also by their *category*. Such a *category variable* is assignment-compatible with all objects that belong to the category, even if the objects among themselves are not substitutable because they originate in different type hierarchies and therefore have completely different internal structures.

As an example of the additional semantic meaning that can be expressed by separating categories and types, consider an object-oriented environment that provides a graphical editor as well as an electronic mail system. Clearly, these two applications are fairly separate; there is no need for their object types to be derived from a common ancestor – or is there? Imagine that the graphical editor provides iconized clocks that permanently display the correct time on the screen. Hence, all clock-objects need to receive *update messages* at regular intervals (an alternative architecture, by which clocks have separate processes associated to them, consumes considerable processor resources and may not be suitable for every platform). Similarly, the mailbox needs to check for the arrival of new mail items periodically, which are then displayed in a scrolling directory window. Might it not be sensible to make clocks and mail directories compatible with each other in some way, so that the same mechanism can be used for controlling their periodic updates?

In Lagoona, there is an easy answer by the definition of a category *Active* (providing a message *Tick*). The types of the graphical editor and those of the electronic mail system can then be constructed independently of each other up to the point at which they need the common behavior of being able to receive *Tick* messages. This common behavior is injected into both type hierarchies by membership in the category *Active* (Figure 1). Because of their common category membership, arbitrary instances of *Clock* and *MailDirectory* can then be assigned to *Active*-categorized variables and passed as actual parameters to suitable procedures (with formal parameters of category *Active*), in spite of the fact that they have no common type-ancestor[1].

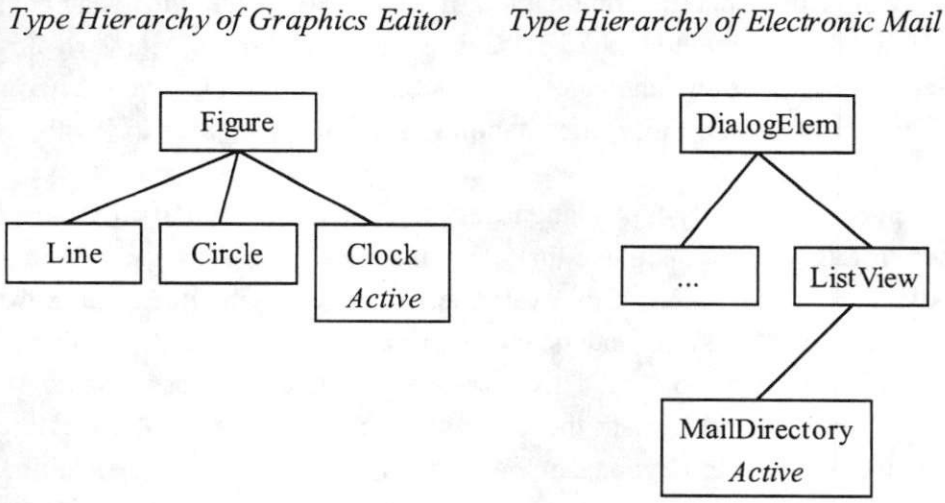

Figure 1: Two Separate Type Hierarchies That Share Behavior in Leaf Nodes

[1] In Lagoona, objects in fact do not belong to categories individually, but whole types are associated with categories; this is explained below.

As the example illustrates, categories allow to introduce common behavior into separate type hierarchies in a consistent manner while preserving static verifiability. Membership in the category *Active* guarantees that objects understand a message *Tick*. Hence, if the procedures of the tasking mechanism have formal parameters of category *Active*, then only "compatible" objects can be passed to them – but these objects need not be derived from just a single type. The latter is important, because it enables the independent construction of applications that can nevertheless access common services. It makes the re-use of existing libraries much simpler than would be the case if type-structure needed to be inherited.

4 Stand-Alone Messages

In most object-oriented languages that provide static typing and checking as their leading principles, *messages* belong to the scopes of classes and are intimately bound to them. This facilitates the efficient implementation of message dispatch using virtual function tables [ES90] and related techniques. Lagoona, on the other hand, treats messages as independent language elements. It also has weaker compatibility rules for message-sends than for typing, which makes extensible programming particularly simple and elegant.

In Lagoona, messages are not associated *a priori* with any particular type or category, and any message can be sent to any object in principle. The effect of sending a message to an object is governed by the presence or absence of a *method* that associates a certain behavior with a specific combination of receiver object and message being sent; when there is no such method, the message is simply ignored. Many weakly-typed object-oriented programming languages, such as Smalltalk [GR83] have similar message semantics, but in Lagoona, more information is statically available about the receiver object.

Just as in other statically-typed languages, the type declaration of a receiver variable can be used in Lagoona to guarantee that a certain message protocol *will be* understood by every dynamic instantiation of that variable. What *cannot* be ruled out, however, is that the same receiver will also understand *additional* messages that are not part of the statically visible message protocol. This becomes particularly important in combination with Lagoona's *resend* mechanism that allows objects to *forward* messages they have received to other objects. In Lagoona, an object forwarding a message need not be able to "understand" it at all.

Stand-alone messages and the resend mechanism can be applied together to implement *generic iteration* over compound data structures. For example, an object representing an abstract "container" data structure (such as a linked list or a tree) can be sent messages destined for one or more of its elements. In Lagoona, it is simple to

program the container to accept *arbitrary messages* and distribute them to its elements. In most other statically-typed languages, the set of messages that such a container can receive and forward would have to be specified already in the interface of the container.

For the purpose of statically guaranteeing message comprehension, *categories* play an important role. A category may be associated with a message protocol, i.e., with a number of messages that together as a group form an interface. Every object that belongs to the category *must* then provide the corresponding methods, and this is verified by the compiler. Hence, when objects are passed as parameters to sub-systems whose interfaces specify categories rather than types, there are statically verifiable minimal guarantees as to which message protocols the actual parameter objects understand.

By turning messages into independent elements of the language (which are completely unrelated to each other *per se*) rather than subjugating them under classes, Lagoona also avoids the *repeated inheritance* dilemma that has plagued the designers of object-oriented programming languages. This dilemma arises when a class *C* inherits from two parent classes *P* and *Q*, which in turn have a common ancestor class *R*: does *C* then inherit the common elements of *P* and *Q* twice (once via each path), or only once (in which case one would have to specify a precedence for inherited methods). This question is resolved differently in different programming languages; there is no consensus as to which solution is preferable. Instead of multiple inheritance, Lagoona offers *multiple category membership*. Category membership merely specifies a minimum functionality that each category member has pledged to provide, it has nothing to do with inheritance. Hence, the protocols of individual categories may overlap without any conflict.

5 Elements of the Programming Language Lagoona

In the following, we will introduce Lagoona as a variant of Oberon [Wir88a]. A basic familiarity with the syntax of Oberon and the concept of type extension [Wir88b] is assumed.

5.1 Objects and Object Types

An object is an instance of an *object type*. Each object type is based on a record type that describes the internal structure of objects, and may additionally be associated with one or more *categories*. Object types are similar to pointer types, except that the variables declared in this manner gain the added capability of being able to receive messages. In Lagoona, just as in Oberon, exported identifiers are marked in the source text by a trailing asterisk; for emphasis, they are also rendered in boldface here.

Example:

```
TYPE
   Figure* = OBJECT FigureCategory BASED ON FigureDesc;
   FigureDesc* = RECORD
      x*, y*, w*, h*: INTEGER;  (* bounding box *)
      link: Figure
   END;
```

The type *Figure* in the example above (that we imagine to lie in a module called *Graphics*) has been declared as an object type associated with the category *FigureCategory*; we also say it *belongs* to the category. Each object of this type will have five instance variables, namely four integers representing the bounding box of its graphical representation, and a hidden (i.e., non-exported) reference to another object.

Oberon's type-compatibility rules for pointer types can be extended to object types in a straightforward manner. Hence, in Lagoona, type-compatibility between variables of different object types is established by the compatibility of the underlying record-types. An additional requirement needs to be imposed, however: an object type E is an extension of another object type B only if E belongs to all of the categories that B belongs to; it may also belong to additional ones.

In the following example, the object type *Circle* (in a different module *Circles*) has been declared as an extension of the object type *Figure* above. The extension relationship between the two object types is established via the base types *CircleDesc* and *FigureDesc*, and is validated by the fact that *Circles.Circle*, just like *Graphics.Figure*, belongs to the category *Graphics.FigureCategory*. In addition, circles also belong to the category *CircularCategory* declared in module *Shapes*.

Example:

```
TYPE
   Circle* = OBJECT Graphics.FigureCategory, Shapes.CircularCategory
            BASED ON CircleDesc;
   CircleDesc* = RECORD (Graphics.FigureDesc)
      r*: INTEGER
   END;
```

5.2 Messages

Messages are signals that can be sent to objects. A message consists of a name and a formal-parameter-list signature. Messages are not associated *a priori* with any particular object type; in Lagoona, any message can be sent to any object. Sending a message has an effect only if the receiving object possesses a corresponding method for the message. Such a method is guaranteed to exist when the receiving object belongs to a category that

lists the message in its protocol. Lagoona does not allow the overloading of message names (with different parameter lists) within the same module.

Message declarations form a separate section of Lagoona programs, on the same level as constant declarations, type declarations, variable declarations and category declarations. The example below shows the declaration of four different messages. Although these messages have been declared with the application of a graphical editor in mind, they are completely non-specific and may be used also by unrelated client modules.

Example:

```
MESSAGE
   Draw*();
   Selectable*(x, y: INTEGER): BOOLEAN;
   Read*(VAR R: Files.Rider);
   Write*(VAR R: Files.Rider);
```

5.3 Categories

Messages can be grouped together in a *category* to form a consistent interface. Every object type that belongs to a certain category *must* have methods for every message of the category; this condition is verified by the compiler. Hence, membership of a receiver object in a certain category provides the guarantee that all messages declared in the category will be understood by the object.

In our example below, membership in the previously mentioned category *Graphics.FigureCategory* is revealed to mandate the implementation of the methods *Graphics.Draw*, *Graphics.Selectable*, *Graphics.Read*, and *Graphics.Write*.

Example:

```
CATEGORY
   FigureCategory*= (Draw, Selectable, Read, Write);
```

Note that the protocols mandated by different categories can overlap. For example, a completely unrelated category *UnrelatedCategory* might also be using the message *Graphics.Draw* in its protocol. An object type that is a member of both *FigureCategory* and *UnrelatedCategory* still only requires a single method implementing the message *Graphics.Draw*; there are no resulting name conflicts (as are possible with some implementations of multiple inheritance).

Also note that each object type can belong simultaneously to arbitrarily many categories, and that every category can have arbitrarily many member types. Two object types that are both members of the same category can be completely disjoint, i.e. they don't need a common structural ancestor in order to implement the same category

protocol. This is the key to the loose coupling of sub-systems possible in the Lagoona language.

5.4 Methods

A method specifies the actions that are performed when a message is received by an object of a certain object type. Each method is associated with exactly one message and exactly one object type, and must be declared in the same module as the receiver type it is associated with. There is no automatic method inheritance in Lagoona.

The example below shows an implementation sketch of the message *Graphics.Read* for the object type *Circles.Circle*. Since circles belong to the category *Graphics.FigureCategory*, the compiler will output an error message if no such implementation is present in module *Circles*.

Example:

```
METHOD Graphics.Read(VAR r: Files.Rider) -> c: Circle;
BEGIN  (* read the parameters c.x, c.y, and c.r from rider r *)
END Graphics.Read;
```

5.5 Variables: Typed vs. Categorized

Objects can be assigned to suitable variables. In Lagoona, such variables can be declared not only by the *type* of object that is assignable, but alternatively also by a *category*. In the following, we will refer to the first kind as *object-typed variables* and to the second one as *categorized variables*.

Example:

```
VAR
   fig: Figure;   (* object-typed variable *)
   grafobj: FigureCategory;   (* categorized variable *)
```

Lagoona's *object-typed variables* are similar to pointer variables in Oberon, except that one can also send messages to them. More interesting are *categorized variables*, which can represent objects of any type that belongs to a certain category. Just as with typed variables, messages can be sent to objects represented by categorized variables. Both kinds of variables can be compared to *NIL*, and the value *NIL* can be assigned to them. The operation *NEW* can be performed only on pointer variables and object-typed variables, but not on categorized variables.

Categorized variables are assignment-compatible among each other only within the same category. Assignments from typed variables to categorized variables are legal if the

type of the typed variable belongs to the category of the categorized variable. To enable type-safe execution of assignments in the reverse direction, categorized variables can be subjected to *Type Guards*, by which operation they become *typed*. This allows to recover the type information of an object assigned to a categorized variable.

Categorized variables introduce a kind of polymorphism into the Lagoona language that isn't available in Oberon, as they enable the programmer to concentrate on an object's behavior without having to deal with structural aspects. They also allow to express commonality of behavior across type hierarchies. Any object that may be assigned to a certain categorized variable has pledged to fulfill the contract stipulated by the interface of the category, but it need not have been derived from a common ancestor type. This enables the black-box reuse of existing sub-systems using different, but protocol-compatible types. For an extended example, see the module *PriorityQueues* below.

5.6 Message Dispatch

The phrase "sending a message to an object" means the invocation of the method associated with the combination of dynamic receiver type and message being sent. If no such method exists, the send operation is ignored. Syntactically, sending of a message is denoted by a *message activation designator*. This consists of the name of the message, the formal parameters (if any), the send arrow "->" and a (categorized or object-typed) receiver designator.

Example:

```
Graphics.Read(r0) -> fig;
```

Note that the message designator appears *before* the receiver designator; this reflects the fact that (unlike in other object-oriented languages) messages are completely independent of objects and do not belong to their scopes.

5.7 Message Forwarding

Sometimes, a message cannot be handled by its original receiver, but needs to be forwarded to one or more other objects. For this purpose, every method in Lagoona can forward its current message using the keywords *RESEND TO*. This is equivalent to sending the original message again explicitly, passing the *current values* of the original parameters in the original order. An example for this feature is provided in the next paragraph.

Unlike *super-calls* in other languages, in which the self-reference remains bound to the original receiver object, resending a message establishes a new receiver. If the original receiver needs to be preserved along the calling chain, it can simply be passed as an explicit parameter of the message. The semantics of resend are easier to understand than those of super-calls, and more localized in the source text, leading to programs that are simpler to maintain.

5.8 Default Methods

An object type may specify a *default method* that is executed whenever no specific method is defined for a particular arriving message. The default method has no parameters, but can replace *proper methods* with arbitrary parameter lists. Default methods do not apply to *functional methods* that return a result value. By providing a default method, a type is relieved from the requirement to supply a separate method implementation for every message in the categories that it belongs to; it then needs to implement only the functional methods. Default methods are also a convenient feature for debugging.

The following example comes from the Oberon System [WG89, WG92]. In this system, objects of type *Viewer* represent display windows. Whenever such a Viewer receives a message that it doesn't understand, it simply forwards it to the two sub-frames that represent its menu bar and content area.

Example:

```
METHOD DEFAULT -> v: Viewer;
BEGIN
   RESEND TO v.menuFrame
   RESEND TO v.contentFrame
END DEFAULT;
```

This programming style, in which messages "trickle down" a structural hierarchy until they are finally understood at some level, has been employed with great success in the Oberon System. Conversely, in conventional object-oriented systems, messages typically only "trickle down" an inheritance hierarchy, using the *super-call* mechanism. The scheme used in the Oberon System, which is given added language support in Lagoona, is more flexible. At the same time, it avoids the intricate self-recursion patterns of inheritance that can make program comprehension very difficult.

5.9 Generic Broadcast

Default methods, along with *RESEND*, also make possible an elegant *generic broadcast* mechanism. Imagine that the display sub-system offers a mechanism, by which a message can be sent to all of the viewers on the screen. The list of viewers is maintained by the display sub-system and is considered private to it. In the example below, the display system exports a distributor object *Viewers.all* that will broadcast any method sent to it to the whole private display data structure.

```
MODULE Viewers;

   TYPE
      Viewer* = OBJECT BASED ON ViewerDesc;
      ViewerDesc* = RECORD
         state*: INTEGER;
         ...
         link: Viewer
      END;

      Distributor = OBJECT BASED ON RECORD
         root: Viewer;
      END;

   VAR
      all*: Distributor;

   METHOD DEFAULT -> d: Distributor;
      VAR v: Viewer;
   BEGIN  v := d.root;
      WHILE  v # NIL  DO
         RESEND TO v;
         v := v.link
      END
   END DEFAULT;

   ...

END Viewers.
```

Note that this style of programming is aesthetically consistent with the "trickle down" approach of message distribution (as in the example above), in which the same message is distributed to several recipients if an object has multiple descendants. In this respect, the approach of Lagoona is preferable over one in which messages can be passed as explicit parameters to proper procedures that then perform the distribution (as implemented in the Oberon System without specific language support for messages).

Also, the *Viewers.all* object is now tangible, which opens the path to novel software architectures. For example, consider the following change to the module above:

```
CATEGORY
   DistributorCategory* = ();

TYPE
   Distributor = OBJECT DistributorCategory BASED ON RECORD ... END;
```

It now becomes possible to parametrize a whole sub-system by the distributor object required for notification, using a categorized variable of category *DistributorCategory*. This provides a unprecedented flexibility. For example, in the Oberon System's implementation of the *Model-View-Controller* [KP88] triplet, models notify the viewer hierarchy by "up-calling" an ordinary procedure. This procedure is installed during creation of the model-object, leading to an unnatural distinction between displayable models (e.g. in the Oberon System, *Texts* created using the procedure *TextFrames.Text*) and non-displayable ones. Using the above mechanism, distributor objects can be passed downwards to a model object with every message. If the model changes in response to the message it has received, it notifies the distributor contained in the message.

5.10 The Category ANY

Every object type belongs to the predefined category *ANY*. Apart from the fact that object types need not explicitly mention that they belong to it, *ANY* behaves just like categories declared by the programmer. *ANY* is useful for programming completely generic algorithms. Note that *ANY* is *not* an unsafe feature, unlike mechanisms in other programming languages that allow to circumvent the type system, such as the *SYSTEM.PTR* construct of Oberon-2 [MW91].

The fundamental problem of the type *SYSTEM.PTR* in Oberon-2 (and similar loopholes in many other languages) lies in the fact that all pointers are compatible with *variable* parameters of this type. The type of a pointer-variable bound to such a *VAR*-parameter can be changed inside of a procedure (by an assignment), and the changed value be passed back outside in violation of the type system. In Lagoona, on the other hand, the type compatibility rules apply also to the category *ANY*. These rules mandate that the type of the formal parameter and that of the actual parameter must be *identical* for call-by-reference (but not for call-by-value). Hence, if an object is passed into a procedure as *ANY*, it must also be passed out as *ANY*. Before it can be further assigned to a typed variable, it must always be subjected to a type guard.

6. Discussion

The following section elaborates on the usefulness of novel aspects of the Lagoona language and relates them to other work.

6.1 Locality of Method Implementations

Lagoona has no automatic method inheritance; the association between methods that can be sent to objects and the actions that are performed in response have to be provided explicitly by the programmer. The compiler is merely able to inform the programmer if he omits any such association that is mandated by category membership. The visibility rules of Lagoona make it possible to hide a message and all of the corresponding methods wholly within a module. However, if a message is exported, then all types that implement a method for it (even those in other modules) must advertise this fact in their public interface. Hence, the clients of an object type can see "from the outside" if it will react to some specific message, unless also the method is invisible to them.

The exclusion of method inheritance from Lagoona is in acknowledgement of the fact that it can lead to loss of *locality*, forcing the programmer to "browse" through inheritance hierarchies to gather information about an object's behavior. In many languages, the specification of a class always includes the *full* specification of all of its super-classes (not just of their interfaces); some language definitions, for example that of *Sather* [SOM94], go as far as to *define* inheritance to be equivalent to textual inclusion of source code. In Lagoona, a conscientious decision has been made to implement all object behavior explicitly in the same module that contains the type declaration. Of course, this does not rule out code reuse by way of ordinary procedure activation across module boundaries. A related gain of locality is achieved by replacing the semantics of *super-call* by the much simpler ones of *resend*.

Lagoona also does not provide "friend functions" [Str87], i.e. procedures that can access the private details of objects although they lie in another scope. Visibility in Lagoona is controlled strictly by the module mechanism; if a type hides some of its data fields, then even the extensions of that type lying in other modules will not be able to access the fields directly. If these invisible fields need to be manipulated by extensions in any way, then the module hiding them must provide procedures for this purpose. This is the only way to guarantee the *invariants* of the hidden data structure, and a necessary prerequisite for black-box reuse (i.e. without access to the source text) of existing object types. "Friend functions" are necessary only in languages without proper modularization facilities.

6.2 An Example for the Use of Categories

Lagoona offers the system designer a choice whether an interface should specify a *type* or a *category*. The former leads to a tight coupling between modules, and to efficient code. The latter, on the other hand, provides the necessary flexibility to structure complex software systems into long-lived and independently maintained sub-systems. Since no storage representation is assumed in this interface, each of the sub-systems can be considered an independent "black box", with the contract between sub-systems consisting solely of the category interface. An object type can belong to several categories concurrently and thereby be combined with several sub-systems.

For example, the following module *PriorityQueues* implements an abstract data type (object type) *Queue* that parametrizes its elements by a category-based definition. Any object that belongs to the category *PriorityQueues.ElemCategory* can be put into such a queue. The category-based interface serves a dual purpose here: It establishes type safety, by requiring every queue-element to posses a method *Priority* that is invoked for inserting the object at the correct position. But furthermore, membership in *PriorityQueues.ElemCategory* also serves as a valuable documentation aid to the programmer, because it expresses information about the intended uses of an object type. This makes the resulting program much more transparent than any inheritance of a "priority-queueable" trait from a "general object" type could ever be.

```
MODULE PriorityQueues;

  CATEGORY
    ElemCategory* = (Priority);        (* every element-type must provide
                                          a Priority method *)
  MESSAGE
    (* element messages *)
      Priority*(): LONGINT;

    (* queue messages *)
      Init*;
      Put*(elem: ElemCategory);
      Get*(): ElemCategory;

  TYPE
    Link = POINTER TO LinkDesc;        (* note that this is NOT an object type
    LinkDesc = RECORD                     and that it is NOT exported *)
      link: Link;
      elem: ElemCategory
    END;
```

```
    Queue* = OBJECT BASED ON QueueDesc;
    QueueDesc = RECORD
        first: Link
    END;

    Sentinel = OBJECT ElemCategory BASED ON RECORD END;


METHOD Priority(): LONGINT -> s: Sentinel;
BEGIN  RETURN MAX(LONGINT)
END Priority;

METHOD Init* -> q: Queue;
    VAR s: Sentinel;
BEGIN  NEW(s);  NEW(q.first);  q.first.elem := s;  q.first.link := q.first
END Init;

METHOD Put*(elem: ElemCategory) -> q: Queue;
    VAR cur, new: Link;
BEGIN  cur := q.first;
    WHILE  (Priority -> cur.link.elem) < (Priority -> elem)  DO  cur := cur.link  END;
    NEW(new);  new.elem := elem;  new.link := cur.link;  cur.link := new
END Put;

METHOD Get*(): ElemCategory -> q: Queue;
    VAR cur: Link;
BEGIN
    IF  q.first.link = q.first  THEN  RETURN NIL
    ELSE  cur := q.first.link; q.first.link := cur.link;  RETURN cur.elem
    END
END Get;

END PriorityQueues.
```

The module *PriorityQueues* is completely self-contained and can be used without knowledge of its implementation. Programmers wishing to use its services need to derive their objects from *PriorityQueues.ElemCategory*, which means that they have to supply a *Priority* method, but they don't need to inherit any structure and thereby become dependent on changes in the implementation of *PriorityQueues*. The coupling between *PriorityQueues* and its clients is only as strong as necessary for guaranteeing proper interaction. Also note that the module *PriorityQueues* implements an internal object type *Sentinel* that is not visible on the outside. Sentinels belong to *ElemCategory* and hence require a *Priority* method.

Taken together, categories and types give programmers a choice of *abstraction level* to base their interfaces on. The more general the interface, the easier the re-use of the component without access to the implementation, but in general also the less efficient. For example, if we had known that every object could appear in at most one priority

queue at a time, we might have used a type-based interface in which the *link* field was part of the objects themselves.

```
MODULE RestrictedPriorityQueues;

   TYPE
      Elem* = POINTER TO ElemDesc;
      ElemDesc* = RECORD
         link: Elem
      END;

      Queue* = Elem;

   ...

END RestrictedPriorityQueues.
```

Clients of the simpler module *RestrictedPriorityQueues* would need to define their objects as extensions of the type *RestrictedPriorityQueues.Elem*, so that they would inherit the field *link*. They would also need to know enough about the implementation to take into account that objects can belong to only one queue at a time. Note however that the *link* field in the example below is not exported; object types that are derived from *RestrictedPriorityQueues.Elem* will inherit it, but they cannot manipulate it directly. This serves to guard the invariants of *RestrictedPriorityQueues*.

6.3 Applications for Category-Based Interfaces

The loose coupling between sub-systems that is afforded by Lagoona's category interface is particularly beneficial in conjunction with *run-time-extensible systems* based on *compound-document component architectures*. A compound document is a container that seamlessly integrates various forms of user data, such as text, graphics, and multimedia. These various kinds of content are supported by independent *content editors* ("applets") that cooperate in such a way that they appear to the end-user as a single application program.

In order to maintain the illusion of a united application, the various applets need to interact closely. They have to negotiate the use of shared resources, such as the display space (which can become non-trivial in the case of overlapping objects and irregular shapes), and hence need to be able to communicate with each other and with the application controlling the container document. On the other hand, applets need to be self-sufficient enough that they can be developed and distributed independently of each other.

Lagoona's differentiation between categories and types fits this situation well. Category-based interfaces can be used to describe the interactions between applets, while each individual applet provides its own unique type hierarchy that is disjoint from all the others. Categories represent the interfaces between applets; they are general and long-lived. Types instantiate these interfaces and can be defined and re-defined in a flexible manner.

6.4 Related Work on Programming Languages

As early as 1986/87, *Snyder* [Sny87] and *Liskov* [Lis87] have differentiated between *implementation hierarchies* and *type hierarchies*, and advocated the use of separate mechanisms in programming languages to model the two. In her paper, Liskov presents a strong argument in favor of multiple implementations of the same category by different (disjoint) types. We also note that both papers use the term *class* in the same meaning as we use *type*, while they use *type* where we use *category*. Lagoona's particular nomenclature has historical reasons, as it continues the Algol-Pascal-Modula-Oberon lineage in which the term *type* has a well-understood meaning.

The programming language *Emerald* [RTL91] has similarities with Lagoona, in that it also doesn't provide automatic *code inheritance*, but only *structure inheritance* by way of type extension. Emerald stresses the concept of *locality* (which its authors call *object autonomy*) by forcing all behavior to be encapsulated within the definition of each individual object type. In Emerald, the domain of encapsulation is the type; there is no separate module concept (which would have aided the construction of larger systems). Emerald also has a type *ANY* with similar properties to its counterpart in Lagoona. An interesting aspect of Emerald is the fact that it bases object substitutability on *interface conformity* (rather than common type-ancestry); hence multiple implementations of the same category (by our nomenclature) are possible.

The language *Objective C* [Nex92] offers a construct called *protocol* that is similar to the category mechanism of Lagoona. Protocols declare methods that are not associated with a fixed class, but which any class, and perhaps many classes, might implement. However, messages are not separate stand-alone entities in Objective C as they are in Lagoona; if a method is shared between two protocols, this can be expressed only by providing two textually equal definitions. Hence, the orthogonal structure of Lagoona that allows to assemble arbitrary messages originating in different modules into new protocols cannot be duplicated so easily. Objective C provides (single-parent) method inheritance. The programming language Java [AG96] adopts the protocol concept of Objective-C under the new name *interface*.

The language *Sather* [SOM94] differentiates between a *subtyping relationship* and a *code inheritance relationship*, which can form separate hierarchies. The idea of a code

inheritance hierarchy is interesting (albeit dangerous), as it allows the construction of classes whose methods can be "mixed into" several different type hierarchies simultaneously. Recall that inheritance in Sather is defined as being equal to source code inclusion: several disjoint types that simply happen to use instance variables of the same names can then share some "mix-in" methods by inheriting from the corresponding class. Note that the "mixed-in" methods have full access to the internal state of an object; in this respect they are similar to the *friend* functions of *C++* [Str87].

Sather also has the interesting property that only variables of *abstract types* can be polymorphic. Hence, in some respect, Sather's abstract types are similar to Lagoona's categories: both represent unions of object types without having concrete instances of their own. Unfortunately, however, abstract types in Sather follow the normal rules of *subtyping* (in fact: multiple subtyping, with all the associated problems) and don't represent an orthogonal hierarchy as the categories of Lagoona do. In general, Sather is a much more complex language than Lagoona.

We know of no existing language that separates all three concepts of *structure*, *behavior*, and *messages* in the same way as Lagoona does. There are, however, other languages that turn messages into separate stand-alone entities; these are the languages offering *generic functions* (sometimes also called *multi-methods*). Generic functions originated in the realm of functional programming, for example, they are the basis of the *Common Lisp Object System* (CLOS) [DG87]. A generic function is a group of methods implementing a certain behavior for different types of receiver. In order to activate a desired behavior, one invokes the corresponding generic function, which autonomously chooses one of its constituent methods, for example, based on the types of the specified arguments. Hence, while in traditional object-oriented languages methods are grouped together by the classes to which they apply, generic functions group them together by the operations they perform. Lagoona's messages represent the same mechanism as generic functions that type-dispatch on their first argument, in a context of strong typing and category-based classification.

The work of Harrison and Ossher [HO90] on *subdivided procedures* is also similar to Lagoona's messages and generic functions. Their system provides *functional extension* by the addition of *alternate procedure bodies* to a procedure, which are selected based on criteria specified by the programmer. The addition of methods to an existing message (with a new type of receiver argument) can be viewed as a specialization of the subdivided procedure mechanism with regard to the subdivision criterion (dispatch on type only).

6.5 Implementation

In adapting an existing Oberon compiler for Lagoona, only the *method dispatch mechanism* requires changes in the back-end and in the linker. All other features of Lagoona can be handled locally in the front-end of the compiler. There is no need to add any run-time support for the category mechanism, since Oberon already provides run-time type information for its polymorphic variables and parameters. Note that Lagoona doesn't offer an explicit mechanism for testing category membership, but merely proper type tests (which already exist in Oberon).

In Lagoona, methods are bound to (pointer-valued) object types, and not to the underlying record-types. Hence, two different object types that are based on the same record can provide their object instances with different behavior. Binding methods to object types requires a separate descriptor for every object type. In contrast, in the language Oberon-2 [MW91], methods (called type-bound procedures in the defining report) are bound to *record types*.

Two different ways of implementing Lagoona's method dispatch mechanism immediately come to mind. The straightforward, albeit inefficient solution is based on the programming technique of *message handlers* that is widely employed in the Oberon System; in fact, this programming technique inspired some of the features of Lagoona (which are safer and more descriptive than those of its precursor language). Using this technique, each message is assigned an arbitrary unique identifier and the different methods that apply to a single object type are strung together in a single piece of code that tests for the different cases explicitly. Every object type has such a message handler associated with it; it forms part of its *type descriptor*.

Example:

```
(* implementation of all methods for object type T *)
BEGIN
   IF  message = M1  THEN  (* code for method M1 *)
   ELSIF  message = M2  THEN  (* code for method M2 *)
   ...
   ELSE  (* code for default method, if one exists *)
   END
END
```

An alternative, more efficient solution uses a *method table* for dispatch (Figure 2). Such a table provides a direct reference to the implementing method for every combination of message and type. Hence, it can become quite large, although only a small fraction of the entries are used. Rather than maintaining one large system-wide method table, implementations are therefore likely to store the table as a number of separate rows, each of which is associated with a particular type, or a number of separate columns, each of which is associated with a specific message. The first of these organizations is similar to

the well-known *virtual function tables* [ES90], while the second comes closer to understanding messages as type-dispatched generic functions. Since types and messages are both stand-alone entities of the Lagoona language, both alternatives have straightforward implementations. The size of the method table represents a challenge, but fortunately the compression of sparse dispatch tables is an active area of research that has promising results [AGS94, DH95].

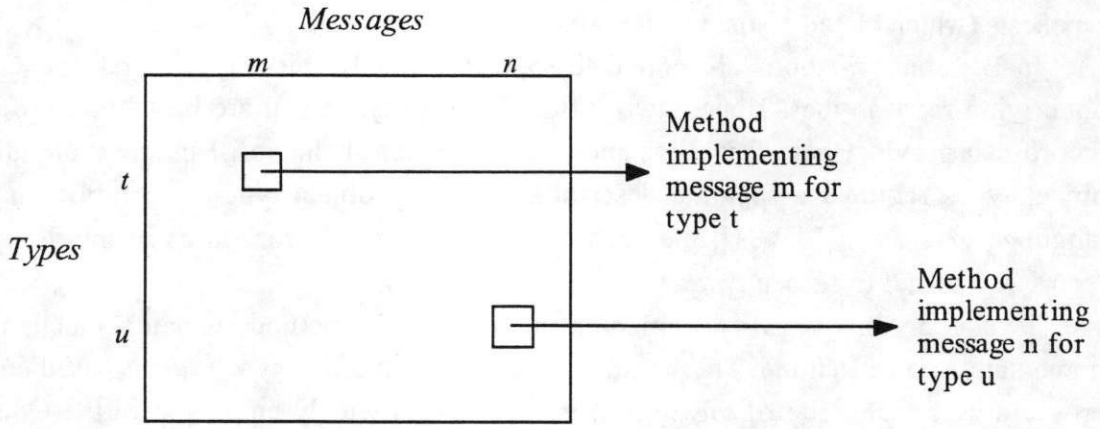

Figure 2: Message Table for Type-Dispatch

An added complication arises when Lagoona is used in an environment supporting dynamic loading. This means that further modules can be added to a running system at any time. As a result, message and type numbers (i.e., the indices into the method table) cannot be assigned statically by the compiler, but only at the time of loading. A previous paper by the author [Fra95] explores the resulting problems in some detail and describes an implementation that solves them. Note that the language presented in the earlier paper went even further than Lagoona, allowing the addition of further methods to existing types by external modules, and also supporting code inheritance. These features have been discarded in Lagoona in favor of locality.

7. Conclusion

Lagoona is a simple, object-oriented language that continues the tradition of Pascal, Modula-2 and Oberon. Rather than providing a *class* construct, Lagoona adds *messages* and *methods* to the *type extension* mechanism already present in Oberon. Further, Lagoona offers *categories* that facilitate a weaker coupling between sub-systems than

possible with types. Its innovative language features make Lagoona an attractive tool for developing extensible object-oriented systems.

Acknowledgement

The author would like to thank Martin Reiser, who commented thoroughly on this paper and provided valuable criticisms that helped to improve the presentation of this material considerably. Thanks are also due to Mark Hamburg for his thoughtful comments on many aspects of Lagoona.

References

[AG96] K. Arnold and J. Gosling; *The Java Programming Language*; Addison-Wesley, 1996.

[AGS94] E. Amiel, O. Gruber, E. Simon; "Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables"; in *OOPSLA '94 Proceedings*, published as *ACM Sigplan Notices*, 29:10, 244-258; 1994.

[DG87] L. G. DeMichiel and R. P. Gabriel; "The Common Lisp Object System: An Overview"; in *ECOOP '87 Proceedings*, Springer Lecture Notes in Computer Science, No. 276, 151-170; 1987.

[DH95] K. Driesen and U. Hölzle; "Minimizing Row Displacement Dispatch Tables"; in *OOPSLA '95 Proceedings*, published as *ACM Sigplan Notices*, 30:10, 141-155; 1995.

[ES90] M. A. Ellis and B. Stroustrup; *The Annotated C++ Reference Manual, ANSI Base Document*; Addison-Wesley; 1990.

[Fra95] M. Franz; "Protocol Extension: A Technique for Structuring Large Extensible Software-Systems"; *Software-Concepts and Tools*, 16:2, 86-94; 1995.

[GR83] A. Goldberg and D. Robson; *Smalltalk-80: The Language and its Implementation*; Addison-Wesley; 1983.

[HO90] W. Harrison and H. Ossher; "Subdivided Procedures: A Language Extension Supporting Extensible Programming"; in *Proceedings of the 1990 International Conference on Computer Languages*, IEEE Computer Society Press, 190-197; 1990.

[KP88] G. E. Krasner and S. T. Pope; "A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-89"; *Journal of Object-Oriented Programming*, 1:3, 26-49; 1988.

[Lis87] B. Liskov; "Data Abstraction and Hierarchy"; in *OOPSLA '87 Addendum to the Proceedings*, published as *ACM Sigplan Notices*, 23:5, 17-34; 1988.

[MW91] H. Mössenböck and N. Wirth; "The Programming Language Oberon-2"; *Structured Programming*, 12:4, 179-195; 1991.

[Nau60] P. Naur (Editor); "Report on the Algorithmic Language Algol 60"; *Communications of the ACM*, 3:5, 299-314; 1960.

[Nex92] *The Objective C Language, Release 3.0*; NeXT Computer, Inc.; 1992.

[RTL91] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul; "Emerald: A General-Purpose Programming Language"; *Software-Practice and Experience*, 21:1, 91-118; 1991.

[Str87] B. Stroustrup; *The C++ Programming Language*; Addison-Wesley, Reading; 1987.

[Sny87] A. Snyder; "Inheritance and the Development of Encapsulated Software Components"; in *Research Directions in Object-Oriented Programming*; The MIT Press; 1987.

[SOM94] C. Szyperski, S. Omohundro, and S. Murer; "Engineering a Programming Language: The Type and Category System of Sather"; in *Programming Languages and System Architectures*, Springer Lecture Notes in Computer Science, No. 782, 259-281; 1994.

[WZ88] P. Wegner and S. B. Zdonik; "Inheritance as an Incremental Modification Mechanism, or, What Like Is and Isn't Like"; *ECOOP'88 Proceedings*, Springer Lecture Notes in Computer Science, No. 322, 55-77; 1988.

[Wir71a] N. Wirth; "Program Development by Stepwise Refinement"; *Communications of the ACM*, 14:4, 221-227; 1971.

[Wir71b] N. Wirth; "The Programming Language Pascal"; *Acta Informatica*, 1:1, 35-63; 1971.

[Wir82] N. Wirth; *Programming in Modula-2*; Springer; 1982.

[Wir88a] N. Wirth; "The Programming Language Oberon"; *Software-Practice and Experience*, 18:7, 671-690; 1988.

[Wir88b] N. Wirth; "Type Extensions"; *ACM Transactions on Programming Languages and Systems*, 10:2, 204-214; 1988.

[WG89] N. Wirth and J. Gutknecht; "The Oberon System"; *Software-Practice and Experience*, 19:9, 857-893; 1989.

[WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992.