Dimensions of Object-Based Language Design

Peter Wegner
Department of Computer Science
Brown University
Providence RI 02912

Abstract:

The design space of object-based languages is characterized in terms of objects, classes, inheritance, data abstraction, strong typing, concurrency, and persistence. Language classes (paradigms) associated with interesting subsets of these features are identified and language design issues for selected paradigms are examined. Orthogonal dimensions that span the object-oriented design space are related to non-orthogonal features of real languages. The self-referential application of object-oriented methodology to the development of object-based language paradigms is demonstrated.

Delegation is defined as a generalization of inheritance and design alternatives such as non-strict, multiple, and abstract inheritance are considered. Actors and prototypes are presented as examples of classless (delegation based) languages. Processes are classified by their degree of internal concurrency. The potential inconsistency of object-oriented sharing and distributed autonomy is discussed, suggesting that compromises between sharing and autonomy will be necessary in designing strongly typed object-oriented distributed database languages.

1. Design Space for Object-Based Languages

In order to examine design alternatives for object-based languages the following "dimensions" of language design are considered:

objects
classes
inheritance
data abstraction
strong typing
concurrency
persistence

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commerical advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

© 1987 ACM 0-89791-247-0/87/0010-0168 $1.50

These features span the design space of object-based languages. Each relates to a different aspect of computational behavior. Objects are autonomous entities that respond to messages or operations and share a state. Classes classify objects by their common operations. Inheritance serves to classify classes by their shared behavior. Data abstraction hides the representation of data and the implementation of operations. Strong typing imposes static constraints on the applicability of operations, both within and among objects. Concurrency allows objects to execute concurrently with other objects and to have internal concurrency. Persistence allows object identity to persist across applications and to be independent of values or keys used in object selection.

Language classes worthy of special study are identified and the efficiency, simplicity, and methodology of the associated paradigms is examined. We first consider just objects, classes, and inheritance, then add data abstraction and strong typing, and finally consider concurrency and persistence. Along the way global properties of design dimensions such as consistency and orthogonality are introduced and related to non-orthogonal features that occur in real languages.

2. Objects, Classes, and Inheritance

Objects have the following properties:

object: An object has a set of "operations" and a "state" that remembers the effect of operations. Objects may be contrasted with functions, which have no memory. Function values are completely determined by their arguments, being precisely the same for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience, its reaction to an operation being determined by its invocation history.

The term "object-based language" may now be defined as follows:

object-based language: A language is object-based if it supports objects as a language feature.

Support of objects is a necessary but not sufficient requirement for being object-oriented. Object-oriented languages must additionally support object classes and class inheritance:

object-oriented language: An object-based language is object-oriented if its objects belong to classes and class hierarchies may be incrementally defined by an inheritance mechanism. That is:

object-oriented = objects + classes + inheritance

The notions "class" and "inheritance" used in the above definition can be defined as follows:

class: A class is a template (cookie cutter) from which objects may be created by "create" or "new" operations. Objects of the same class have common operations and therefore uniform behavior. Classes have one or more "interfaces" that specify the operations accessible to clients through that interface. A "class body" specifies code for implementing operations in the class interface.

inheritance: A class may inherit operations from "superclasses" and may have its operations inherited by "subclasses". An object of the class C created by the operation "C new" has C as its "base class" and may use operations defined in its base class as well as operations defined in superclasses. Inheritance from a single superclass is called single inheritance; inheritance from multiple superclasses is called multiple inheritance.

Inheritance is here defined narrowly as a mechanism for resource sharing in class hierarchies. In the literature the term is used loosely to denote a variety of other forms of hierarchical resource sharing. We will later define "delegation" as a more general class-independent term for dynamic hierarchical resource sharing.

The class of object-based languages includes Ada [DOD], Modula [Wi], CLU [LSAS], and Actor languages [Ag] but not Pascal, Algol, or Fortran. Ada's objects are realized by packages, Modula's objects are called modules, and CLU's objects are instances of clusters.

The class of object-oriented languages is narrower than the class of object-based languages, excluding languages like Ada, Modula, and CLU but including languages like Smalltalk and C++.

Ada is object-based but is not object-oriented according to our definition because its objects (packages) do not have a class (type). CLU's clusters are effectively classes since they serve as templates for creating instances and allow instances to be "first-class objects" in the sense that they can be assigned to variables, be passed as parameters, and be components of structures. But CLU does not have an inheritance relation for defining hierarchical relations between clusters, and is therefore not object-oriented.

In accordance with our approach of naming "interesting" language classes we call object-based languages which require every object to have a class "class-based" or "classical" languages.

class-based (classical) languages: An object-based language is class-based (classical) if every object has a class.

Class-based languages are a proper subset of object-based languages, while object-oriented languages are in turn a proper subset of class-based languages. Ada is an example of an object-based language that is not class-based while CLU is an example of a class-based language that is not object-oriented. This hierarchy of language classes is illustrated in Figure 1.

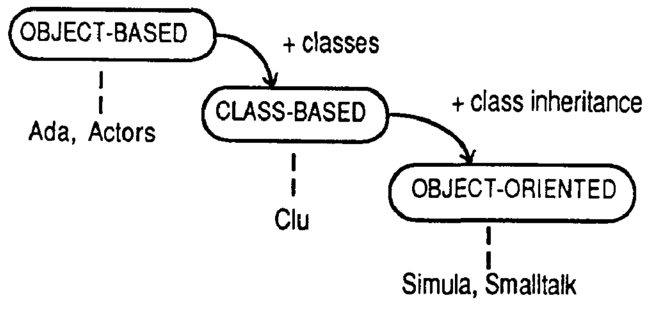

Figure 1: From Object-Based to Object-Oriented Languages

Figure 1 may be viewed as an inheritance hierarchy which uses object-oriented techniques to classify object-based languages. We may think of class-based languages as inheriting the attributes of object-based languages, and of object-oriented languages as in turn inheriting the attributes of both the class-based and the object-based languages. This self-application of object-oriented methodology to object-based languages both illustrates its general power in classifying and organizing knowledge and provides substantive insight into the particular domain which is the subject of this paper.

It is not surprising that object-oriented inheritance surfaces as a technique for defining the relation between language classes. Figure 1 classifies languages into hierarchies by imposing progressively stronger requirements on the features they possess. Such classification is precisely the purpose of inheritance in object-oriented systems, as pointed out in [We].

We briefly consider the impact of objects, classes, and inheritance on programming methodology. Objects serve to group operations with the data they will transform and provide a data-oriented principle for program design. Classes serve to manage collections of objects, allowing objects to be treated as first-class values within the language so that they can be passed as parameters, assigned as values of variables, and organized into structures. Class inheritance serves to organize collections of classes, allowing application domains to be described by class hierarchies.

Object-based languages like Ada support the functionality of objects. But object management must be handled by mechanisms outside the language like libraries, because it is not supported within the language. Class-based languages languages provide some degree of management of objects within the language but no mechanism for the management of classes. Object-oriented languages allow both objects and classes to be managed within the language, thereby providing a uniform mechanism for both design and implementation of applications. They are "wide spectrum languages" because they support both the high-level design of class hierarchies and the low-level implementation of objects.

3. Data Abstraction and Strong Typing

The terms "data abstraction" and "strong typing" may be defined as follows:

data abstraction: A data abstraction is an object whose state is accessible only through its operations. The state is generally represented by instance variables. Instance variables of a data abstraction are hidden from its clients and are accessible only through the object's operations.

strong typing: A language is strongly typed if type compatibility of all expressions representing values can be determined from the static program representation at compile time.

Object-oriented languages with data abstraction and strong typing are a narrower class of languages with stronger structuring properties than the class we have chosen to call object-oriented. This narrower class excludes Simula, whose objects are not data abstractions because their instance variables can be accessed by other objects, and Smalltalk, which is not strongly typed because its variables may be assigned values of different type at different points of execution. The term "object-oriented" has been carefully defined to be sufficiently narrow to exclude languages like Ada, Modula, and CLU and sufficiently broad to include languages like Simula and Smalltalk.

Object-oriented languages which have strong typing and require all objects to be data abstractions will be called strongly typed object-oriented languages:

strongly typed object-oriented language: An object-oriented language is strongly typed if it has strong typing and requires all objects to be data abstractions.

Strong typing and data abstraction have the common objective of strengthening object modularity but are independent in the sense that strong typing is possible for objects that are not data abstractions (as demonstrated by Simula), and data abstraction is possible without strong typing and indeed without any typing at all (as demonstrated by Smalltalk).

Should object-oriented languages require abstraction and strong typing? Smalltalk has consciously avoided strong typing in order to achieve dynamic binding, while Lisp-based object-oriented languages in the Flavors tradition [Mo,DG] have consciously avoided both strong typing and abstraction. Flavors goes even further and does not actually have objects as a language primitive. It has the more primitive notion of a data template to which operations may be attached. Flavors do not specify operations but may be used as an anchor for operations. Thus Flavors-style languages are not, strictly speaking, object-based but serve as a substrate which may be used in an object-based or object-oriented way.

The inclusion of data abstraction and strong typing is clearly not an unqualified benefit. It involves a tradeoff between structure and discipline on the one hand and flexibility and efficiency on the other. Abstraction is good when we can commit ourselves to particular abstractions early in the design process, but may be unduly constraining when we are unsure of the precise abstractions appropriate to a problem and wish to experiment with abstractions as part of the design and prototyping process. This is often the case with artificial intelligence applications or in other experimental applications concerned with understanding concepts that underlie a class of problems rather than with the solution of a specific problem. Lisp-based object-oriented systems are intended for such applications and consciously provide non-abstract objects to enhance the conceptual flexibility of problem solving.

Does it make sense to have non-abstract strongly typed languages? Although languages like Simula illustrate that commitment to strong typing is possible without commitment to abstraction, this may be a historical accident. It may well be that non-abstract objects are useful primarily for untyped formalisms where the absence of both abstractions and types encourages conceptual flexibility. Typed formalisms may discourage experimentation to such an extent that non-abstract objects are no longer useful. However, this is just speculation, and closer analysis might well reveal that non abstract strongly typed objects are in fact useful in certain kinds of experimental applications.

In spite of these reservations, the accepted wisdom is that strongly typed object-oriented languages should be the norm for application programming and especially for programming in the large. An object-oriented programming environment should probably support Lisp-style untyped programming for purposes of prototyping and strongly typed object-oriented languages for traditional application programming. Moreover, there should be provision for automatically freezing experimental prototype code to turn it into strongly typed code if and when it is ready to be used for production programming.

4. Consistency and Orthogonality

The consistency of a collection of language features may be defined as follows:

consistency: A collection of language features is consistent if they can coexist, that is, if there is a "model language" that realizes the features. Consistency can be demonstrated by exhibiting a language that has the set of features.

The five features that define strongly typed object-oriented languages are consistent since strongly typed languages exist, for example Owl [Sc]. Moreover, consistency of a set of features implies consistency of any subset of the features.

A collection of language features is orthogonal if no feature is a consequence of any of the other language features. This notion of independence is captured by the following condition:

orthogonality: A collection of features is orthogonal (independent) if, for every subset, there is a language that possesses that subset of features and no features in the complementary subset.

Objects, classes, and inheritance are far from orthogonal. Classes are defined in terms of objects and inheritance is in turn defined in terms of classes. We write this dependence as follows:

classes $\rightarrow$ objects
inheritance $\rightarrow$ classes

This lack of orthogonality suggests that we look for orthogonal concepts that define the essence of being a class in an object-independent way and the essence of inheritance in a way that is independent of classes and objects. In the case of classes the orthogonal concept is the notion of type which, as a first approximation, may be defined as follows:

type: A type is a behavior specification that may be used to generate instances having the behavior.

The orthogonality of objects and types is illustrated by Ada which has a well developed notion of type and even strong typing but does not support typed objects.

In order to define a notion of inheritance orthogonal to classes and objects the class-independent "essence" of inheritance must be identified. We view inheritance as a mechanism for sharing incrementally defined resources that internalizes shared resources, treating them as part of an extended self. Following Cook [Co], we define a class-independent form of inheritance in terms of a particular mechanism for self-reference that allows remotely defined operations to be internalized as part of the extended identity of an object. The class-independent notion of inheritance will be called delegation and may be defined as follows:

delegation: Delegation is a mechanism that allows objects to delegate responsibility for performing an operation or finding a value to one or more designated "ancestors". A key feature is that self-reference in an ancestor dynamically denotes the delegating object, thereby allowing the ancestor to be part of the extended identity of the delegating object. Dynamic binding of self-reference realizes sharing and reusability by allowing the resources of an ancestor to be part of the extended identity of different delegating objects at different points of execution.

Delegation is defined independently of classes. The key concept is the internalization of delegated operations so they can be treated as part of an extended self. Delegation is defined to be a form of resource sharing that allows shared resources to be viewed as belonging to the entity on behalf of which they are executed. This effect is realized by dynamically binding "self" to each entity on behalf of which it is executed for the duration of its execution. Thus a given operation can "belong" to different entities on different instances of execution.

Inheritance may be viewed as a specialization of delegation in which the entities that inherit are classes, and is therefore considered to be in the same "design dimension" as delegation. If we had to choose between these two notions to characterize this design dimension we would choose delegation, since it is purer and is "orthogonal" to other design dimensions. However, inheritance is more familiar and is needed to characterize object-oriented programming

Methodologically, orthogonality is a nice property of design dimensions that is useful for purposes of classification. When dimensions are not orthogonal it is often useful to go through the exercise of identifying what it takes to make them orthogonal, as we did in identifying the notion of delegation as an orthogonal form of the notion of inheritance.

How can we extend the design space determined by objects, types, and delegation to take account of data abstraction and strong typing? Data abstraction is not orthogonal to these dimensions because it depends on objects. The associated orthogonal notion is abstraction or information hiding defined so that it is uniformly applicable to any entity:

abstraction: An abstraction is a specification of an entity by an interface that controls access to the entity by other entities.

Strong typing is not orthogonal since it is a form of typing. There is no additional dimension because strong typing simply requires every value to have a type and that operator/operand compatibility can be determined at compile time. Thus strongly typed object-oriented languages can be characterized in a design space with the following four orthogonal dimensions:

objects — modular computing agents
types — expression classification mechanism
delegation — resource sharing mechanism
abstraction — interface specification mechanism

These dimensions provide a design framework for object-based languages in terms of computing agents, classification mechanisms, sharing mechanisms, and interface specification mechanisms. Specific languages in this design space are defined by constraints on these design dimensions such as the specialization of types to have classes and include strong typing, of delegation to be inheritance of classes, and of abstraction to be data abstraction. The constraints define a subspace of the design space determined by orthogonal design dimensions.

5. Design Alternatives for Delegation

Delegation may be specialized by selecting among the following design alternatives:

(1) classless delegation versus inheritance
Classless delegation realizes dynamic sharing in an instance hierarchy while while inheritance realizes dynamic sharing in a class hierarchy.

(2) strict versus non-strict inheritance
Strict inheritance requires descendants to be behaviorally compatible with ancestors, while non-strict inheritance allows operations of ancestors to be arbitrarily redefined and captures the notion of "similarity" rather than "behavioral compatibility. In between there are forms of controlled redefinition. In [WZ] we refer to strict behavioral compatibility as an "is-a" relation and to non-strict similarity as a "like" relation.

(3) single versus multiple inheritance
Multiple inheritance allows an object to inherit from multiple ancestors and provides a

more flexible behavior composition mechanism than single inheritance. There is no agreement on the mechanisms of method combination that multiple inheritance should support.

(4) abstract interface versus code sharing
Should sharing by inheritance be at the level of code sharing or abstract interfaces? Smalltalk and Flavors view inheritance as a code sharing mechanism. CommonObjects [Sn] is based on abstract inheritance. Actra [LTP] supports both specification hierarchies based on abstract inheritance and implementation hierarchies based on code sharing.

Figure 2 demonstrates the hierarchical selection of design alternatives. It shows that the object-oriented classification method can be applied to progressively more finely grained design decisions to select among alternative delegation strategies.

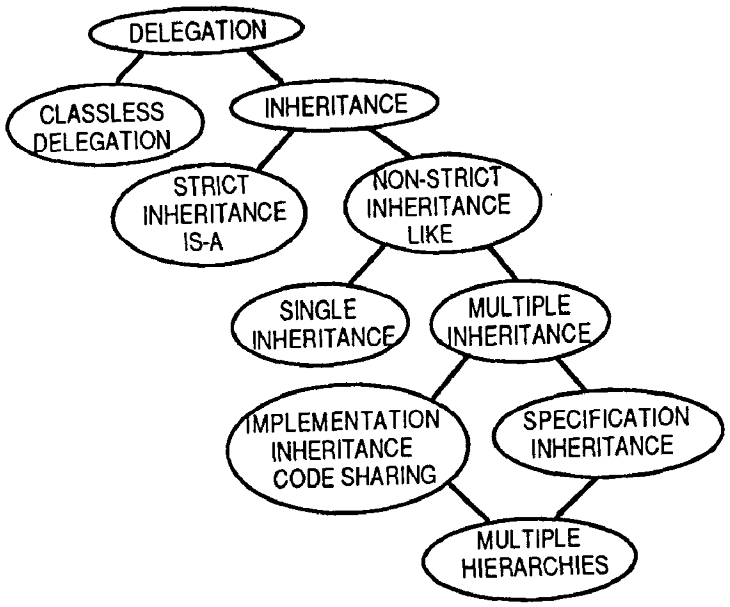

Figure 2. Design Alternatives for Inheritance

6. Delegation-Based Languages

Since classless object-based languages with delegation are interesting we christen them as follows:

delegation-based languages: A delegation-based language is an object-based language that supports classless objects and delegation:
delegation-based = objects - classes + delegation

Delegation was conceived by Lieberman [Li] and used by Cook [Co], Stein [St], Lalonde, Thomas, and Pugh [LTP], Hailpern and Nguyen [HN], and others in exploring classless models of inheritance.

Our definition of delegation differs from that of Lieberman [Li] in that we view inheritance as a special case of delegation while Lieberman views delegation and inheritance as two distinct mechanisms. We have taken the liberty of slightly redefining the term to focus on the class-independent essence of inheritance rather than on a concept that parallels inheritance. We could have used another term but the term "delegation" seems to capture the intuition of sharing by delegating responsibility without commitment to either classes or class-independence.

Delegation captures the essence of the dynamic resource-sharing paradigm underlying inheritance in a purer form than the earlier class-dependent definition. Features of dynamic hierarchical resource sharing are more clearly characterized by delegation than in terms of specialized notions of inheritance. We consider two such features, namely object autonomy and virtual operations.

The dynamic resource sharing provided by delegation has its costs in object autonomy. It is as though objects are connected to ancestors by an umbilical cord which they can never cut. Such dynamic sharing is expensive in distributed systems where ancestors are in different distributed components from their descendants. Delegation is an acceptable sharing mechanism for objects in a single address space but may be unacceptable as a sharing mechanism between address spaces.

The notion of virtual operations of Simula and other class-based languages may be defined more generally for delegation-based languages. Virtual operations arise when an ancestor specifies resources that will be implemented later in a descendant:

virtual resource (operation): A resource (operation) named and specified in an ancestor whose implementation will be provided by a descendant.

In class-based languages classes with virtual resources are called abstract classes and cannot be instantiated because there is generally no way to bind virtual to actual resources except at instance creation time.

abstract class: A class with virtual resources that can be instantiated only as an instance of a subclass for which virtual resources of the class are implemented.

However, there is no inherent reason why separately defined actual resources should not be connected to already created objects either at load time or dynamically as they are generated by the system. Specification and implementation should be sufficiently decoupled so that implementations can be bound to specifications in a flexible way.

7. Classless Languages

Objects not required to have a class will be called classless objects and languages with classless objects will be called classless languages:

classless objects: Objects that do not have a class.
classless languages: Object-based languages whose objects may be classless.

Classless languages represent a more radical form of typelessness than languages that are object-oriented but not strongly typed. The relation between Smalltalk and classless languages is similar to that between Snobol4 and Lisp. Values in Snobol4 have a type but allow variables to assume values of different types at different points of execution, while values in Lisp are untyped. Classlessness provides greater freedom for experimental programming than lack of strong typing but is correspondingly less structured. The arguments in favor of classlessness include the following:

(1) class-independent operations
It does not always make sense to associate an operation with a specific class, since:
a) an operation may have objects of several different classes as arguments, and transform the state of several different classes.
b) an operation may transform not only an object's state but also its interface.
c) an operation may be applicable to many different classes (friends in C++)

(2) classes with a singleton element
When classes have just a single instance (the class of all planets nearest the sun or of all successors of 0) the separate specification of shared behavior and a non-shared state makes no sense, since the shared behavior is shared by only one object. However, when such classes are formed by specialization (inheritance) from more general classes (the class of planets or integers) then classes with even a singleton element may be worthwhile.

(3) auxiliary entities
Classes are auxiliary entities having no necessary existence in the domain of discourse being modelled. The class hierarchy (Integer, Number, Magnitude, Object) used to model integers in Smalltalk is determined by an arbitrary design decision of Smalltalk rather than by a necessary property of the integers. We may in principle dispense with auxiliary abstractions and capture the properties of collections of objects directly by prototypes whose properties serve to specify both particular instances and defaults for dependent instances.

Languages with classless objects are object-based but not object-oriented. We can subdivide such languages into two categories:

classless languages without delegation
classless languages with delegation

We shall examine Actor languages as an example of classless languages without delegation, and Lieberman's prototypes [Li] as an example of classless languages with delegation.

8. Actors

Actor languages support objects, abstraction, and concurrency but not classes, inheritance, or strong typing:

actor languages =
objects + abstraction + concurrency
- classes - inheritance - strong typing

Actor languages are low-level languages that may be used to build higher-level, more structured languages. The concurrency supported by actors is fine-grained in the sense that actors not only execute concurrently with other actors but may also execute their internal actions concurrently. Actors represent a point in the design space of object-based languages very different from that of traditional object-oriented languages. They raise fundamental questions relating to the nature of concurrency that are beyond the scope of this paper. We are here interested in actor languages because of the clear and simple model they provide of objects without classes rather than as a basis for a practical programming language.

Actors are objects which have a mail address (mailbox name) and a behavior. The mail address designates a buffer which can store an unbounded linear sequence of messages (called "communications"). The behavior of an actor is defined by its actions in response to a communication. A pure actor can process just a single communication from its mailbox before it "dies". Computational actors, like the human ones described in Maçbeth, "strut and fret their hour upon the stage and then are no more".

The identity of an actor is determined by its mailbox name which is firmly decoupled from its state and behavior. Moreover, the state and behavior for an actor is totally independent of the state and behavior of its successor. The process of creating a successor is not unlike reincarnation in that the "soul", represented by the mailbox name, is reincarnated for an entirely new body.

An actor may respond to a communication by sending messages, creating new actors, and creating its replacement, as illustrated in Figure 3:

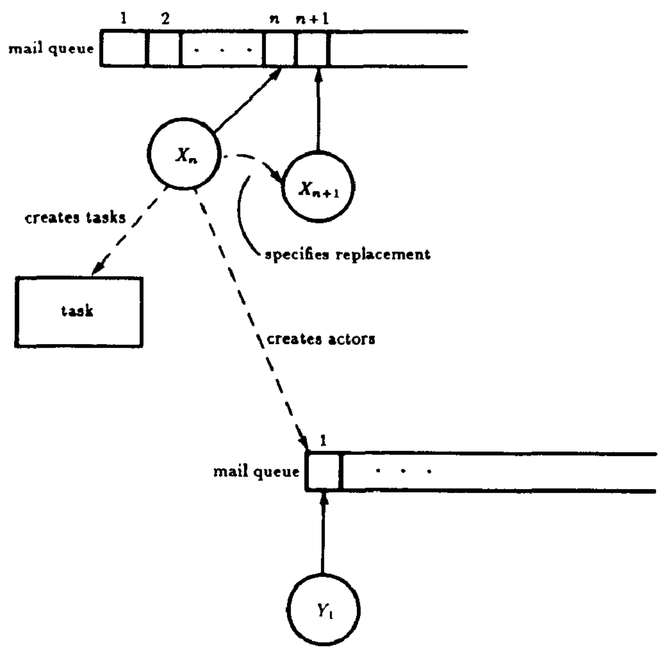

Figure 3: Behavior of Actors

(1) An actor may send a finite number of communications to other actors with known mailbox names (its acquaintances).

(2) It may create a finite number of new actors. The mailbox name of newly created names is known to the creating actor and may be disseminated to other actors by sending them communications that contain the mailbox name.

(3) It must designate a successor with the same identity (mailbox name) as its parent to process the next communication to the actor. The behavior of the successor is called the replacement behavior. There are no constraints on the relation between the behavior of an actor and its replacement behavior. In particular, the message set meaningful for an actor (corresponding to its set of operations) need bear no relation to the message set meaningful for its successor.

Supposing an actor sends p communications to other actors and creates q new actors as well as its successor before it dies. These p+q+1 activities are in principle concurrent, so that the processing of a single communication spawns p+q+1 concurrent tasks. The successor actor occupies a special place among these since it represents the continuation of the main process. The successor may initiate its activity by responding to the next communication in the mailbox as soon as it is fully created, and may execute concurrently while its progenitor is completing its other tasks. This permits pipelining, that is, concurrently applying the sequence of incarnations of a given actor to a sequence of communications in its mailbox.

The actor associated with a mail queue has a sequence of incarnations with Xn representing the incarnation that processes the nth communication. Xn must wait until its communication (in slot n of the mailbox) arrives. It then creates a replacement behavior X(n+1) which processes the (n+1)th communication, and may send communications to other actors and create new actors with associated new mail queues.

Following [Ag] we briefly show how the factorial function is computed in the actor formalism. We define a factórial actor which responds to messages of the form (n,r), where n is the integer whose factorial is being computed and r is the mailbox to which the result will be sent. The factorial actor has a simple response when n=0 and a more complex response when n>0.

When n=0 the factorial actor simply sends a message with value 1 to the actor with mailbox address r. When n>0 it performs the following actions:

(1) It creates a replacement behavior Fn that has the same behavior as the original factorial actor.

(2) It creates an actor, say "An", which computes "n*k" on receiving the message k and sends the result to r.

(3) It sends a message to "self" consisting of the integer "n-1" and the return address of the newly created actor "An". If $(n-1) > 0$ this in turn causes its replacement behavior to
 1) create an actor "A(n-1)" which, on receiving the message k, computes "(n-1)*k", and
 2) send the message with integer (n-2) and return address "A(n-1)" to self.

Thus the message "(n,r)" sent to the factorial actor will create a sequence of n incarnations of the factorial actor Fn, F(n-1), ..., F1, and an associated set of created actors An, A(n-1), ..., A1 such that Ai will receive a message with value (i-1) factorial and send i factorial to A(i+1). Finally, "An" will receive (n-1) factorial and send n factorial to r.

This example does not demonstrate concurrency since the computation takes 2*n sequential steps: n to create the auxiliary actors An,...,A1, and another n to perform the n multiplications. Using actors to compute factorials is like using a sledgehammer to crack a nut. But it does illustrate the role of replacement behavior, actor creation, and message creation in a concrete, albeit simple, computation. In doing so it provides insight into the reasons for requiring these three mechanisms as a basis for defining actor computation.

Actors provide a flexible model of computation based on a powerful computation primitive. The model is too powerful and flexible for most computations that arise in practice, and certainly too powerful for computing factorials. However, the actor model simply demonstrates how computations on classless objects may be realized and how the model may be specialized to class preserving computations, namely by constraining the replacement behavior.

9. Prototypes

Delegation-based languages allow objects to share and internalize operations of "ancestor" objects, called prototypes, that serve both as instances and as templates for descendants:

prototype: A prototype is an object that is both an instance and a template. Objects may delegate responsibility for performing operations or finding values to a prototype. The prototype provides defaults for its operations and values to objects that request the prototype to perform an operation on their behalf.

When classes that have only a single member we are assured that there will never be an occasion when the prototype needs to be used as a template, and it can play the role of an instance without worrying about the effect of changes on delegating objects. Even when classes have many members prototypes are a natural way of representing the first member of the class that is encountered but may cause problems when additional members delegate their default behavior to the prototype.

For example, if we encounter an elephant, say Clyde, there is no need to store both the instance and its class and we may store just the instance. If we then encounter a second elephant, say Fred, we can view Clyde as a prototype for representing knowledge about Fred. Fred may be represented by his differences from Clyde. Properties that Fred shares with Clyde can be omitted from the representation of Fred since the default values in Fred's prototype Clyde may be used.

In this example, prototypes require less overhead than the alternative of creating an elephant abstraction independent of Clyde and Fred. They also appear to model the cognitive acquisition of knowledge about elephants more naturally. It is only after seeing many elephants that an elephant abstraction becomes cognitively established and practically useful. The prototype mechanism appears to model knowledge acquisition more closely than the class mechanism for human cognitive processes, and its computational models have less overhead for classes with a small number of instances. However, classes model cognitive processes and knowledge organization of the specialist so that both mechanisms are needed to span the complete range of cognitive situations.

We shall refer to delegation-based languages based on prototypes as "prototypical":

prototypical languages: Prototypical languages are delegation-based languages that realize delegation by prototypes.

The transition from a prototypical to a classical representation of objects could in principle be performed automatically. It reflects the leap in abstraction that comes from recognizing the common structure of a collection of instances and defining a class that captures this structure. It is performed repeatedly in childhood in learning the meanings of words like cookie, dog, and table. In computation, unification is an example of an automatic method of finding the common structure of a collection of patterns. A similar technique could be developed for finding the common class structure of a collection of objects. Moreover, the commonality assumptions could be expressed as constraints on variability and the "unification" technique could be reapplied whenever variability constraints were violated.

The distinction between classical and prototypical systems reflects a long-standing philosophical debate concerning the status and representation of abstraction. Plato viewed abstractions like "ideal" tables as having an existence more real than instances of tables in the real world. Object-oriented languages like Smalltalk are Platonic in their explicit use of classes to represent similarity among collections of objects.

The alternative view, that abstractions are unnecessary auxiliary constructs, has not been propounded as cogently, probably because, taken to the extreme, it may simply be wrong. While any given set of auxiliary entities may be unnecessary in the description of a domain of discourse, the stronger position that complex domains should be described without any auxiliary entities whatsoever seems untenable.

Prototypical systems are adequate as a primitive substrate for organizing domains of discourse, just as untyped computational formalisms such as assembly languages or the lambda calculus are adequate for expressing all possible computations. However, when a prototypical system or untyped formalism is used to model a complex universe, types and classes for expressing regularities in the domain creep in by the back door, and it becomes preferable to introduce explicit typing and classification schemes rather than rely on ad hoc ingenuity. It may well be appropriate to adopt a prototyping view of the world in the early stages of modelling a domain and to switch to a typed view when the classes appropriate to the domain become established. In switching from an untyped to a typed model we give up some flexibility in the interests of structure and regularity.

Classical and prototypical languages have different approaches to the knowledge representation of shared abstractions. Classical languages distinguish between two sharing mechanisms: sharing of class attributes by instances and sharing of superclass attributes by classes. In contrast, prototypical languages have just one kind of sharing, namely the sharing by instances (which may be prototypes) of default properties defined in their prototypes.

Delegation may be used for managing shared information represented by prototypes. However, our definition of delegation is broader than Lieberman's and may be used also for managing shared information in classical inheritance.

10. Object-Based Concurrency

Concurrent object-based languages model the world by concurrently executable objects called processes. The term "process" is used in the context of operating systems to mean a machine language representation of a computation that is performed on a processor. We use the term in the context of programming languages to capture the higher-level notion of a concurrently executing object:

process: Processes have an interface of executable operations or entry points and one or more threads of control that may be active or suspended.

Process-based languages are object-based languages whose objects may execute concurrently.

process-based language: A process-based language is an object-based language that has processes.

Object-based languages model the world by autonomous objects that are constrained to execute sequentially. Process-based languages extend the autonomy of objects to autonomy in time.

The primitive executable processing elements within a process are called threads:

thread: A thread consists of a thread control block containing a locus of control and a stack which represents its "state" of execution and is initially empty.

Threads are data structures that can become active by being loaded into a processor. Thread data structures may be passed as message requests to processes, and may be queued in message buffers until a process is ready to execute them. They may be suspended if conditions required for their execution are not appropriate and reactivated when the conditions again obtain. We classify processes in terms of the properties of their threads.

We distinguish between sequential processes with a single thread of control, quasi-concurrent processes with at most one active thread of control, and concurrent processes with multiple threads of control.

sequential process: A process that has just one thread of control.
quasi-concurrent process: A process that has at most one active thread of control.
concurrent process: A process that may have multiple active threads of control.

Sequential processes (Ada and Nil) generally have a body with an interface of entry points at which messages to perform operations may be queued. An invoking operation (incoming message) must wait until the already executing process is ready to accept it by means of a "rendezvous" which joins the incoming and active threads of control for purposes of synchronization and argument communication and then separates the threads so that invoking and invoked processes may again proceed in parallel.

Quasi concurrent processes allow threads of control to be suspended while waiting for a condition to be fulfilled and resumed when the condition is satisfied. They differ from sequential processes in having "condition queues" of suspended threads as well as entry queues of threads that are waiting to enter the process. An incoming thread can become active only if the current thread terminates or is suspended, or if the incoming thread fuses with the active thread by a mechanism such as rendezvous. Monitors [Ho, Ha] are an example of quasi-concurrent processes.

In concurrent processes there is no restriction on active threads and an invoking operation may freely create a new thread. But attempts to access shared data in critical regions (atomic objects in Argus) may cause a thread to be suspended until the shared data can safely be accessed. Concurrency within processes allows finer-grained control that permits suspension to be delayed from process entry time to the time of entry to critical regions.

The concurrent languages CSP [Ho1], Ada, and Nil [SY] have sequential processes. Monitor-based languages like DP [Ha], ABCL/1 [YBS], and Orient 84/K [IT] have quasi-concurrent processes. Actor languages and Argus [Li] have concurrent processes.

Note that all three language classes are fully concurrent. They differ in their restriction on concurrency within processes but are similar in placing no restriction on concurrency between processes.

Restrictions on concurrency between processes are in fact useful in defining weaker (subconcurrent) languages that allow multiple independent but not concurrent threads. For example, Simula with its coroutines and Smalltalk 80 with its "processes" are "quasi-concurrent languages" because they allow objects to have independent threads of control but allow only one thread to execute at a time:

sequential language: A languages with a single thread of control.
quasi-concurrent language: A language with multiple independent threads but only one active thread.
concurrent language: A language with multiple active threads.

In this section we are committed to full concurrency at the language level and focus on design alternatives for concurrency within processes. The question whether processes should have internal concurrency can be addressed at the level of both conceptual modelling and language design. At the conceptual level some applications are more naturally modelled by sequential or quasi-concurrent processes while others are more naturally modelled by fully concurrent processes.

At the design and implementation levels sequential and quasi-concurrent processes allow the unit of modularity and concurrency to be the same and result in much simpler languages than concurrent processes. Concurrent processes permit units of modularity to contain multiple units of concurrency and require distinct synchronization and communication mechanisms for inter and intra process concurrency at both the language and system levels [SYW].

However, concurrent processes are more uniform in permitting the same concurrency primitives to be used both within and between processes. They have a hierarchical rather than a flat process structure. Moreover, concurrent processes permit more finely grained concurrency and are more expressive in modelling situations in the real world which require such concurrency.

The step from sequential to quasi-concurrent processes makes scheduling of threads within a process more flexible without causing mutual exclusion problems for simple access to data structures. However, quasi-concurrent processes present mutual exclusion problems when processing transactions because suspending a thread in the middle of a transaction could cause integrity constraints of the transaction to be violated.

Transactions may be viewed as "temporal modules" in the sense that they represent uninterruptible non-atomic temporal units of execution. Quasi concurrent processes present no mutual exclusion problems for atomic operations but cause problems when we try to combine the temporal modularity of transactions with the traditional spatial modularity of objects and processes. Concurrent languages based on quasi-concurrent processes, like ABCL/1 or Orient 84K, are harder to extend to transaction processing than languages based on concurrent sequential processes. Thus there is a tradeoff between flexibility and extensibility in replacing sequential by quasi-concurrent processes.

Concurrent object-oriented systems must be able to handle transactions and must therefore deal with temporal modularity (atomic actions) as well as spatial modularity (atomic objects). Mechanisms for transaction-based concurrency control have been reviewed in [BG]. The carefully crafted concurrency control mechanisms of the Argus system are decribed in [LS]. A model for nested transactions in terms of input/output automata is presented in [LM].

11. Distributed Processes

Is concurrency consistent with and orthogonal to the design dimensions of sequential object-based programming? This question has a simpler answer for orthogonality than for consistency. Concurrency in its general form is clearly orthogonal to other design dimensions. However, in the context of object-based programming we are concerned with concurrently executing objects. Processes specialize the notion of concurrency in the direction of object-based languages. They implement the notion of concurrency in a particular way and determine a value or range of values in the dimension of concurrency. Thus concurrency is an orthogonal dimension of language design and processes are a specialization of that dimension.

Concurrency is a consistent extension of sequential object-based programming, since actors provide an existence proof of concurrent object-based programming. However, there is a potential conflict between the independence required for concurrency and the structured sharing required for inheritance. This is particularly true when concurrency is augmented by the stronger requirement that processes be distributed:

distributed process: A distributed process is a process with a separate address space, that is, it cannot directly access any resources outside its local address space and can communicate with the outside world only by message passing.

Distribution increases the autonomy of processes but makes it expensive to share nonlocal resources by mechanisms such as inheritance or delegation. In fact, we can say that distribution is inconsistent with inheritance. This explains why there are no languages with distributed processes that support inheritance.

The inconsistency between distribution and inheritance arises because the goals of modularity and sharing are incompatible. Modularity requires strong separation between components of a system while sharing requires fusion of components. Dynamic sharing requires fusion of components during execution and is incompatible with distribution which requires execution time separation.

Design alternatives for distributed processes involve interaction between the units of modularity, concurrency, and name space.

unit of modularity:
Unit that defines the user interface
unit of concurrency:
Unit that represents a single thread
unit of naming: Unit that determines name space

Processes for which the unit of modularity, concurrency, and naming are the same are called distributed sequential processes.

distributed sequential processes: A distributed sequential process is a distributed process with its own name space.

Distributed sequential processes are aesthetically appealing because the interface for message passing, mutual exclusion, and transactions can be identified. They are a basis for the process model of NIL [Str]. But this clean identification of interface, concurrency, and name space comes at a cost of conceptual flexibility and efficiency. Conceptual flexibility is sacrificed because the unit of sharing must have the same granularity as the unit of modularity and concurrency so that sharing among modules or concurrent units is precluded. Efficiency is sacrificed because of the high cost of making the transition between distributed components.

Two distributed processes A, B may share a third process C if ports of A and B are both connected to a port of C. Such sharing is at the level of abstract interfaces. Inheritance of abstract interfaces [Sn] is in principle possible for distributed processes, although it is a good deal more expensive than in shared memory.

An important dichotomy in distributed systems is that between static and dynamic interconnection:

statically interconnected distributed processes: The connections of each process to its environment is determined at process creation time and cannot be changed during the subsequent lifetime of the process.

dynamically interconnected distributed processes: The connections of a process to its environment can be changed by language commands during process execution.

Ports in dynamically interconnected distributed processes are variables to which process connections (sometimes called channels) can be assigned. It is prudent to associate types with ports and to permit connection only if the type and input/output mode of port values are compatible with that of the port variable to which it is assigned. Input ports may be thought of as sockets and output ports as plugs that must fit the sockets. Dynamically interconnected distributed processes may be modelled by a plugboard with wires corresponding to channels of communication.

12. Object-Oriented Persistence

Persistence is a property of data that determines how long it should be kept. In traditional languages the lifetime of data generally does not transcend the lifetime of a particular program. Some data, such as locally declared data or procedure parameters, have an even shorter lifetime. Databases store data whose persistence transcends that of individual programs. Adding persistence to an object-oriented language allows it to be used as a basis for database implementation.

Objects provide a better starting point for databases than procedures since their state persists between the execution of operations. They provide a more flexible way of organizing data than relations in a relational database. The class declarations of object-oriented languages can serve as a data definition language for databases. However, an object-oriented language by itself is insufficient to realize an object-oriented database.

A database may be viewed as a long-lived object or process with special properties. It is globally accessible (sharable) by a large number of users. Generally access is asynchronous from the point of view of the user, and we may think of the database as a non terminating process that services asynchronous user requests. Asynchronous access may be handled either directly by the database process or by a database server that organizes user requests and feeds them to the database. The database itself may be a sequential process (dealing with requests in a serial order), a quasi concurrent process, or a fully concurrent process with locks that enforce mutual exclusion for data access.

Some of the special features of database processes are enumerated below:

(1) To support persistence we need a strong notion of object identity that is independent of the key used in object selection and persists across programs and projects.

(2) We need a query language that can process traditional database queries (such as finding the set of all employees that make more than their managers). This kind of query may involve objects of more than one type and and produces results that are collection of objects. Queries in relational database languages may be viewed as "select" operations on an aggregate type, namely the type "set" or "relation". They have the form:
select(set, predicate)
Query complexity and efficiency are determined by the nature of the predicate. Relational query languages specify all queries in terms of a restricted set of relational query primitives whose optimization has been extensively studied. Object-oriented query languages must accommodate the greater richness of object-oriented specifications for which optimization is not as well understood. One of the issues in object-oriented query languages is to make them efficient, so that the user does not pay in terms of efficiency for the flexibility provided by object-oriented programming.

(3) Since object-oriented databases are particularly suited to the management of evolutionary systems they require a mechanism for version control and other tools for evolving systems.

(4) Databases should be able to specify constraints and check that constraints are not violated as the database is modified. This may be achieved by active variables of triggers [ZW].

(5) Multiple views should be supported with automatic updating of all views when the data is modified. Lazy updating for views that are not currently active is clearly appropriate.

Databases must support transaction processing and concurrency control so that user requests can be processed in a safe but efficient manner. The level of safety and resilience in the face of software and hardware failures must be much greater than for traditional programs. Facilities for aborting transactions and for failure recovery must be provided. Type dependent concurrency control [Wei] could considerably increase the efficiency of object-based database transactions over corresponding relational transactions.

13. Conclusions

Persistence is orthogonal to concurrency and to other dimensions for object-oriented language design. We therefore have identified a total of six orthogonal dimensions of object-oriented language design:

objects
types
delegation
abstraction
concurrency
persistence

This list of dimensions is not in itself surprising and could easily have been generated on the fly. Our contribution has been to relate these dimensions to non orthogonal concepts in real programming languages, to examine design tradeoffs and interrelations between instances, and to identify and interesting points in the object-oriented design space.

The study of languages as points in a design space and of language classes as classes of an object-oriented hierarchy is a novel approach to the analysis of programming languages. We have tried to demonstrate that this approach is worthwhile when there are sufficient points in the design space that meaningful comparisons can be made.

There are many dependence relations and design alternatives we have not considered, and our examples only scratch the surface. But they do demonstrate the potential of exploring dependence among design dimensions of a language design space, and illustrate that the use of object-oriented classification techniques to structure the design space simplify the presentation of design alternatives and help us to better understand the complex issues involved.

Acknowledgements: Discussions with Tom Doeppner and Stan Zdonik helped to clarify the ideas in the sections on concurrency and persistence. This research was supported in part by the National Science Foundation, IBM Research, ONR contract N00014-83-K146, and DARPA order No 4786.

14. References

[Ag] Agha G., Actors, A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[BG] Bernstein B. A. and Goodman N., Concurrency Control in Distributed Database Systems, Computing Surveys, June 1981.

[Co] Cook William, Self-Referential Models of Inheritance, Brown University Report, March 1987.

[DG] DeMichel L. G. and Gabriel R. P., The Common Lisp Object System, Proc ECOOP 1987.

[Do] Doeppner T. W., Threads - A System for the Support of Concurrent Programming, Brown University Computer Science Tech Report CS-87-11, June 1987.

[DOD] Ada Reference Manual, US Dept of Defense, July 1980.

[GR] Goldberg A. and Robson D., Smalltalk 80: The Language and Its Implementation, Addison-Wesley 1983.

[Ha] Hansen P. B., Distributed Processes, A Concurrent Programming Concept, CACM 1978.

[HN] Hailpern B. and Nguyen V., A Generalized Object Model, In Research Directions in Object-Oriented Programming, Eds. Shriver and Wegner, MIT Press, 1987.

[Ho] Hoare C. A. R., Monitors, An Operating System Structuring Concept, CACM, October 1974.

[Hol] Hoare C. A. R., Communicating Sequential Processes, CACM, August 1978.

[IT] Ishikawa Y. and Tokoro M., Orient 84K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, In Object-Oriented Concurrent Programming, Eds Yonezawa and Tokoro, MIT Press 1987.

[Li] Lieberman H., Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages, OOPSLA 86.

[LM] Lynch N. and Merritt M., Introduction to the Theory of Nested Transactions, MIT/LCS/TR-367, July 1986.

[LS] Liskov B. and Scheifler R., Guardians and Actions, Linguistic Support for Robust Distributed Programs, TOPLAS, July 1983.

[LSAS] Liskov B., Snyder A., Atkinson R., and Schaffert C., Abstraction Mechanisms in CLU, CACM, August 1977.

[LTP] Lalonde W. R., Thomas D. A., and Pugh J. R., An Exemplar-Based Smalltalk, OOPSLA 86.

[Mo] Moon D., Object-Oriented Programming with Flavors, OOPSLA 86.

[Sc] Schaffert C. et al., An Introduction to Trellis/Owl, OOPSLA 86.

[Sn] Snyder A., Encapsulation and Inheritance in Object-Oriented Languages, OOPSLA 86.

[St] Stein Lynn, Delegation is Inheritance, OOPSLA 87.

[Str] Strom R., A Comparison of the Object-Oriented and Process Paradigms, SIGPLAN Notices, October 1986.

[SY] Strom R. and Yemini S., NIL: An Integrated Language and System for Distributed Programming, Proc SIGPLAN '83 Symposium on Language Issues in Software Systems, June 1983.

[SYW] Strom R., Yemini S., and Wegner P., Viewing Ada from a Process Model Perspective, International Ada Conference, Paris, May 1985.

[We] Wegner P., The Object-Oriented Classification Paradigm, in Research Directions in Object-Oriented Programming, Eds Shriver and Wegner, MIT Press 1987.

[Wei] Weihl W. E., Specification and Implementation of Atomic Data Types, PhD Thesis, MIT March 1984.

[Wi] Wirth N., Programming in Modula 2, Springer Verlag 1982.

[WZ] Wegner P. and Zdonik S., Why Like Isn't Like Is-a, Brown University Technical Report, April 1984.

[YBS] Yonezawa A., Briot J. and Shibayama E., Tokyo Institute of Technology, OOPSLA 1986.

[ZW] Zdonik S., and Wegner P., Language and Methodology for Object-Oriented Databases, Hawaii Conference on System Sciences, Jan 1986.