Beyond Objects: Objects *

Dennis Tsichritzis, Oscar Nierstrasz and Simon Gibbs†

Abstract

Object-orientation offers more than just objects, classes and inheritance as means to structure applications. It is an approach to application development in which software systems can be constructed by composing and refining pre-designed, plug-compatible software components. But for this approach to be successfully applied, programming languages must provide better support for component specification and software composition, the software development life-cycle must separate the issues of generic component design and reuse from that of constructing applications to meet specific requirements, and, more generally, the way we develop, manage, exchange and market software must adapt to better support large-scale reuse for software communities. In this paper we shall explore these themes and we will highlight a number of key research directions and open problems to be explored as steps towards improving the effectiveness of object technology.

*.In *International Journal of Intelligent and Cooperative Information Systems* (IJICIS), Vol. 1, No. 1., March 1992, pp. 43-60.

†.*Authors' address:* Centre Universitaire d'Informatique, University of Geneva, 24, rue du Général-Dufour, CH-1211 Geneva 4, Switzerland. *E-mail:* {dt,oscar,simon}@cui.unige.ch. *Tel:* +41 (22) 705.71.11. *Fax:* +41 (22) 320.29.27.

1 Introduction

With the advent of object-oriented programming languages (OOPLs), there has been the gradual realization that these languages alone are not enough to have a major impact on the software industry, but that more fundamentally our attitudes to software development must adapt to an object-oriented view. "Object-orientation" is not a technique. It is an *approach* which can potentially change the way we build, think and use computer systems.

The object-oriented approach is open-ended. It is not a closed set of ideas. The object-oriented approach is not revolutionary in terms of ideas. Most of its basic principles have been well-known for decades. It starts with some basic notions fairly well accepted in computer science, namely encapsulation and reuse. It is only revolutionary in that, if applied vigorously, it changes software development from a custom-made boutique to a *component-oriented* industry.

It has been commonplace to say that, within a few years, we will no longer speak about "object-oriented programming" any more than we do now about structured programming —- it will have been completely absorbed into the programming culture. However, object-orientation goes beyond structured programming, or even programming in general. What is truly *different* is not that we can bundle data and operations, or that we can define new classes from old by inheritance, or even that dynamic binding and polymorphism enhance the reuse potential for software. Rather, what is different is that, in order for an object-oriented approach to be applied effectively, we must change the way we *think* about software development as a labour-intensive activity in which each application is hand-crafted from individual requirements, to a capital-intensive one in which we invest more in the development of reusable components that can be plugged together to produce standard "lines" of applications. The things that change, then, are our software life-cycle, the roles that software engineers play, the way we manage software components, designs and descriptions, and the way we manage projects. In short, the software *culture* becomes object-oriented.

We will outline a number of different areas in which the object-oriented approach is applied, and we shall offer some views on the directions object-orientation is taking them. In the following section we will outline how object-orientation is changing programming languages specifically and programming in general. In the third section we will outline how object-orientation changes the development cycle of software. In the fourth section we will outline how object-orientation changes the way software is treated as a product. Finally we end with some concluding remarks on new ideas and directions.

2 Object-Oriented Programming Languages

The first object-oriented programming language was Simula [3]. Simula is essentially a dialect of Algol that introduces objects, classes and inheritance, mainly as mechanisms for modeling real-world objects for the express purpose of programming simulations.

Smalltalk was the first attempt to develop a "fully object-oriented" programming language, that is, one in which "everything" is an object. In particular, all primitive data types, such as numbers and truth values, statements (blocks), and even classes, are themselves objects. The fundamental contribution of the Smalltalk project was to recognize that OOP is not just a technique for programming simulations, but that it can also serve well as a general approach for software development. To achieve this goal, however, it is important to acknowledge two principles:

- *Object-oriented features are not just an "add-on" to existing languages:* to achieve the maximum benefit of these features, one must either design an OOPL "from the ground up" to be fully object-oriented, or one must take great pains to ensure that the added features be consistent with the existing language design.
- *The language is not enough:* just by switching to an OOPL, one's productivity is not necessarily improved. A reusable class library, a browser, an interactive programming environment, and a development methodology that encourages collaboration, reuse and evolution, are all

important ingredients to the successful application of OO technology.

In short, to be effective as a software development approach, OOP features must be well-integrated with the programming language and environment. In section three we shall see what this means for application development in general. Let us first consider some of the technical problems inherent in the integration requirement.

Wegner [38] [39] has proposed an OOPL classification scheme including several *dimensions* of object-based language design. The term "object-based" is introduced to distinguish languages that merely provide some mechanism for encapsulating data and operations (such as Ada) from those that also provide classes and inheritance (such Simula and Smalltalk). (In this scheme, languages such as Self [36], which are based on prototypes rather than classes, are not strictly considered to be object-oriented [39].) We have refined the classification scheme to include "fully" object-oriented languages:

Object-based languages allow one to encapsulate *objects*;

Class-based languages support the instantiation of objects from *classes*;

Object-oriented languages are class-based languages in which new (sub-)classes can be derived from existing classes by *inheritance*;

Fully object-oriented languages are homogeneous, that is, "everything" is an object, in particular, both primitive data types (like integers) and classes are objects;

Strongly-typed OOPLs provide abstract interfaces (types) for all objects, and guarantee that all expressions are type-consistent;

Concurrent OOPLs provide support for multiple concurrent threads of control (objects may or may not be physically distributed);

Persistent OOPLs support objects that may outlive the process that creates them; these languages are closely related to database programming languages and may also support, for example, transactions and querying over sets of such objects.

A naive view of these various dimensions might lead one to believe that they are all orthogonal, and that one might easily design a language to support them all by considering each aspect independently. In fact, the case is very much the opposite. For every additional dimension, one can point to fundamental conflicts in the object paradigm that must be resolved in order to achieve a clean integration. In each case, the source of the conflict is an incomplete application of the object paradigm. By paying careful attention to the role and function of object-oriented features and by applying them uniformly, we claim that these conflicts can be resolved. We will mention some of these conflicts and discuss the ways they can be resolved.

The encapsulation/inheritance conflict

Should subclass methods have access to private instance variables and operations inherited from superclasses? Should the fact that the class of an object inherits from some superclass be visible to clients of the object?

These problems become critical when the implementation of a class evolves during development and subclass instances are subsequently "broken" because inherited features have been changed. In this sense, inheritance "violates encapsulation" because the designer of a reused class no longer has complete freedom in revising the implementation [32]. The simple solution, on the other hand, of restricting subclass methods to access only the public interface of a superclass, forces one to make public all instance variables to be shared by various subclasses, even if they are not intended to be seen by clients.

The source of the problem is that classes have two kinds of clients, and a single interface is not sufficient to describe the different kinds of contracts between them. Classes are semantically overloaded as templates that can be used either to instantiate objects or to derive new sub-classes. Only by clearly separating these two roles and making explicit the contract between the two kinds of client can the conflict be resolved. There are several possible ways to achieve this:

- Explicitly declare which inherited features are visible to subclasses;
- Provide mechanisms to extend inherited behaviour in a more controlled fashion, such as the *inner* construct of Simula and Beta that indicates where subclass code may be inserted in inherited methods [3] [19];
- Instead of inheritance, adopt a compositional approach in which the dependencies of class templates (or *habitats* [31]) are made explicit.

Whatever approach is chosen, the key point is that the object paradigm works best when the services and obligations are clearly stated for different kinds of clients.

The first-class/second-class conflict

Are classes objects? Are messages objects? Is it possible to inherit from the "primitive" data types, like numbers and Booleans?

In some languages, like Ada, the notion of an object may be present, but there is no way to create multiple instances from a class template. In other languages, like C++, it is possible to inherit from user-defined classes, but not from built-in datatypes, like integers. In many object-oriented languages, classes are not themselves objects, so it is not possible to create new classes at run-time. Yet other languages introduce special kinds of manipulable entities, like locks, or message queues, but treat these as "second-class" objects which do not belong to any object class and can only be used in a very restricted way.

Generally these restrictions are made for pragmatic reasons —- if classes are objects, then the compiler must be part of the run-time environment; if built-ins like integers are objects, then a special implementation strategy must be used for subclasses of the class integer —- and so on.

Smalltalk has largely demonstrated the clear advantages of a fully object-oriented language: a homogeneous object model encourages reuse and design for reuse, whereas a heterogeneous model with many different kinds of entities frustrates abstraction and may force what is essentially the same algorithm or behaviour to be implemented many times for different special cases. More recently, projects such as Self [5] have demon-

strated that the run-time overhead traditionally associated with Smalltalk-like languages can be reduced to insignificant levels.

As a consequence we can only conclude that the conceptual and practical benefits of homogeneous object models overwhelmingly outweigh any advantages that might be gained by having separate, second-class objects.

The specification/implementation conflict

Is a class a type? Are subclasses subtypes? Can we perform type-checking on the basis of the inheritance hierarchy?

It is tempting to view the two concepts as one and to attempt to unify them in a programming language, as is done in Eiffel [23]. There are essentially two difficulties with this approach. First, it is possible to define classes that are type compatible but unrelated through the class hierarchy. Second, a subclass may introduce changes to the interface inherited from a superclass that may not be type compatible. Methods may be dropped, their names may be changed, or, more subtly, the arguments to methods may be refined, thus invalidating substitutability of subclass instances for superclass instances. In fact, many of these problems are apparent in Eiffel [8].

The source of the problem is that types and classes serve two different purposes and these should not be confused. A class is a template for instantiating objects, whereas a type is a specification of the object/client contract. A subclass reuses, extends and modifies the interface and implementation of its superclasses. A subtype is a strictly stronger specification of the object/client contract than that of any of its supertypes. In a sense, classes represent reuse for the implementer whereas types represent reuse for the user [21].

In some languages, such as POOL-I [2], these two concepts are explicitly separated.

The active/passive conflict

In the presence of concurrency, should objects be viewed as active agents with their own threads of control, or as passive entities shared by concurrent threads? Is there the need for a mix of both active and passive objects?

Since most objects in any given application are essentially passive in nature, it is tempting to conclude that all objects should be passive, and a separate "process" concept should be provided for defining new threads of control, as is the case in Smalltalk. Passive objects, however, have the disadvantage that it is up to the clients to ensure that concurrent requests do not interfere. This responsibility belongs with the object rather than with the client, since only the object can know when it is in a consistent state and ready to handle new requests. An active object is one that synchronizes requests with its internal state and any threads that it may currently be running.

Some languages provide for both active and passive objects, and protect passive objects by restricting their use to within single-threaded active objects. This approach leads to another manifestation of the first-class/second-class conflict, since we will find ourselves with two separate class hierarchies and redundant code for the two different kinds of objects. In general, a homogeneous model of active objects is more consistent with an object-oriented approach [26] [28] [29].

The other conflicts we have already mentioned are also further aggravated in a concurrent setting. Reuse through inheritance may be impaired if synchronization code is embedded in the methods of a class: subclass methods must violate encapsulation of the superclass methods in order to consistently extend the synchronization policy; worse, it may be necessary to reimplement superclass methods to cooperate with the extended synchronization policy of the subclass [18]. These observations have led several researchers to propose various ways of separating synchronization policies from the implementation of the method bodies in the form of declarative "synchronization predicates." Though there is general agreement that this is the correct approach to take, there is great variety in the proposed solutions, and no consensus has yet been reached. So both the encapsulation/inheritance conflict and the specification/implementation conflict are concerned.

The structural/behavioural conflict

How can we pose a query over a set of objects without referring to "hidden" instance variables? Are the "objects" of OOPLs and object-oriented databases really the same kind of object?

In databases we are typically more concerned with structure rather than with behaviour. We more often deal with sets than with instances and we are more concerned with protecting the integrity of the database as a whole rather than that of individual objects [34]. Object-oriented databases are in many ways an attempt to bring these two views closer together. Objects are neither pure structures nor are they just procedural interfaces to hidden data structures. They are behaviours that represent a set of services. These services may present themselves as operations to be invoked by message-passing, or they may be visible "attributes" representing some aspect of the abstract state of the object. Attributes may or may not be actual instance variables of an object. For example, a complex number may have several attributes, including its real and imaginary components and its polar coordinates. Some of these may be internally stored and others may be dynamically calculated. Any of them may be used to query over a set of complex numbers.

As with the other conflicts, the difficulty only arises if we fail to understand what it is that the object-oriented approach has to offer, and try to use it in an inappropriate way.

Let us sum up our observations. Classes encapsulate services. It is important to distinguish the various roles played and the contracts established with the different kinds of clients. It helps to have as homogeneous a model as possible, and not to introduce artificial distinctions between first and second class objects. Types are not classes, and it is doubtful whether there is anything to be gained by trying to make a single mechanism serve both purposes. In the presence of concurrency (and perhaps even in its absence), it is appropriate to view objects as active entities with complete control over acceptance of requests and their associated threads of control. Reuse through inheritance is encouraged if it is possible to separate the synchronization policies from the implementation of objects' methods. There is no "impedance mismatch" between programming language objects and database objects if we view visible attributes as part of the services provided by an object.

3 Object-Oriented Application Development

We mentioned at the start of the previous section that switching to an OOPL does not necessarily improve one's productivity. In fact, productivity is likely to drop if one does not also adopt an object-oriented approach to application development. Without adapting the development methodology, the best one can hope for is that new applications will be somewhat more naturally organized and responsibilities of subsystems will be more cleanly drawn, thus leading to a more robust and maintainable system.

What more can we hope for? Consider the following scenario for application development:

1. Starting with some very general application requirements, select a *generic application frame* (GAF) from a software information base (SIB). The GAF encapsulates for an application domain the domain knowledge, the requirements model, generic designs, and abstract and concrete classes (software components).
2. Guided by the GAF, you specify the specific requirements for your application and identify and refine relevant designs and software components.
3. Reusable components are bound, or *scripted* together to form a running application.
4. The application is monitored for correct behaviour and modified to adapt to evolving requirements.
5. Experience gained from the new application is used to refine the GAF.

Note that in this scenario, the role of object-oriented *programming* is minimized. Instead, we focus on reuse and software *composition*. Clearly, the place for OOP here is in the development of the software components, not in the development of the specific application. This, then, is a scenario for *capital-intensive*, as opposed to labour-intensive, software development [37].

In order to realize this scenario, however, there are a number of difficult questions to answer first:

- What are GAFs and where do they come from?
- How does one find GAFs and software components in a SIB?
- How are software components scripted?
- How do we make this scenario cost-effective?

A Component-Oriented Software Life-Cycle

A key assumption behind this scenario is that sufficient domain knowledge and experience have been gathered to make it possible to abstract from specific applications and encapsulate this knowledge in a GAF. The activity of developing a GAF we call *application engineering*. This activity is labour-intensive and requires expert knowledge.

The purpose of a GAF is to make the job of the *application developer* as easy as possible. Without a GAF, an application developer would, following some general object-oriented methodology, specify the application requirements, design the application architecture and implement the required object classes. The collected requirements, the domain knowledge, the design choices and the implemented components constitute a *specific application frame* (SAF). However a SAF typically contains both information specific to the requirements of the application under development as well as generic information concerning the application domain. What the GAF should do is streamline the development activity by fixing as many choices as possible relevant to the application domain, and make explicit those choices that remain. In effect, a GAF encapsulates a *specialized methodology* that is tailored to a domain.

GAFs may be very general or very specific. In the most general cases, only domain concepts and very general abstract designs might be provided. In the most specific cases, GAFs will be like recipes in cookbooks that can be followed step-by-step to produce a running application from available ingredients, or possibly used in a more creative way, by substituting ingredients or combining and adapting recipes to produce variations on standard themes. Furthermore, although we speak of GAFs as though they were disjoint, they will typically overlap, and one GAF may be just a refinement of a higher-level GAF. For example, a GAF for implementing accounting systems may be tailored to the needs of a particular corporation.

The essential difference between a component-oriented software life-cycle and a traditional one is that application engineering introduces design for reuse:

1. Starting from existing applications and domain knowledge, an application engineer factors out the requirements model, the generic designs and the reusable software components. This information is encapsulated as a GAF.
2. An application developer, guided by a GAF, specifies the requirements for a specific application and refines it to a SAF.
3. The SAF evolves according to changing application requirements.
4. The application engineer evaluates the quality of the GAF with respect to the ease with which the SAF was developed and the degree to which components could be reused without modification. The GAF is re-engineered to take into account this evaluation.

The key points to observe are:

- GAF development is iterative and evolutionary.
- Software components are not reusable in isolation, but only as part of a generic design.

The software life-cycle we have described has been elaborated as part of ITHACA [1] [12], an Esprit technology integration project to develop a complete object-oriented application development environment.

Software Information Management

We have mentioned that GAFs should be stored in a software information base. A SIB is a repository not just for object classes, but more generally for *software descriptions* pertaining to any aspect of application development [7] [12] [13]. There are two conceivable models for repositories:

- *The software junkyard:* in this scenario any and all software is stored and catalogued as it becomes available; re-

use is by querying, browsing and subsequent modification to meet new needs.

- *The software cookbook:* only software that is designed as part of a more general or more specific *recipe* is stored and catalogued; reuse is by locating and following recipes, sometimes by combining and adapting recipes.

It should be fairly clear that the second model is more consistent with the principle of a GAF and that it offers better potential for reuse to the developer. Of course, it depends on the fact that someone has taken the time and trouble to write the cookbook. To write the software cookbook in the first place it may be necessary to scrounge through a software junkyard, but this should not be the general model of reuse for developers.

We will discuss in more detail issues of object-oriented software management in section four.

Frameworks and Scripting

Applications can be constructed from prepackaged software components only if these components have been designed to be *plug-compatible* [10]. This means that standard interfaces and protocols must be defined, and that all plug-compatible components must conform to the same interfaces. Although this may seem a vacuous statement, it does entail some difficulties. First of all, nothing in any OOPL constrains a programmer to use only previously defined interfaces. It is too easy to extend interfaces by inheritance and define new protocols. Second, it is a non-trivial problem to define the *right* interfaces for a set of object classes. This is the *framework design problem*.

A *framework* [17] is a collection of abstract classes (i.e., templates for classes that define interfaces and the implementation of generic methods) designed to work together to solve a particular class of problems. Frameworks can take a long time to stabilize, and are the result of much experimentation and redesign. They are an integral part of GAFs, constituting the generic designs and components.

Once a framework has been defined and a suitably rich component set conforming to it has been implemented, one should be able to construct many applications without any further programming, but by *scripting* components together.

A *visual scripting tool* is a direct manipulation graphical editor that visually presents software components and allows a user to connect them together to construct running applications. The standard interfaces defining plug-compatibility for a particular component set constitute a *scripting model*. Every component is shown together with its *input and output ports*, which represent parameters, services, acquaintances, etc., of the software components. Ports may be visually presented in a variety of ways, such as knobs, buttons, text fields, menus, etc., depending on what the intended semantics are.

Some potential advantages of scripting over programming are:

- *Direct manipulation:* one can directly construct the running application without the need for an edit/compile or edit/interpret cycle;
- *Self-documentation:* components show only their interfaces, and can be connected to a SIB to provide on-line help;
- *Interactive type-checking:* the interfaces of a component set can be known to the scripting tool and checked interactively;
- *Scripts as components:* higher-level components can be built up as scripts and encapsulated as new components; incomplete scripts may constitute generic designs.

Many special-purpose "scripting tools" already exist, but each is tailored to a specific scripting model, to support, for example, dataflow, or form design, or user interface design. We are developing, within ITHACA, a general-purpose scripting tool, called Vista, which is capable of supporting multiple scripting models [27] [22].

To sum up, the capital investment required for developing GAFs can only be justified if, in the long-term, a component-oriented software life-cycle is cost-effective. The shift towards *open systems* guarantees that this must be the case. Whereas in the past large computer systems were monolithic and relatively independent, today's computer systems are increasingly open in terms of *topology*, *platform* and *evolving requirements* [35]. What this means is that (1) traditional software development methodologies that rely on relatively stable requirements are bound to fail, and (2) standard interfaces and component reuse are concerns of *software communities*, not just individual companies.

This shift of attention from closed to open architectures suggests that component-oriented software development can rely on certain economies of scale to justify the investment in reuse, but there are both technical and cultural problems to be resolved first. This is the topic of the next section.

4 Large-Scale Reuse and Object-Orientation

This section explores possible consequences, and related research problems, that may arise should object-oriented programming become widely used within the software industry. Object-orientation will then affect the way software is treated as a product to be packaged and marketed. Some advocates of object-orientation have pointed out that a pervasive use of object-oriented technology would likely lead to a restructuring of software production and marketing practices. For example, Meyer speaks of a new "software culture" occurring when developers take reuse into account [24], while Cox has advocated a software component industry based on object-oriented "software-ICs" [9].

It is our view that a restructuring of the software industry does not necessarily follow from the adoption of object-orientation, but rather the causative factor is what we call *large-scale reuse*. Large-scale reuse is a style of software development; it is characterized by the reuse of software and software-related information across boundaries of specific projects, products and organizations. Object-oriented programming is certainly conducive to large-scale reuse since it creates software components amenable for reuse. There are additional reasons, however, why large-scale reuse is becoming more feasible. These include the use of distributed software development environments, which often require a high degree of software portability and reusability, and improvements in networking al-

lowing the exchange of software and software-related information.

Suppose we set aside the question of how large-scale reuse benefits software producers and consumers, and instead focus on the following question: What changes are needed in object-oriented technology (in particular, programming and runtime environments) in order to encourage or enable large-scale reuse? There appear to be three major areas of concern:

1. Class management,
2. Object interaction and communication, and
3. Object pricing and marketing.

The remainder of this section explores these topics and their relationship to large-scale reuse.

Class Management

To fully take advantage of an object-oriented language, a class collection is needed. In large scale-reuse, such collections will be used by many programmers. (And, in extreme cases, the user population for a class collection is essentially the user population for the underlying language.) Viewing a class collection as a shared resource, a number of problems arise which we collectively refer to as *class management* [13].

For instance, since the collection is shared, the names of classes must be agreed upon to some extent, and care must be taken in assigning names in a consistent manner that avoids conflicts. Thus there is a problem of *class naming*, i.e., administering the class name space. Possible approaches include naming conventions, dividing the name space among software producers, and setting up standards-like bodies which oversee the collection.

A related problem deals with modifying existing classes within the collection. Such changes may have a vast impact and so need to be controlled. In some cases programs may no longer perform correctly with the new collection so it may be necessary to maintain historical versions (or "releases") of the collection. The examination of the various types of modifications to a class collection and how to assure that existing applications can run (and be recompiled if necessary) is referred to as the problem of *class evolution*. Many techniques have been proposed for dealing with class evolution (see [4] for a survey of this area) but their integration with development environments and methodologies is still problematic.

A particular form of evolution is the extension of a class collection by subclassing. This leads to the problem of when to incorporate such extensions within the collection and how to manage variants. For example, a particular project may, for its own purposes, define a number of new subclasses. At some point it may then decide to merge the local extensions with the shared collection. Again, as with name space administration, agreed upon procedures, and tools which support those procedures, will be needed.

Given a large class collection one needs retrieval aids. *Class cataloging* addresses how to index or describe classes in order to facilitate their retrieval. (Examples of software indexing schemes are presented in [16] and [30].) Class cataloging must also address the design of storage structures and access methods for class retrieval. Here a better understanding of the development process is needed: What search criteria will most assist developers looking for components to reuse? It seems clear that a class retrieval facility should deal not only with descriptions of classes but also with other information —- including design documents, performance data, and user feedback. However, designing suitable indexing and access methods for such an amorphous collection of information is a difficult task, and is further complicated by the need for many views (such as those of a programmer, designer, or project manager) of the information [11].

Software information systems are a key part of the infrastructure needed for large-scale reuse. In the previous section we mentioned briefly the "software junkyard" and the "software cookbook." These refer to the problem of determining the organization and content of a software information system. Another problem, and one critical to large-scale reuse, is the nature of the interface between the software information system and development or programming environments. Here many scenarios are possible, ranging from loosely-coupled software clearinghouses (such as are developing on the Internet), to cases where a distributed geographically-disperse software information system is transparently integrated with local programming environments. It is not clear which (if any) of these scenarios will predominate. However, it is fair to say that development environments are making increasing use of networking, and some form of linkage to wide-area services should be expected.

Object Interaction, Communication and Cooperation

Large-scale reuse is particularly appropriate when applications are developed by scripting plug-compatible components. In order to avoid a profusion of components, each object should operate in a variety of contexts. It should be possible to construct applications requiring interaction and communication between objects that possibly run on different hardware platforms, were developed using different programming languages and/or environments, or even present non-compatible interfaces. In other words, it should be possible to construct applications from *co-operating*, yet autonomous, objects. Basic issues here are:

- *Interface representation:* techniques for describing object interfaces and their parameters;
- *Naming:* name spaces for objects and their services, name lookup services;
- *Communication mediation:* use of intermediary objects as "glue" between non-com\-pa\-tible objects;
- *Security:* preventing unauthorized access to an object.

These issues have been addressed for some time by distributed object systems (see [6]), but these systems usually possess some degree of homogeneity —- either of operating system or programming language. The same issues also need be addressed for loosely-coupled, wide-area, non-homogeneous systems. As an example, OMG, the Object Management Group (an organization which aims to develop and promote standards for software interfaces) is concerned with object interaction and communication for large, system-like, objects running in heterogeneous environments. The OMG has specified an appli-

cation framework [33] including an "Object Request Broker" for handling such things as request and reply dispatching, parameter encoding, and name services.

Object *cooperation* can be viewed as a higher-level concept than either interaction or communication. As an example, two objects with very minimal interaction, perhaps the single exchange of an instance variable, can hardly be said to be cooperating. But one object delegating a task to a second comes closer to the notion of cooperation. Object cooperation clearly requires interaction and communication, in addition there must be some shared goal or purpose. The question is: Can a more rigorous definition of object cooperation simplify the task of building applications from interacting objects? At the moment the answer is not known, but it seems clear that some notion of cooperation is needed if applications constructed from pre-designed components are to satisfy requirements and solve problems.

Object Pricing and Marketing

If we look at large-scale reuse from the perspective of software providers, two critical issues arise:

1. There must be mechanisms in place which compensate providers for reuse of their software. We call this *object pricing*.
2. There must be mechanisms in place which allow providers to promote their software without revealing full details of design or implementation. We call this *object marketing*.

If an application is constructed solely from in-house or public domain software, object pricing is not an issue. This, however, is becoming infeasible because of the increasing complexity of application software. Many applications now intimately depend upon or utilize separate systems (e.g., window systems or databases) that were not developed explicitly for the application in question. The large-scale reuse paradigm takes this approach one step further —- applications are constructed from many objects coming from many sources. Furthermore, software developers, in addition to designing complete applications, may choose to concentrate on just the smaller objects needed to build applications. These "object providers" clearly will not emerge unless mechanisms are established by which they are compensated for the reuse of their product. Possible solutions to this problem include service-based and royalty-based pricing schemes [14]. The latter may be particularly appropriate for large-scale reuse and so we see as an important problem the development of "pay as you go" schemes, such as the Japanese "Superdistribution" proposal [25], for multi-object applications.

There is a essential difference between constructing applications from large reusable components (subsystems) and constructing applications from small reusable components (e.g., classes). In either case the application developer requires a component's specification in order to incorporate the component within the application. However, given a component's specifications, the smaller the component the easier it is to implement it. Consequently the providers of small components face a dilemma: Descriptions of the component must be available to potential reusers, but, by making this information available, the provider risks a reuser reimplementing the component, so avoiding any obligation towards the provider. Object marketing deals with this problem. In particular it must address: protecting designers and implementors of components from similar reimplementations of their components, progressive levels of disclosure of component specification and implementation information, and procedures for supplying this information to potential reusers.

To illustrate these issues, consider the following possible example of component marketing. Suppose each component has the following, progressively more detailed, levels of description:

1. the name of the component and its domain of applicability
2. an informal description of the component
3. the names and informal descriptions of the functions provided by the component
4. the specification of the component's interface
5. the object code for the component
6. the source code for the component

Here disclosure up to level 3 would probably be sufficient for a reuser to decide whether or not the component is of interest. Disclosure to level 5 would be needed to actually reuse the component, and disclosure of the final level would perhaps be needed to refine or modify the component.

One marketing scenario would be to allow the provider of a component to determine a (onetime) charge for each level of disclosure. An object pricing mechanism would then be needed to keep track of access to component descriptions and accumulate charges. If a provider wanted to advertise a component then the first few levels of disclosure would have zero or negligible cost.

Large-scale reuse requires a high-degree of automation of both object pricing and object marketing —- non-automated accounting of who is using which component would make the approach too burdensome. Thus, the crucial problem is how to build, into software development and runtime environments, support for object pricing and object marketing. Issues here include designing the necessary communication infrastructure and, more generally, establishing a legal framework governing the reuse of software components.

What is needed in the long term is a software community where objects can be exchanged and evolved, and bought and sold in an open, competitive market. Such a community is slowly emerging and initiatives like OSF, the Open Software Foundation, OMG, the Object Management Group, and ESSI, the European Software and Systems Initiative, can help it emerge faster. Needless to say, the effects of these changes will be very profound on both the software users and the companies that build and sell software.

5 Concluding Remarks

We have argued that object-orientation has the potential to greatly affect the way that software is developed and used. Needless to say, in most applications there is a certain inertia due to established methods and vested interests that will hinder

the wide-spread use of object-oriented ideas. There are, however, many areas that are relatively new and where innovative ideas can have a tremendous and immediate influence, such as, for instance, graphics, multimedia, and computer-supported cooperative work (CSCW). (Examples of applying the object-oriented approach in these areas can be found in [20], [40] and [15].)

A wider influence of object-orientation will be the different ways in which it affects how we view software as a product. Like any other product, a software application has to be designed, built, sold and used. Like every product, a software application is designed to look a certain way and to have certain functionality. Object-orientation affects both the user interface and the visualization of a software application. Object-orientation affects the way that its functionality is conceived in terms of blocks of more basic functionality. Like every other product, a software application needs to be built. Object-orientation affects the way that a software application is built from components. Finally, object-orientation can have a tremendous effect in the way software is marketed both in terms of components and in terms of final integrated systems.

References

[1] M. Ader, O.M. Nierstrasz, S. McMahon, G. Müller and A-K. Pröfrock, "The ITHACA Technology: A Landscape for Object-Oriented Application Development," in *Proceedings, Esprit 1990 Conference*, Kluwer Academic Publishers, Dordrecht, NL, 1990, pp. 31-51.

[2] P. America, "A Parallel Object-Oriented Language with Inheritance and Subtyping," Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, vol. 25, no. 10, Oct 1990, pp. 161-168.

[3] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[4] E. Casais, "Managing Evolution in Object Oriented Environments: An Algorithmic Approach," Ph.D. thesis (no. 369), Centre Universitaire d'Informatique, University of Geneva, May 1991.

[5] C. Chambers and D. Ungar, "Making Pure Object-Oriented Languages Practical," Proceedings OOPSLA/ECOOP '91, ACM SIGPLAN Notices, vol. 26, no. 11, Nov 1991, pp. 1-15.

[6] R.S. Chin and S.T. Chanson, "Distributed Object-Based Programming Systems," ACM Computing Surveys, vol. 23, no. 1, March 1991, pp. 91-124.

[7] P. Constantopoulos, M. Dörr, E. Pataki, E. Petra, G. Spanoudakis and Y. Vassiliou, "The Software Information Base — Selection Tool Integrated Prototype," ITHACA report FORTH.91.E2.#3, Foundation of Research and Technology — Hellas, Iraklion, Crete, Jan 12, 1991.

[8] Wm. Cook, "A Proposal for Making Eiffel Type-safe," in *Proceedings ECOOP '89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 57-70.

[9] B.J. Cox, *Object Oriented Programming — An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[10] B.J. Cox, "Planning the Software Industrial Revolution," IEEE Software, vol. 7, no. 6, Nov. 1990, pp. 25-33.

[11] P. Devanbu, R. Brachman, P. Selfridge and B. and Ballard, "LaSSIE: A Knowledge-Based Software Information System," CACM, vol. 34, no. 5, May 1991, pp. 34-49.

[12] M. G. Fugini, O.M. Nierstrasz and B. Pernici, "Application Development Through Reuse: The ITHACA Tools Environment," in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 99-114, To appear ACM SIGOIS Newsletter.

[13] S.J. Gibbs, D. Tsichritzis, E. Casais, O.M. Nierstrasz and X. Pintado, "Class Management for Software Communities," Communications of the ACM, vol. 33, no. 9, Sept 1990, pp. 90-103.

[14] S.J. Gibbs and D. Tsichritzis, "Software Licensing versus Software Reuse," in *Object Management*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, pp. 107-115.

[15] S. Gibbs, "Composite Multimedia and Active Objects," Proceedings OOPSLA/ECOOP '91, ACM SIGPLAN Notices, vol. 26, no. 11, Nov 1991, pp. 97-112.

[16] R. Helm and Y.S. Maarek, "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries," Proceedings OOPSLA/ECOOP '91, ACM SIGPLAN Notices, vol. 26, no. 11, Nov 1991, pp. 47-61.

[17] R.E. Johnson and B. Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming, vol. 1, no. 2, 1988, pp. 22-35.

[18] D.G. Kafura and K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," in *Proceedings ECOOP '89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 131-145.

[19] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen and K. Nygaard, "The BETA Programming Language," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver, P. Wegner, The MIT Press, Cambridge, Massachusetts, 1987, pp. 7-48.

[20] K.-Y. Lai and T.W. Malone, "Object Lens: A "Spreadsheet" for Cooperative Work," in *Proceedings CSCW '88*, Portland, Oregon, Sept. 1991, pp. 115-124.

[21] W. LaLonde and J. Pugh, "Subclassing /= Subtyping /= Is-a," Journal of Object-Oriented Programming, vol. 3, no. 5, Jan 1991, pp. 57-62.

[22] V. de Mey, B. Junod, S. Renfer, M. Stadelmann and I. Simitsek, "The Implementation of Vista — A Visual Scripting Tool," in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 31-56.

[23] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[24] B. Meyer, "The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design," in *Proceedings TOOLS '89*, Nov. 1989, pp. 13-23.

[25] R. Mori and M. Kawahara, "Superdistribution: The Concept and the Architecture," Transactions of the IEICE, vol. E 73, no. 7, July 1990, pp. 1133-1146.

[26] O.M. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents," Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, vol. 25, no. 10, Oct 1990, pp. 38-43.

[27] O.M. Nierstrasz, D. Tsichritzis, V. de Mey and M. Stadelmann, "Objects + Scripts = Applications," in *Proceedings, Esprit 1991 Conference*, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552.

[28] O.M. Nierstrasz, "Towards an Object Calculus," in *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Programming*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, LNCS 612, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991, to appear.

[29] M. Papathomas and O.M. Nierstrasz, "Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space," in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 189-204.

[30] R. Pietro-Diaz, "Implementing Faceted Classification for Software Reuse," CACM, vol. 34, no. 5, May 1991, pp. 88-97.

[31] R.K. Raj and H.M. Levy, "A Compositional Model for Software Reuse," in *Proceedings ECOOP '89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 3-24.

[32] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," Proceedings OOPSLA '86, ACM SIGPLAN Notices, vol. 21, no. 11, Nov 1986, pp. 38-45.

[33] R. Soley (ed.), *Object Management Architecture Guide*, Object Management Group, Frameington, MA, Nov. 1990.

[34] D. Tsichritzis and O.M. Nierstrasz, "Fitting Round Objects into Square Databases," in *Proceedings ECOOP '88*, ed. S. Gjessing and K. Nygaard, LNCS 322, Springer Verlag, Oslo, August 15-17, 1988, pp. 283-299.

[35] D. Tsichritzis, "Object-Oriented Development for Open Systems," Information Processing 89 (Proceedings IFIP '89), North-Holland, San Francisco, Aug 28-Sept 1, 1989, pp. 1033-1040.

[36] D. Ungar and R.B. Smith, "Self: The Power of Simplicity," Proceedings OOPSLA '87, ACM SIGPLAN Notices, vol. 22, no. 12, Dec 1987, pp. 227-242.

[37] P. Wegner, "Capital-Intensive Software Technology," IEEE Software, vol. 1, no. 3, July 1984.

[38] P. Wegner, "Dimensions of Object-Based Language Design," Proceedings OOPSLA '87, ACM SIGPLAN Notices, vol. 22, no. 12, Dec 1987, pp. 168-182.

[39] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," ACM OOPS Messenger, vol. 1, no. 1, Aug. 1990, pp. 7-87.

[40] P. Wisskirchen, *Object-Oriented Graphics*, Springer-Verlag, Heidelberg, 1990.