Subject-Oriented Programming (A Critique of Pure Objects)

William Harrison and Harold Ossher
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

Object-Oriented technology is often described in terms of an interwoven troika of themes: encapsulation, polymorphism, and inheritance. But these themes are firmly tied with the concept of *identity*. If object-oriented technology is to be successfully scaled from the development of independent applications to development of integrated suites of applications, it must relax its emphasis on the *object*. The technology must recognize more directly that a multiplicity of subjective views delocalizes the concept of object, and must emphasize more the binding concept of identity to tie them together.

This paper explores this shift to a style of object-oriented technology that emphasizes the subjective views: *Subject-Oriented Programming*.

1. Introduction

Figure 1 illustrates the definition of a **tree** in a commonly accepted way of thinking about objects, sometimes called the *classical model* [22]. In this model, a tree is defined by defining a class, the class of all trees, in terms of internal state information and methods that can be applied. Proponents of the advantages of *data abstraction*, a form of *encapsulation*, emphasize the fact that *client programs* manipulating these trees do so only through the exposed *operations*.

Ideally, the designer of such an object-oriented tree defines and works with the *intrinsic* properties and behavior of a tree. In the real-world, properties of a tree like its height, cell-count, density, leaf-mass, etc. are intrinsic properties. Intrinsic behaviors include things like growth, photosynthesis, and other behaviors that affect the intrinsic properties.

This ideal classical model is utterly inadequate to deal with the construction of large and growing suites of applications manipulating the objects. Designers of such suites are forced either to forego advantages of the object-oriented style or to anticipate all future applications, treating all extrinsic information as though it were intrinsic to the object's nature. Figure 2 shows an example of the situation that gives rise to this pressure. In it, we see that a tax-assessor has his own view of characteristics and behaviors associated with a tree. The characteristics include its contribution to the assessed value of the property on which it grows. The behaviors include the methods by which this contribution is derived. These methods may vary from tree-type to tree-type. In fact, such methods may form part of a tax assessor's view of all objects, tree and non-tree alike. These characteristics and behaviors are *extrinsic* to trees. They form part of an assessor's subjective view of the object-oriented tree.

With the classical object model, the designer of the tax-assessor application is faced with one of two choices. On one hand, the application can be constructed as a *client* using the encapsulated methods but forgoing the advantages of *encapsulation* and *polymorphism* for the tax assessor application's state and methods. On the other hand, the application's function could be integrated into the same **tree** class manipulated by other applications; in effect, treating

OOPSLA'93, pp. 411-428

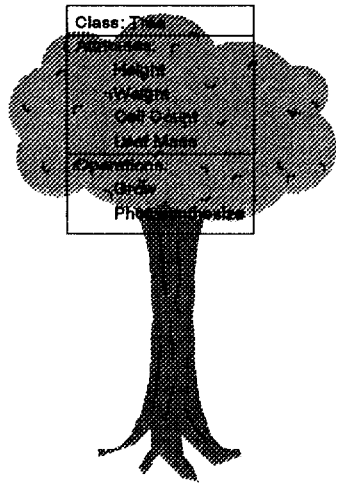

Figure 1 - An Object-Oriented Tree

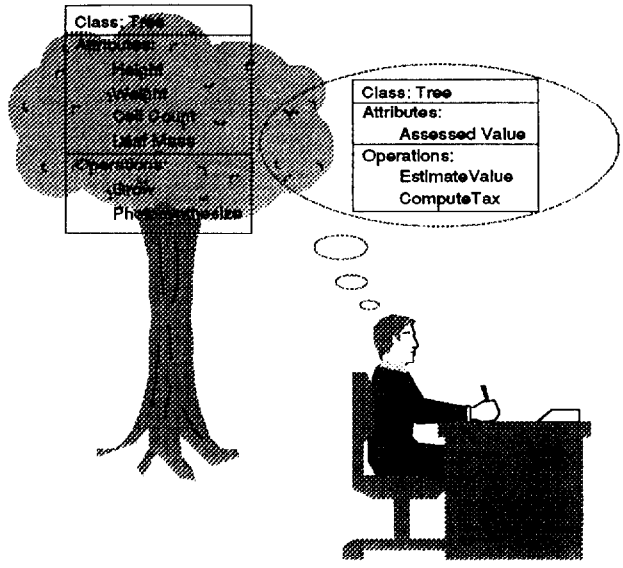

Figure 2 - A Tax Assessor's View of the Tree

the extrinsic characteristics of the tax-assessor application as though they were intrinsic to trees. Both choices are objectionable.

Figure 3 illustrates how unmanageable the latter approach is by reminding us that tax-assessor is merely one of a suite of applications, each of which has its own subjective view, its own extrinsic state and behavior for the tree.

Although the theme of subjects in anthropomorphic terms is illustrative, we should not lose sight of its importance in tool and application integration settings. The tree could easily be a node in a parse tree, the bird an editor, the assessor a compiler, and the woodsman a static-semantic analysis tool. Each of these tools defines its own state and methods on the parse-tree nodes, e.g. the editor has display status, the compiler has associated code expansions, and the checker has use-definition chains.

Either the developers of these applications cannot encapsulate their own state and behavior with the parse-tree node to gain the advantages of encapsulation and polymorphism, or the system designer must manage an ever-expanding collection of extrinsic state and behavior becoming part of the intrinsic node. In fact, in the presence of market pressure to adopt applications provided by vendors rather than do all development in-house, the definer of the node faces the impossible task of anticipating all future extrinsic requirements. This burden demands a more powerful model than the classical object model in order to facilitate the development of application suites. We propose *subject-oriented programming* as such a model.

Section 2 outlines the goals of subject-oriented programming. Section 3 then provides an overview of subjects, and sections 4 and 5 discuss aspects of subject interaction. Section 6 then describes a model of subjects, and elaborates some details in terms of the model. Section 7 discusses considerations in implementing efficient support for subject-oriented programming. Section 8 provides a more concrete example of the use of subjects in defining software development environments, and Section 9 discusses related work.

2. Goals

The overall goal of subject-oriented programming is to facilitate the development and evolution of suites of cooperating applications. Applications cooperate both by sharing objects and by jointly contributing to the execution of operations. The following requirements are important in this context:

- It must be possible to develop applications separately and then compose them.
- The separately developed applications should not need to be explicitly dependent on the other applications they are to be composed with.
- The composed applications might cooperate loosely or closely, and might be tightly bound for

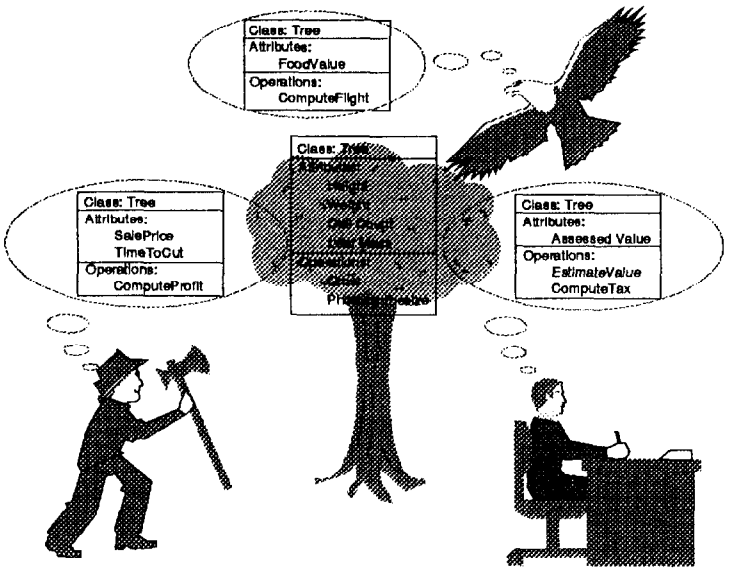

Figure 3 - Many Subjective Views of an Object-Oriented Tree

frequent, fast interaction, or be widely distributed.

- It must be possible to introduce a new application into a composition without requiring modification of the other applications, and without invalidating persistent objects already created by them. Ideally, even recompilation of the applications should not be required, except to facilitate global optimization if desired.
- Unanticipated new applications, including new applications that serve to extend existing applications in unanticipated ways, must be supported. The notion of extending an application by writing an "extension application" and composing it with the base application is discussed in detail in [14].
- Within each application the advantages of encapsulation, polymorphism and inheritance – object-oriented programming – must be retained.

These requirements and their relationship to object-oriented technology are discussed in more detail in [10]. As illustrated in Section 1, the classical object model does not satisfy them.

In the subject-oriented paradigm, each application is a subject or a composition of subjects: it defines just the state and behavior pertinent to the application itself, usually fragments of the state and behavior of collections of relevant classes. As discussed in the rest of this paper, the semantics of subject composition and interaction ensure that the requirements listed above are satisfied.

3. Subjects

This section discusses general characteristics of subject-oriented programming without introducing a specific model. Because the use of a more format or detailed model makes discussion more precise, however, Section 6 will introduce such a model and revisit some of the topics addressed in this section in some more detail.

We use the term *subject* to mean a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool.[1] Thus, although for smoothness of flow we may occasionally speak of subjects as individuals, they are not the individuals themselves, but the generalized perception

[1] The term "subject" differs somewhat from its use by Coad and Yourdon [2], although both usages share the idea of reflecting a smaller, more focussed perception of a complex shared model. We avoided the similar term "view" in order to emphasize the stronger philosophical similarity with non-classical philosophical trends that emphasize the idea that subjective perception is more than just a view filtering of some objective reality. The perception adds to and transforms that reality so that the world as perceived by a body of perceptive agents is more than the world in isolation.

of the world shared by some individuals. Similarly, subjects are not classes. They may introduce new classes into the universe, but subjects generally describe some of the state and behavior of objects in many classes.

One often thinks of particular state and behavior as being *intrinsic* to an object: the state and behavior that describe its essential characteristics, as opposed to additional state and behavior associated with it by various subjects. For example, the height, weight and other attributes and behavior associated with trees in Figure 1 might be considered intrinsic to trees, whereas the other attributes and behavior such as assessed value and sale price are not. In the subject-oriented model there is no special status accorded to the intrinsic properties. The developer is free, if she chooses, to have a subject that implements the intrinsic properties of one or more classes of objects, and to require that any manipulation of the intrinsic properties employ that particular subject to carry it out.

The essential characteristic of subject-oriented programming is that different subjects can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects. Only object identity is necessarily shared.

A subject is definitional or schematic — it corresponds to a traditional (though usually incomplete) class hierarchy, describing the interfaces and classes known to this subject. A subject does not itself contain any state.

3.1. Activations and Compositions of Subjects

A *subject activation*, often referred to just as an *activation*, provides an executing instance of a subject, including the actual data manipulated by a particular subject.

Subjects can be combined to form cooperating groups called *compositions*. The composition also defines a rule, the *composition rule* that specifies in detail how the components are to be combined; for example, how methods from different subjects for the same operation and class are to be combined, and whether nested compositions form separate scopes or are all combined into a single scope. A great variety of composition rules is possible, and some examples will be given in subsequent sections.

In distributed systems, subject activations may be separated in space and time, and state changes to independent subject activations of an object may occur in separate transactions. This makes even the concept of a unified "state of the object" inaccurate and misleading. A vital aspect of subject-oriented programming is that it be possible to extend subjects and to introduce new subject activations without disrupting others. It is therefore important that neither source nor object code rely on the global "state of the object" or on its format.

Accordingly, we use the term *object-identifier* or *oid* to mean the globally known unique identification of the *object* as it appears in the context of one or more subjects of interest. In the context of a particular subject, we also use *object* to mean the state and behavior associated with an object identifier by that subject. Similarly, there is no global concept of class: each subject contains class descriptions that describe state and behavior from that subject's point of view.

Aspects of a this way of manipulating objects will be expanded and explored in the following sections, in the course of explaining the features of subject-oriented programming. As mentioned, a more precise but also more particular model will be introduced later in Section 6.

3.2. Relationship to O-O Technology

The classical object model is, in many respects, the model of objects seen by any one subject. Within a subject, an object has an implementation class that defines the implementation of behaviors provided for the various operations supported by the object and the state information needed by these implementations. Subject-oriented programming thus includes object-oriented programming as one of its technological elements.

Interfaces

Interfaces describe in abstract terms the *operations* that a class of objects supports. Variables that point to objects , whether instance variables of objects, static or dynamic program variables or parameters, are declared in terms of the interfaces their contents must support, rather than in terms of specific classes.

This approach leads to greater encapsulation, and hence to greater flexibility and reuse. It is being increasingly accepted within the object-oriented community [5, 23]. Even in languages such as C++ that do not explicitly require separate interfaces, conventions are frequently adopted that amount to using certain classes (abstract base classes in C++) as interfaces. Separation of interface definition from implementation characterization is of even greater importance in subject-oriented programming, because interfaces are a point of agreement between separate subjects as to the operations that are available on an oid, without one subject needing to have any access at all to the class hierarchy describing the implementations provided by another subject.

Behavior

Behavior is specified by means of *methods*, which are the actual code implementing operations on specific classes of objects. In classical class-based models, all methods are associated, either explicitly or by inheritance, with the single class of which the object is an instance. In the subject-oriented approach, the class associated with each oid can differ from one subject activation to another. This means that each subject can specify its own behavior for each object. What is more, the inheritance relationships among classes can be different in different subjects.

State Information

State information is retained in *instance variables*. Classically, all of an object's state is treated as a unit, and is known and accessible to all methods associated with that object, though it might not all be used by all of them. In the subject-oriented approach, the state associated with a particular oid can vary from one subject to another. In addition, since the subject really just provides a template for state and behavior, there can even be be multiple activations of a single subject, each with its own state for the object.

4. Interactions Among Subjects

Since the model of an object seen by any one subject is essentially the classical object model, this conventional model is adequate if all subjects remain isolated. The need for a subject-oriented model arises when dealing with interacting subjects. This interaction can take any of several forms:

- A request for function or state change to be supplied by another subject. For example, the woodsman's invocation of "cut-down" affects the tree's height as well as her own caloric consumption
- Performance of an activity in which another subject might participate e.g. the assessor's estimate of the value of a tree may be reflected by a private interpretation of the woodsman's decision to estimate the effort to cut the tree down,
- Notification of an occurrence which may be of interest to another subject e.g. the bird's nest-building activity might influence the woodsman's schedule for cutting down the tree
- Use of one subject's behavior as part of the "larger" behavior of another e.g. the tax assessor may use the woodsman to cut down many trees to pay delinquent taxes (perhaps he's the Sheriff of Nottingham?).
- Sharing of state, e.g. the bird and the woodsman might both have the same notion of tree height.

Subjects interact only if they are composed with one another in a universe. Details of the interaction are determined by the composition rule. The rest of this section discusses some of the semantic details of subject composition.

4.1. Operation Invocation

All code in a subject-oriented framework executes in the context of a particular subject activation. An operation call can therefore be modelled as a tuple (a, op, p), where

- a is the subject activation making the call.
- op identifies the operation to be performed.
- p is a list of parameters. Some of these parameters will be oids. Some of them will be used to control operation dispatch, details dependent on the language used. In many embodiments, the first parameter will be considered the "controlling object" or "receiver", to be used for dispatch.

When an operation is invoked in a subject-oriented model it might cause execution of methods in multiple subjects. The composition rules control what happens, so that within a subject-oriented model, there is freedom to craft and use different composition rules. It is therefore possible to describe complex combinations of detailed aspects of the separate subjects. The most useful composition rules, however, are likely to be those that are simple and can specify

briefly some frequently useful ways of combining the details of separate subjects.

The simplest composition rule, closest to the rule in C++, Smalltalk or OMG CORBA[23], is perhaps: "An operation can be dispatched to only a single subject activation." However, this rule rather severely limits the usefulness of composition to preplanned extension of existing frameworks. An example of a simple and more appealing composition rule, called *merge*, is:

1. For each subject activation in arbitrary order (but not in parallel), dispatch the operation within that activation. This "local" dispatch within an activation is whatever form of object-oriented dispatch is provided by the language in which that activation's subject is written. When an operation implementation is dispatched to within a separate subject, the dispatch is directed using the receiver's classification of the object. Thus dispatch can be based on different classes in different subjects.
2. If the methods return values, all the return values must be identical, or an exception is raised.
3. If the operation defines *inout* parameters, they may be set by one method and used by the next. However, the operations performed on them should be commutative.

These characteristics provide a commutative composition with the desirable characteristics discussed in [14]. Tool composition in OOTIS provides a composition rule similar to merge, with an efficient underlying implementation [10].

It is worthwhile to illustrate the way in which the *merge* composition rule enables the distributed implementation of an object. Assume, for example, that both the tax assessor and the woodsman are tracking the existence of a nest in the tree. The bird, uses operations called **make-nest** and **abandon-nest** to perform its nest-building work. To track this behavior, the woodsman and tax assessor each can supply behaviors for **make-nest** and **abandon-nest**. In the implementations used by the woodsman and tax assessor, make-nest increments a state variable in their representation of the tree and abandon-nest decrements the variable. So, each time a bird makes a nest in a tree, the assessor and woodsman also update their "mental models" of the tree to record the nesting. Further, if one or more of the subjects (bird, woodsman, tax-assessor) provides a **has-nest** the result should always be the same, whether the implementation is supplied locally or by sharing from another subject.

The example rule above, by cycling through all subject activations, does not respect subsidiary compositions. An alternative rule, called *nesting*, treats compositions as scopes, and allows calls to propagate beyond scopes only if specified explicitly. A discussion of nesting is made within the framework of a more specific model in Section 6. As with merge, tool composition in OOTIS also provides an efficient implementation for a composition rule similar to nesting.

These examples should make it clear that many variations are possible, both subtle and dramatic. Formulation and efficient implementation of various composition rules is an interesting topic for future research.

4.2. Object Creation and Initialization

The sharing of behavior among subjects leads to the fact that creation and initialization behavior must be shared as well. A request to create an object can be modelled as a tuple (a, c), where

- a is the subject activation making the request.
- c is the name of a class defined in a's subject.

For reasons noted below, parameters specifying initial values are not permitted in the create request. Although creation of an object is thus requested by one subject, other subjects also need to initialize information before they operate upon the object.

The steps involved in object creation and initialization are:

1. Allocation of an oid from the OID set.
2. For each subject activation, determining the appropriate class.
3. For each subject activation, allocating space for the object's state
4. Placing the appropriate initial values in the storage allocated.

The creation operation specifies the class to be created from the point of view of the activation requesting the creation. Classifying the object within that activation is straightforward. Classifying the object for the other subject activations involves class matching across subjects according to the composition rule. This issue

is discussed in Section 5. The composition rule is also responsible for specifying whether any instance variables are to be shared across subjects. How this is accomplished depends on the details of the subject model and is explained further in Section 6.

The composition rule can use either of two approaches to the timing of a subject's initialization of an object:

- *Immediate initialization*, in which all subjects participate in the initialization at the time the object is created. This has the advantage of the conceptual simplicity of its determinism.
- *Deferred initialization*, in which subjects participate in initialization of the object only as they need to respond to an operation on it at later times. This can have substantial performance benefits: time is saved by avoiding communication with numerous and potentially remote subjects, and space is saved because it is not allocated unless the subject actually participates in behavior of the object. Deferred initialization also facilitates graceful introduction of new subjects that extend existing objects. For these reasons, deferred initialization was selected in CLORIS[9].

Deferred initialization precludes the use of parameters to the creation operation for determining initial values. Such parameters are undesirable in the subject-oriented context in any case, however, since it would be undesirable to require addition of parameters to all creation invocations whenever a new subject arises. (One might, in fact, argue that the fact that even an perception of the object's intrinsic state and behavior gradually evolves should argue against the use if initialization parameters even in the classical model. Considerations like this illustrate the value of viewing even the "intrinsic" object as a subject.)

4.3. State References

When a method within an activation is executing, it can access only the instance variables that it's subject specifies. Note that this subject-oriented approach to state references provides tighter encapsulation than classical object-oriented models: only a subset of the instance variables are accessible to each method, in general. Methods in different subjects can manipulate the same instance variables if data sharing between subjects is done, as described above.

4.4. Points of Agreement

It should be clear from the discussion above that two arbitrary subjects cannot necessarily be composed with any expectation that they will cooperate effectively. There does need to be limited agreement between them:

- Since multiple subjects can respond to the same operation call, there needs to be agreement among subjects regarding the operation interfaces. When an operation is called, all relevant subjects must agree as to what operation it is that is being called, and must understand the parameters. Each subject contains descriptions of the interfaces provided for the classes it defines. Determining appropriate correspondences among interfaces used by composed subjects is termed *interface matching*.
- Since one subject can operate upon an object (oid) that another subject has created, there needs to be agreement among subjects regarding the nature of objects. Each subject contains its own classification hierarchy. Determining appropriate correspondences among classes defined by composed subjects is termed *class matching*.

Interface and class matching strategies are dictated by the composition rule.

Various strategies are possible for interface and class matching. The simplest and most rigid requires identity, agreeing on a set of interfaces and a set of classes for the whole suite of subjects. Each subject is then written with those global definitions in mind. Even this is less restrictive than the classical model, because agreement on classes is really only on class names; each subject is still free to supply its own state, behavior and superclass definitions for each class.

Nonetheless, identity matching is too rigid to deal with composition of separate subjects (like pre-existing applications or applications developed completely separately) that were not written to predetermined, global definitions. The more flexible matching is, the greater the differences it can cope with, and the more potential there is for composing diverse subjects.

Both interface matching and class matching are interesting and important areas for future research. We do

not address interface matching in this paper. The next section discusses a spectrum of possible approaches to class matching.

5. Matching Classes Across Subjects

Subject-oriented programming can accommodate the real-world's characteristic that different subjects classify objects in different hierarchies, and that one subject might manipulate objects that another subject has not classified at all. Consider, for example, Figure 4. The bird classifies objects into plants (nectar-providing plants, insect-providing plants), nestables, and predators. The woodsman, on the other hand, classifies objects into nontrees, and trees (hardwood and softwood). This classification represents two different ways of looking at an underlying instantiable universe of pine, maple, cherry, dandelion, woodsman, bird, and object.

Often, diagrams of this sort are used to illustrate the fact that *interface* hierarchies are generally not mutually conformable. The most important thing to realize about this illustration is that we are dealing here, however, with *implementation* hierarchies. The **bird** subject defines state information and methods needed for processing plants (nectar-providing plants, insect-providing plants), nestables, and predators. The **woodsman** subject defines state information and methods needed for processing nontrees, and trees (hardwood and softwood).

In all examples thus far, we have discussed the interaction of subjects in a way that presumed all instantiable classes for which processing is shared across subjects are declared and classified by all subjects. This presumption is already less constraining than the classical object model found in language-based O-O technology like that provided by C++, which presumes that all classes, not just the instantiable ones, are declared and similarly classified by all subjects. However, the constraint is really still too strong to support the degree of independence desirable in the composition of a community of subjects. Suppose for example, as shown in Figure 5, that the bird is familiar with locust trees as well as the others. This may have resulted from the release of an enhanced bird and may someday be handled by an enhanced woodsman. But in the available world, locusts are known in detail only to the bird.

Two sorts of problems arise: the semantics of cooperative operation implementations and the possibility of direct discovery of objects of unknown classes. Let us take these in turn.

5.1. Cooperative Operations On an Unknown Class

When subjects interact in a universe, differences between their class hierarchies must be resolved by the composition rule. Such resolution is essential to enable each subject's classification of an object to be set correctly at the object's initialization (whether immediate or deferred).

If the bird nests in a locust, the bird's shared nestable behavior will include possible implementations of the **make-nest** operation from the woodsman. In consequence, the woodsman will be asked to dispatch this operation on a locust, a class of which she has no knowledge.

One way of resolving this undefined situation is to specify that all such circumstances result in null invocations in the subject. In the absence of the "discovery" situations discussed next, this is a semantically well-formed, although perhaps unsatisfactory, definition. It merely makes the subject totally "blind" to objects of that class.

This solution is unsatisfactory in that one might expect the woodsman to treat the locust as other trees. But, unless the woodsman has some way to understand its tree-ness, she has no way to classify it. In fact, however, we also need to develop a stronger solution to treat the cases in which the woodsman is forced deal with a locust because she stumbles right over it.

5.2. Discovering Objects Of an Unknown Class

In the course of normal processing, a subject obtains the identity of an object as the result of a function, instance variable reference, or operation call on another object. If the object is not yet classified by the subject, some determination must be made as to how to classify the object so that the operation can be properly dispatched. Such classification will be based on information obtained either directly from one or more other subjects or from the interface definitions that govern the sharing of objects.

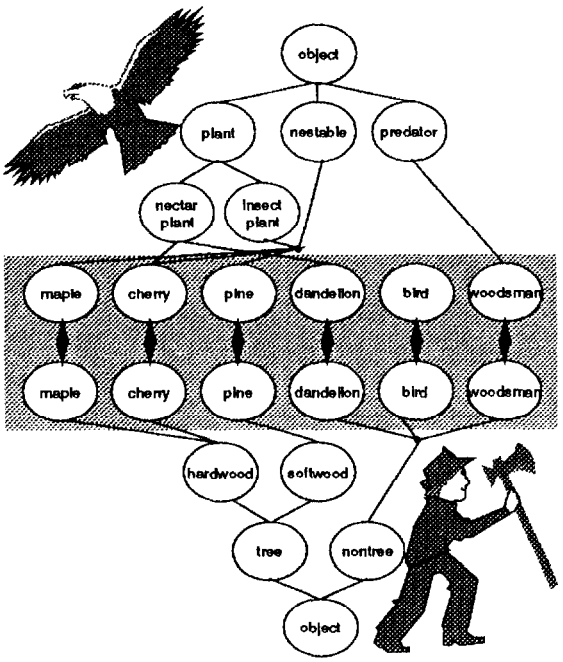

Figure 4 - Two Class Hierarchies over the Same Instantiable Objects

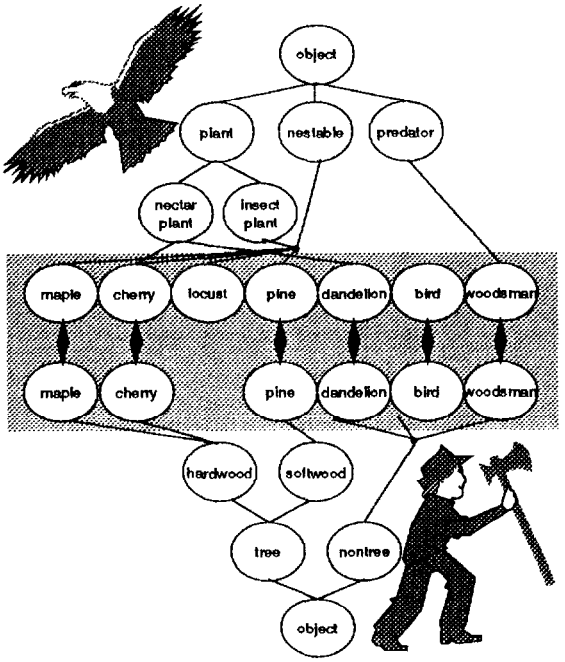

Figure 5 - Two Class Hierarchies over Different Instantiable Objects

There are many possible approaches to performing this classification. Exploration of these possibilities is an interesting and important area of research. We begin this exploration by identifying a spectrum of approaches. The near end of the spectrum, explicit matching, is a simple generalization of today's object-oriented repository technology. It acts as a proof that useful solutions exist. From that known point we outline several, more speculative approaches of increasing power: inferred class matching, interface-based class matching, and operational classification. Detailed discussion and semantics of these approaches is beyond the scope of this paper.

Explicit Class Matching

Assume, for example, that the woodsman's class definitions can share information with the bird's. We might assume, that the woodsman's and bird's definitions share not only the instantiable objects, but the general superclass **object** and, in addition, that the bird's **insect-plant** class and the woodsman's **tree** class are explicitly matched. Given this knowledge, although **locust** is not known to the woodsman, it is known to be a subclass of **insect-plant** which is matched with **tree** which can be used by the woodsman to define the behavior for locust.

Inferred Class Matching

A less preplanned approach is also possible. Although **locust** is undefined in the woodsman's hierarchy, in **bird** it has both **insect-providing plant** and **nestable** as its superclasses. We can determine the set of their instantiable subclasses to be **maple**, **cherry**, and **pine**. These are also all subclasses of the woodsman's **tree**. Hence, the inference might reasonably be drawn that locusts should be treated as trees.

Interface-Based Class Matching

In the discussion thus far, we have treated all of the non-instantiable classes as though they were complete, placing no behavioral or interface constraints on their subclasses. If, on the other hand, they are what are often called *abstract classes*, then declaring or inferring that **locust** is a subclass of **tree** requires **locust** to provide certain behaviors. Since these behaviors are clearly not provided by woodsman, they are required imported behavior.

An alternative strategy for inference is to use the interface definitions rather than the implementation hierarchy for such an inference, even though the result is the determination of an implementation class to be used as the object's superclass. The reference to locust was derived with respect to an interface defi-

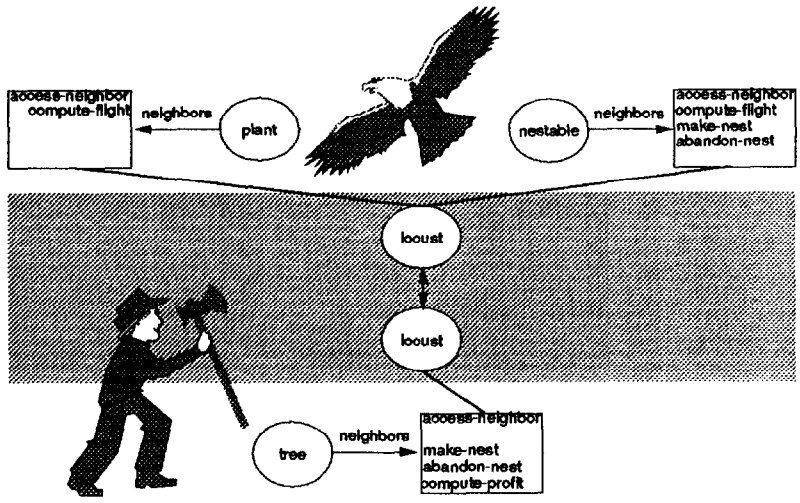

Figure 6 - Classification Using Interfaces for a Discovered Object

nition. It would be possible, therefore, to select any implementation class that meets the same interface. However, in the presence of a shared base of objects, the relationships among the objects becomes a shared property. As discussed before, each subject contains class definitions, one aspect of which is the specification of relationship links (pointers) between objects. These links are defined in terms of the interfaces that must be supported by the target object of the link.

In the world described by two composed subjects, the target objects must satisfy the union of the interfaces. In fact, the object must satisfy all of the interface constraints imposed by the relationships to it. This implies that in classifying an unknown instantiable class, it must be provided with all of the requisite behavior. Some behaviors are supplied by one subject and other behaviors by the other, as illustrated in Figure 6. Knowledge of the class assigned to the object in the other subject allows the determination of a viable classification for the new class, if one exists.

In the example we illustrate how the woodsman may be using an operation called *access-neighbors* to follow the **neighbors** relationship link from one tree to another. The woodsman expects to find an object that implements an interface supporting *access-neighbor*, *make-nest*, *abandon-nest*, and *compute-profit*. The woodsman actually encounters a **locust** which she does not know how to classify. But examining the **bird's** schema elicits the information that *access-neighbor*, *make-nest*, and *abandon-nest*. are provided by the bird. Therefore the locust may safely be classified under **tree** because **tree** defines an implementation for *compute-profit* and imports implementations for the others.

Operational Classification

The classification strategies described thus far can be thought of as "static," in that objects of the same class in one subject always have the same class in other subjects. This need not be so. A subject's classification of an object could be based on operational tests made at the time the object is introduced into the subject's classification. For example, if trees all support a **wood-density** operation, then the woodsman could classify the locust tree of Figure 5 as **hardwood** or **softwood** rather than simply as **tree**, and perhaps different subspecies of locust would be classified differently even though birds see them all as locusts. Operational classification is one way in which a single class in one subject may correspond to many classes in some other subject.

6. A Subject Model

This section presents a model of subject-oriented programming for definitional and explanatory purposes; it is *not* intended to suggest or constrain implementation architecture. Section 7 discusses means of implementing subject-oriented support efficiently. Components of the model are illustrated in the context of the tree example in Figure 7.

A *subject* is modelled as a tuple $S = (N, I, D, P)$ where:

- N is a set of class names

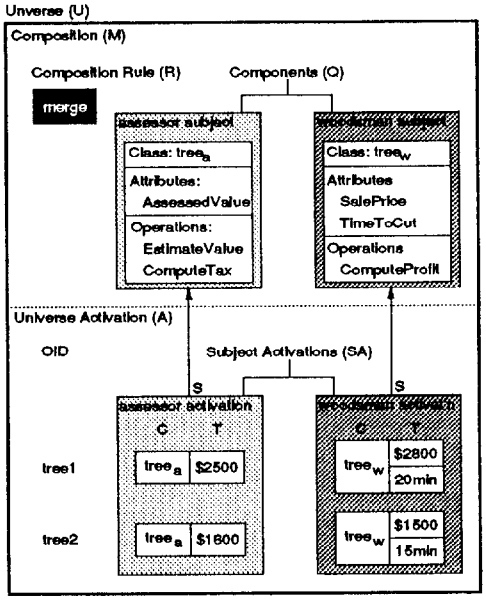

Figure 7 - A Simple Subject-Oriented Universe

- I is a set of interfaces, defining operation signatures
- D is a *class description function*, which maps class names to descriptions of class details, including instance variable declarations and methods.
- P is a *superclass function*, which maps each class name to a sequence of class names representing its immediate superclasses.

A *composition* is a tuple (R, Q), where

- R is a *composition rule*, and
- Q is a sequence of *components*, each of which can be either a subject or a subsidiary composition that includes its own composition rule.

As mentioned above, a great variety of composition rules is possible. Formalization of such rules is beyond the scope of this paper.

A subject-oriented *universe* is a tuple $U = (M, A)$, where

- M is a composition of subjects. Since nested composition is supported by the definition above, a universe can contain an arbitrarily large tree of subjects.
- $A = (OID, SA)$ is a *universe activation* consisting of a global set, OID, of object identifiers (called *oids*), and a set, SA, of *subject activations*, each of which specifies the state associated with each oid by each of the subjects in M. There may be more than one activation of a subject in the universe.

The key characteristic of a universe is that it is a single OID space; all the subject activations associate state with oids in the same space.[2] This concept of a shared space of unique identities for the objects can be somewhat limiting. For example, it hinders the simple inter-operation of a tool that sees a highway as a collection of unidentified lanes and a tool which just sees the lanes in a non-aggregated manner. However, as a simplifying concept it parallels the Knowledge-Based concept of *standard names* or *rigid designators*, which has been found to be a useful simplifying assumption in that domain as well [7].

A *subject activation*, often referred to just as an *activation*, models the actual data manipulated by a particular subject. It is a tuple $A = (S, T, C)$ where:

- S is the *subject*
- T is the *state function*, which maps oids to structures of addresses of instance variables manipulated by that subject. The state function is partial; not all subjects provide state corresponding to all oids.
- C is the *instance_of function*, which maps oids to class names. The class name corresponding to an oid is the name of the class that describes, from the point of view of this subject, the state and behavior associated with that oid. The instance_of function is also partial.

An *object* in a subject-oriented universe is really just an object identifier (oid), an element of the OID set. Through its T and C mappings, each subject can associate its own state and behavior with each oid. We deliberately avoid defining an object as the union of all this state and behavior.[3]

The following sections re-visit some of the discussion in the earlier Section 3.1, providing more detail and provision on certain topics.

6.1. Operation Invocation

2 In some implementations, of course, different subjects may employ different representations for the oids.

3 Of course, explicit dependencies can arise when one subject explicitly imports behavior from another to realize its function.

When an operation $((a, op, p))$ is invoked in a subject-oriented universe (M, A), where M = (R, Q), it might cause execution of methods in multiple subjects. The composition rules in M control what happens with a great degree of freedom. The entire tree Q of subsidiary compositions and subjects, in full detail, is potentially available for use by R.

As mentioned earlier, one such usage is a composition rule called *nesting*. Nesting treats compositions as scopes, and allows calls to propagate beyond scopes only if specified explicitly. For example:

1. Delegate dispatch of an operation to the lowest enclosing composition L, in the tree Q, of the subject whose activation made the call.
2. L performs dispatch as described above.
3. In addition, if explicitly specified by L, the operation call is "imported". This causes the next-higher composition to dispatch the operation also. By successive imports, dispatch can be propagated all the way to the top of the tree, but in their absence, it will be confined to a particular subtree.

6.2. Object Creation and Initialization

Within the subject model presented here, the steps involved in object creation and initialization are more precisely:

1. Allocation of an oid from the OID set.
2. For each subject activation, setting the value of the C function for the new oid, specifying the class to which it belongs in that activation.
3. For each subject activation, allocating space for the state information to be associated with that oid in that activation, and setting the value of the T function appropriately.
4. Placing the appropriate initial values in the storage allocated.

Allocation of oids happens globally, and is the responsibility of the universe activation, independent of all subjects. The remaining steps come under the control of the composition rule, so many approaches are possible.

The creation operation specifies the class to be created from the point of view of the activation requesting the creation. However the class to be created for the object in each other subject is one of the issues addressed by the composition rules. As mentioned earlier, the composition rule is also responsible for specifying whether any instance variables are to be shared across subjects. This specification affects the allocation of storage and the details of the T functions; sharing is accomplished by having the results of the T functions for different subject activations refer to common addresses.

6.3. State References

The addresses of the instance variables that a subject's activation can manipulate are obtained by means of the activation's T function. If T is undefined for the activation and oid, deferred creation and initialization must take place as described above.

Tighter encapsulation than that obtained from classical object-oriented models is possible because only a the subject's subset of the instance variables are accessible to each method.

7. Considerations in Implementing Efficient Subject-Oriented Support

7.1. Package Sharing Between Subjects

The division of processing into a multiplicity of subjects should not be presumed to imply high-overhead implementations in which subjects are implemented as separate processes or threads, in which operation invocation involves interpretive overheads, or in which each subject's representation for an object implies a separate invocation of memory allocation.

The problems involved in resolving these issues efficiently are similar to those faced in the implementation of inherited characteristics in conventional object-oriented languages. For example, in C++ the instance information storage requirements of independent class elements called superclasses are combined efficiently; similarly, in CLOS method combinators present potentially complex dispatching requirements that are solved efficiently.

In efficiently implemented Object-Oriented systems, these potential inefficiencies are resolved by a definition processor that uses information derived from the entire class-definition hierarchy to aggregate and optimize functions across class boundaries. For example, the C++ compiler uses the declaration of the entire class hierarchy to determine the total size

needed for an object to hold the instance variables in all of its subclasses.

Similar techniques can be applied to applications formed from a multiplicity of subjects. Processors like that used for OOTIS [10] employ a language for defining the subjects and their composition so that optimized allocations and linkages can be created. Using that technology, subjects can be composed into a single application process with competitive operation-call costs. The objects manipulated by the application process contain information for all of the subjects, but are allocated in single invocations of the underlying storage allocation mechanism.

7.2. Data Sharing Between Subjects

The division of processing into a multiplicity of subjects should not be presumed to imply drastically inefficient duplication of state information among subjects concerned with an object. As with the use of more conventional object-oriented technologies, when a single organizational provider is defining many classes or subjects, that provider may wish to exploit agreements about the instance variables in the implementation of the classes or subjects. These agreements may avoid the cost of duplicating state information or of indirect accesses. In C++, for example, subclasses or friends of a class have direct access to public and protected instance information.

These agreements take the form of a shared data model among subjects. Such shared data models are common in support frameworks for integrated applications, and mechanisms like the *schema definition set (SDS)* and *working schema* defined in PCTE [24] might be used in relating the instance variable definitions provided by different subjects. In general, data sharing is specified in composition rules, and can be implemented efficiently by subject compositors even if the subjects involved are not developed together.

7.3. Separate OID Spaces

In a distributed, heterogeneous subject-oriented environment, one might expect different activations to want to store their state in different repositories, with each repository having control over its own oids. This seems to be at odds with the requirement imposed by the subject-oriented model that all activations in a universe share the same *OID* space. However, these requirements can be reconciled by an implementation in which the global oids required by the model are implemented as mappings between the separate oids provided by the repositories.

7.4. Subject Compositors

Composition rules in the model are abstract specifications of the semantics of inter-subject interactions. A *subject compositor* is a tool that combines subjects in an environment according to a certain rule or class of rules. Whereas performance is not an issue when dealing with rules in the model itself, it is very much an issue for compositors. Practical composition rules must be capable of efficient implementation, especially in cases of frequent interaction, and subject compositors must be built to ensure high performance.

There is no need for a subject compositor to be present at run time. It could perform its work statically, generating code or stubs that realize inter-subject interaction according to the desired composition rule. An approach of this sort is likely to be necessary to achieve high performance.

A subject compositor provides definitions for:

1. the specification and defaulting rules for method combination
2. the strategies for matching interfaces and classes across subjects
3. the strategies for propagating interactions across compositions (scoping)
4. the packaging of subjects into threads, processes and nodes,
5. the packaging of subjects' object state information into databases and local-id spaces.

8. A Software Development Example Using Subjects

In this section we present an example to illustrate in concrete terms some of the key features of the subject-oriented approach. The example is from the domain of software development environments, and subjects are used to accomplish aspects of the tool integration that is recognized as an important need in such environments. Each tool is a subject, with its own class hierarchy and definitions for the classes. Figure 8 shows three tools to be eventually integrated together. However, since we wish to present the

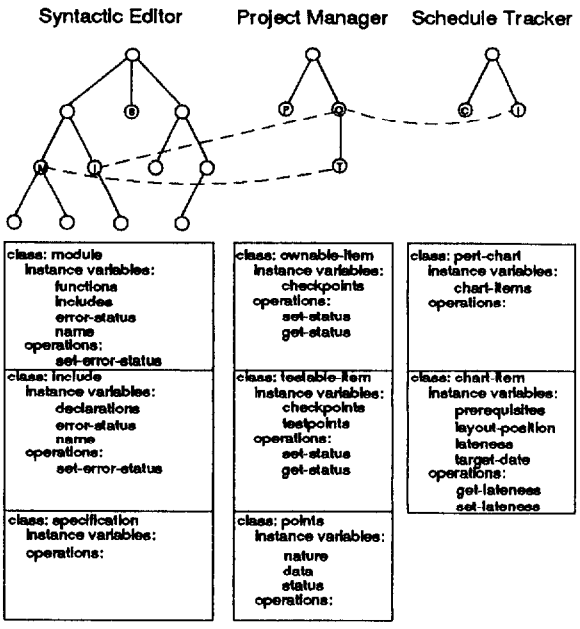

Figure 8 - A Software Development Example as Subjects

gradual enhancement of an environment as an ongoing process, we will introduce the tools one at a time.

The first subject around which the environment is built is an extended syntactic editor, capable of editing system structures, specifications, and code, like that described in [15]. Although the class structure for such an environment is complex, we will focus on aspects of three classes: modules, includes, and specifications. They are scattered about the class hierarchy because they do not have common implementation structures. Each of these classes has its own rich structure which we will not fully elaborate here, but it is likely that each will have state of its own, including relationships to objects containing more detail. For example, modules will probably have names and will be related both to their includes and to the code for the functions implemented in the module. Includes will probably have names as well, but also relationships to the declarations contained in the includes. Specifications may be unnamed primitive elements. For the our example, we will assume that this subject has been in use for some period of time, and the development repository has been populated with large numbers of objects of all sorts.

For illustration, we consider the editor's handling of error tracking in more detail. The editor checks on an ongoing basis the correctness of the modules it is editing. When errors are discovered or removed, the *set-error-status* operation is called to update the *error-status* state variable.

Consider, now, a hypothetical generic project management subject. This tool employs a class hierarchy containing ownable items, a subclass of ownable items called testable items, and a class of objects called *points* for recording test status. We wish to compose these two subjects to achieve a system whose syntactically edited modules and includes are managed by the generic project manager. To do so, we must specify a composition rule that controls how the subjects are to interact.

The class hierarchies of the two subjects are quite different, which is not surprising given the different nature of the subjects. The composition rule must define how objects in the already populated repository, and any new objects created by the editor, are to be mapped to the classes defined in the newly introduced subject. This involves matching of classes across subjects, as discussed in Section 5. Figure 8 indicates this mapping with broken lines across the class hierarchies in the several subjects. Both includes and modules are ownable items, but only the modules are testable. One of the ways that class matching among subjects differs from simple multiple inheritance is that in the syntactic editor modules and includes have no subclass/superclass relationship — having, in fact, different attributes and/or relationships. On the other hand, testables are a subclass of ownables. So different answers to a subclass/superclass test would be given in the project manager from those in the syntactic editor, even for the same objects.

One of the consequences of matching class module with class testable ownable is that any instance of module that existed before the subject composition took place is now also an instance of testable. As such, it has additional operations and instance variables, as defined in the project manager subject. Access to these operations and instance variables is available through the object's single oid, the same oid it had before the composition. However, the new instance variables are directly accessible only to code within the project manager subject. Subject-oriented programming thus includes the ability to expand the operations and state of existing instances.

The new state information belonging to the generic project manager needs to be initialized before use, even though this initialization must necessarily take place after creation of the original module object by the editor. The project manager subject is responsible for this initialization; it must compute the appropriate values, perhaps using information obtained from the human user and from other subjects in the composition via operation call.

The project manager and syntactic editor interact on the issue of error-status. The project manager maintains status information also, but is concerned with other forms of status than just syntax errors. It therefore provides more general *get-status* and *set-staus* operations. The desired semantics for the composition are that the *set-error-status* operation of the editor correspond to the *set-status* operation of the project manager, with parameter "syntax-error" indicating the kind of status. This correspondence is an example of interface matching, mentioned in Section 5.2. The composition rule either defines this correspondence explicitly, or specifies a strategy by which it can be determined. Once it has been established, any call on the editor's *set-error-status* operation results in execution of both the editor's implementation of this operation and execution of the project manager's implementation of *set-status*, with the appropriate additional parameter.

This situation involved matching different operations in the different subjects. In many cases, the subjects being combined have similar operational concepts, such as those being developed in connection with Case Communiqué[21], in which case interface matching is trivial. Subject-oriented programming supports both cases, and includes the ability to have multiple implementations from multiple subjects be executed in response to a single operation call.

Finally, we consider the introduction of a third subject — a schedule tracker. This is a PERT-chart-like application concerned with charts containing chart items, indicating planned and actual schedule information. By identifying the schedule tracker's chart items with the generic project manager's owned items, we allow the modules and includes to be organized into PERT-charts. As with the introduction of the general project manager, no magic applies here; the schedule dependencies must be constructed by the schedule tracker itself. We also identify behavior to be provided for *set-status* in terms of the PERT-chart's *set-lateness*.

The use of a single object, be it module or include, through the three subjects lends a unity of manipulation that makes it easier to discuss shared behavior, and makes it possible to add shared behavior without modification of the individual subjects. The existing objects acquire more state and behavior as more subjects are introduced into a composition, with the manner in which the state and behavior are combined being governed by the composition rule. The composition rule, or the compositor, also dictate how tightly-coupled the subjects should be: linked into a single program, distributed with each running on a separate machine, or various other options. Good compositor implementation will permit many such options with no change needed to the subjects themselves.

Space does not permit detailed discussion of how this example would be handled by conventional object-oriented programming. Suffice it to say that the compositions described, if not anticipated by the authors of the subjects, could not be achieved without source-code modifications to the individual subjects.

9. Related Work

9.1. Views of Objects

Part of the motivation behind subject-orientation arises from the need for functional-extension within an object paradigm. To satisfy this need without widespread change and recompilation requires that applications have their own views of data, and do not depend on global definitions.

In PCTE [24], each tool (application) runs within the context of a *working schema*, which specifies an ordered list of *schema definition sets* (*SDS*s). Each SDS defines a model of some of the data objects manipulated by the tool. An SDS can extend other SDSs, such as those describing the views of data seen by other tools. Extensions provide additional types of objects, and additional attributes of and relationships among both existing and new types. The subject-oriented models extends this general concept to the operations and methods associated with an object, and to allowing each subject to have its own classification hierarchy[11].

Shilling and Sweeney proposed an object-oriented paradigm exploiting *views*, in which an object is seen through a multiplicity of *interfaces* to the object [18]. Each interface determines the visibility and sharing of operations and instance variables. The subject-oriented approach separates the object interface supported by a subject from its implementation, relegating issues such as the sharing of state to a characterization of the subject's implementation. In addition, subject-orientation emphasizes the ability of different subjects to form different behavioral hierarchies over the objects, rather than consolidating them within a single class hierarchy.

In some ways, *aspects* of an object as defined by Richardson and Schwarz [17] can be modelled as different subjects providing their own classification and categorization of the object. In addition, subjects can provide coordinated aspects for a variety of object classes.

9.2. Routing Messages

The topic of routing messages from originators to participants within a suite of integrated tools is addressed in several generic settings:

Field[16]

Field uses message passing as a way to connect tools in a software development environment. Aimed primarily at the integration of existing tools, Field emphasizes the use of one-way notifications created by encapsulations of the tools. These notifications are broadcast through a message server that delivers them destinations that have registered interest by specifying patterns to be matched by the message and its arguments. This paradigm is being exploited by others [21] for both event and more general operation delivery. The subject-oriented model provides a more general setting in which message broadcasts can be seen as one kind of composition (broadcast), but provides a context for richer connection structures such as direct or nested connections can be established and intermixed.

Tools[10] and Toolies[6]

The "toolies" model emphasizes the direct routing of events that are automatically triggered by updates of data in a shared collection of data, and "tools" emphasize the intermix of events and requests. Both emphasize a reduction in size and monolithicity of the packages of software that are produced that can accrue from sharing a common shared model of structured data rather than bulk file manipulation. The subject-oriented approach continues and extends this direction, emphasizing both the need to retain private (subject-specific) information about shared data and the need to compose the elements with flexible packaging and dispatching strategies.

9.3. Composition Technology

The role of composition, including inheritance styles of composition, is increasing in importance, as is the attention being given to describing and constraining the compositions.

An *Object Request Broker* as defined by the Object Management Group [23] supports objects by routing messages and by interfacing with *Object Adapters* that actually support the implementations. Object Request Brokers provide support for a uniform representation of oids.[4]. Their dispatching mechanisms are based on an object registry approach. Subject compositions, on the other hand, perform registry of packaged behavior for a collection of classes at the same time. In addition, the subject concept allows more than one subject to provide state and behavior for the same object. No requirement exists that the oid representation be common across all subjects in a composition. The CORBA specification provides great latitude for implementations. Within this latitude, some manufacturers have provided Object Request Brokers not well suited to support of a subject-oriented methodology, while others have provided more flexible realizations in anticipation of the needs of complex environments. The Object Request Broker is one example of a compositor, but more powerful ones exist as well.

Class composition, as in "Jigsaw" [1], separates inheritance from the troika of encapsulation, polymorphism, and inheritance, in effect depicting inheritance as one of several operations by which classes can be composed. *Hierarchy composition* [4, 14, 19] extends this concept to collections of classes. The subject-oriented approach goes a step further by removing the restriction of having a shared definition of the inheritance hierarchy.

[4] [23] page 34

The *frameworks* model for object-oriented design [20] emphasizes the use of abstract classes which are extended to form concrete classes by either the implementation or re-implementation of some of the operations they define. Well-crafted frameworks greatly facilitate application development in their particular domains, but they are subject to two restrictions:

- They can be specialized or extended conveniently only in those areas specifically designed to be extended. Yet the framework designer cannot anticipate all future extension needs.
- Changing or further subclassing the concrete classes leads to invalidation of existing objects. This is especially important when the objects exist in a persistent, shared store or repository.

The subject-oriented approach overcomes these restrictions. Much of the work done on methodologies for developing and describing frameworks applies to subject-oriented programming as well.

Contracts [12] provide an abstract way to characterize the behavioral interdependencies among a collection of objects. Axiomatic specifications establish what activities in one object are expected to lead to what activities in other objects. Within the subject-oriented model, contracts can be used to characterize the interdependencies among the objects making up a subject. The concept can also be usefully extended to characterizing behavioral interdependencies among multiple subjects.

Where contracts provide an axiomatic characterization describing and constraining the interaction of objects, the *law-governed systems* approach specifies the detailed semantics of interactions by means of a *law* [13]. As with the contracts model, the same similes can be applied to messages between subjects about an object as can be applied to messages between objects within a subject. In fact, the composition rule, R, can be seen as the "law" of a subject composition, (R, Q).

10. Conclusion

The "software chip" is one of the Holy Grails of software development. (Holy Grails are somewhat larger than silver bullets). Objects and classes have been seen as the software chip [3], but a class is too small a package of functionality to play this role. In a sense, objects and classes are more like circuits, or what the hardware designers call "macros". We believe that subjects are far more likely to play the role of software chips as the next higher-order software building blocks in the sequence of procedure, class, and subject. In many respects, a **subject** can be viewed as the software equivalent of the hardware *micro-chip*, with the **subject compositor** providing a way of manufacturing the software equivalent of circuit boards.

The subject-oriented approach brings into focus and provides a model within which to explore a number of important issues associated with software composition:

- Composition rules and compositors
- Object creation/initialization and finalization/deletion protocols
- Interface and class matching, leading eventually to matching for applications that use drastically different models of common domains.
- Implementation issues, including efficiency, distribution and multiple *OID* spaces.

We expect the development of subject compositors to be an important area of research and development over the next few years. The development of a subject compositor along the lines of the one described for OOTIS [10] is proceeding at IBM's T. J. Watson Research Center to support the use of a subject-oriented style and to further the exploration of this domain.

References

[1] Gilad Bracha, Gary Lindstrom, "Modularity meets Inheritance", Proceedings of the 1992 International Conference on Computer Languages, (Oakland), pp. 282-290, April 1992.

[2] Peter Coad and Edward Yourdon, "Object-oriented Design", Prentice-Hall, Inc., (Englewood Cliffs), 1991.

[3] Brad J. Cox, "Object-oriented Programming — An Evolutionary Approach", Addison-Wesley Inc., (Reading, Mass.), 1986.

[4] William R. Cook, *A Denotational Semantics of Inheritance*, PhD. thesis, Brown University, 1989.

[5] William R. Cook, "Interfaces and Specifications for the Smalltalk-80 Collection Classes", Proceedings of the Conference on Object-Oriented

Programming: Systems, Languages, and Applications, (Vancouver), ACM, October 1992.

[6] David Garlan, Gail Kaiser, David Notkin, "Using Tool Abstraction to Compose Systems", IEEE Computer, pp. 30-38, June 1992.

[7] M. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan-Kaufmann, 1987.

[8] William Harrison and Harold Ossher, "Extension-by-addition: Building extensible software", IBM Research Report RC 16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 1990.

[9] William Harrison and Harold Ossher, "CLORIS: A clustered object-relational information store", Proceedings of the Experts Meeting on Object Oriented Computed Research and Development, Information Technology Research Centre, Toronto, May 1991.

[10] William Harrison, Mansour Kavianpour, and Harold Ossher, "Integrating Coarse-grained and Fine-grained Tool Integration", Proceedings of Fifth International Workshop on Computer-Aided Software Engineering, July 1992.

[11] William Harrison, Harold Ossher, and Mansour Kavianpour, "PCTE SDS's For Modelling OOTIS Control Integration", Proceedings of the PCTE'93 Conference, PCTE Interface Management Board, November 1993.

[12] Richard Helm, Ian Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (Vancouver), ACM, October 1990.

[13]Naftaly Minsky and David Rozenshtein, "A Law-Based Approach to Object-Oriented Programming", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, ACM, October 1987.

[14] Harold Ossher and William Harrison, "Combination of Inheritance Hierarchies", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (Vancouver), ACM, October 1992.

[15] Harold Ossher and William Harrison, "Support for Change in RPDE^3", Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments, pp. 218-228, Irvine CA, December 1990

[16] S. Reiss, "Connecting Tools Using Message Passing in the Field Environment", IEEE Software, pp. 57-66, July 1990.

[17] Joel Richardson and Peter Schwarz, "Aspects: Extending Objects to Support Multiple Independent Roles", Proceedings of the 1991 ACM SIGMOD Conference, May 1991, Denver CO, pp. 298-307.

[18] John Shilling and Peter Sweeney, "Three steps to views: Extending the object-oriented paradigm", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (New Orleans), pp. 353-361, ACM, October 1989.

[19] Alan Wills, "Capsules and types in Fresco", In Pierre America (ed), Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP '91), Springer, 1991.

[20] Rebecca Wirfs-Brock and Ralph Johnson, "Current Research in Object-Oriented Design", Communications of the ACM, pp. 104-124, ACM, September 1990.

[21] –, "Achieving Agreement", Case Communiqué, 3404 Harmony Road, Fort Collins CO, June 1992

[22] –, Object Management Architecture Guide, OMG Document 92.11.1, Object Management Group, September 1992.

[23] –, Common Object Request Broker Architecture and Specification, OMG Document 91.12.1, Object Management Group, December, 1991.

[24] –, Portable Common Tool Environment (PCTE), Standard ECMA-149, European Computer Manufacturers Association, December 1990.