Using Types and Inheritance in Object-Oriented Languages

Daniel C. Halbert Patrick D. O'Brien

Digital Equipment Corporation, Object-Based Systems Group
HLO 2-3/M08, 77 Reed Road
Hudson, Massachusetts 01749, USA

Abstract

If the object-oriented style of programming hopes to live up to its potential as an improved methodology for software programming, a clear understanding of how to use types and inheritance is essential. Our experiences with using object-oriented languages and teaching object-oriented techniques to other programmers have shown that effective use of types and inheritance may be problematic. There are no concrete guidelines to assist programmers, and the existing aphorisms often create interpretation problems for novice object-oriented programmers. In this paper we look at how types, subtyping, and inheritance are used in object-oriented languages. We discuss the different ways that types and type hierarchies can be used to structure programs. We illustrate appropriate use of these concepts through examples and develop guidelines to assist programmers in using the object-oriented methodology effectively.

1 Introduction

The object-oriented style of programming offers an improved methodology for software development. Types[1], type hierarchies, and inheritance are fundamental concepts of object-oriented programming, and mastery of these concepts is essential if the benefits of the object-oriented style of programming are to be realized. We will describe the different ways types and inheritance can be used to structure programs and present some guidelines for proper use.

The Object-Based Systems Group at Digital has developed an object-oriented language called Trellis/Owl[2] [Schaffert *et al*]. Over the past three years we have been using the Trellis/Owl language to build several large applications, including the Trellis programming environment, a window management system, and a project manager's workstation. We have also introduced object-oriented programming to several groups within our company. Our experience has shown that the object-oriented methodology increases programmer productivity by enhancing the maintainability, extensibility, and reusability of software.

To achieve this increase in productivity, types and inheritance must be used effectively. Types are the major organizing principle in object-oriented programming, and inheritance enables one to specify important relationships between types. But using these concepts appropriately is a difficult skill to master. Furthermore, the lack of existing guidelines to assist programmers exacerbates the problem. We believe that guidelines and heuristics need to be developed to aid both novice and experienced programmers[3]. In this paper we will discuss some problems object-oriented programmers encounter and develop some guidelines they can use in programming.

Different languages provide types and inheritance in different ways. Consequently, when these concepts are discussed, the reasons for how and why they are used are often couched in terms of a particular implementation. We will concentrate on types and inheritance in general, and not just how they can be used in one particular language.

In the next section we review types and inheritance as they relate to object-oriented programming. Section 3 discusses the use of types in more detail and points out some common difficulties that novices encounter. It also presents some guidelines for structuring code using types. Section 4 discusses the uses of inheritance and the different ways that inheritance can be used with types to structure code. It also describes some uses that should be avoided and presents more desirable alternatives. Section 5 presents a

[1] In this paper, we use the words *type* and *class* synonymously. The same holds for *fields* and *instance variables*, *operations* and *methods*, *operation names* and *selectors*, and *procedure calls* and *messages*.

[2] Trellis is a trademark of Digital Equipment Corporation.

[3] Novice and experienced refer to the user's familiarity with the object-oriented style of programming. Most of our novice users were experienced programmers.

detailed example that illustrates some of the guidelines and principles developed in this paper. We conclude with a summary and recommendations in section 6.

2 Foundations

In object-oriented programming, computation proceeds by objects sending messages amongst themselves. Objects encapsulate both state and behavior. Types capture the common characteristics of objects and can be related hierarchically through inheritance. Inheritance enables types to share various characteristics. Abstract types, which are used to group characteristics and do not have any instances, are useful when utilizing inheritance.

2.1 Types

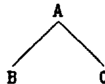
Object-oriented programming combines descriptions of data and procedures within a single entity called an *object*. The data in an object consists of *fields* which may reference other objects. The *operations* associated with an object collectively characterize its *behavior*. Every object has a *type* and the object is referred to as an *instance* of that type. The type mechanism provides a means of classifying similar objects and capturing the common characteristics of those objects. An object's type determines the set of operations that can be applied to it.

Object-oriented programming differs from conventional *control-based* programming. It encourages the programmer to concentrate on the data to be manipulated rather than the code that does the manipulation. This emphasis helps to maintain a balance between data and control flow. Since types are commonly used to model entities in the application domain, they provide a natural basis for modularization in a program. In control-based programming, one is forced to modularize programs based on procedural criteria. This often leads to program structures that are radically different from the structures in the application domain. Since types model application entities, the structures in object-oriented programs can more closely resemble the structures in the application domain.

Types are the major organizing principle in object-oriented programming. A program with a well-designed type structure will minimize and localize the interdependencies among types, thus enhancing maintainability and extensibility.

2.2 Subtyping and inheritance

An object-oriented language often provides a *subtyping* mechanism in its type system. One type may be a subtype of another, with the implication that if B is a subtype of A, an object of type B may be used wherever an object of type A may be used. In other words, objects of type B are also of type A. Some languages permit a subtype to have multiple parent types (*supertypes*). The relationships between the types define a *type hierarchy*, which can be drawn as a graph:



In this example, both B and C are subtypes of A.

Intimately connected with subtyping is the concept of *inheritance*. A subtype may share or *inherit* various characteristics of its supertype(s). The subtype in turn may pass on its own or its inherited characteristics to its subtypes. The characteristics that can be inherited usually include the storage representation of the supertype and the operations provided by the supertype.

Frequently, the inherited characteristics may be changed, or *overridden*. A subtype may respecify the storage representation or give new definitions for operations inherited from a supertype. Different languages allow varying degrees of control over what can be inherited and what can be overridden. For instance, in Smalltalk-80[4] [Goldberg & Robson], operations defined in a supertype are always inherited and may be reimplemented in a subtype. The storage representation of an instance, as defined in the supertype, is

[4]Smalltalk-80 is a trademark of Xerox Corporation.

also inherited, and may be augmented by defining additional storage fields. By comparison, in Trellis/Owl, operations and fields may be declared to be *public* or *private*. Private declarations are visible within but not outside the type, and, therefore, a subtype cannot use a supertype's private operations and fields.

The definition of an operation consists of two pieces: the *interface* and the *implementation*. The interface describes the external characteristics of the operation, such as the operation name, the type of the return value, and what arguments the operation takes. The implementation contains the actual code for the operation.

Although most languages allow an existing implementation of an operation to be overridden by a subtype, they often restrict how an interface can be respecified. For example, in Trellis/Owl, which is a *type-safe* language, an interface can be overridden only in specific ways. These restrictions guarantee that the type checking implied by the interface declarations can always be done at compile-time.

2.3 Abstract types

An abstract type is a type for which no instances are created. Abstract types characterize common behavior and pass on those characteristics to their subtypes. Often, an abstract type just describes the interface to the operations of the type, or is a partial implementation of those operations. The complete implementation is left to one or more subtypes. Other times, an abstract type describes a complete implementation of some data type, but the type is useful only when it is combined with other types in a new subtype. Even if a language does not distinguish in its syntax between abstract and non-abstract types, programmers will often use abstract types and indicate so through coding conventions.

In Smalltalk parlance, one often says that a class obeys a certain *protocol*. This means that the class has defined methods for a particular set of selectors, and that these methods obey a certain semantics. For example, instances of `Integer` and `Point` both obey comparison protocol; that is, both classes provide methods for <, >, etc. A class can obey more than one protocol; for instance, `Integer` will also obey arithmetic protocol by providing methods for +, -, and so forth.

Although the term "protocol" is used informally, the idea can be crystallized and made formal through the use of abstract types. An abstract type that contains operation interfaces, but no implementations, defines a protocol. If some implementations are included, the abstract type defines a protocol with default implementations. Several protocols can be combined by inheriting definitions from multiple supertypes.

Some languages can guarantee that a type implements a certain protocol. For instance, Trellis/Owl checks at compile-time for operations that are unimplemented in a non-abstract subtype. Therefore, one can be certain that a subtype has completely implemented an abstract supertype. In Smalltalk-80 this check is done only by a run-time convention: operations in an abstract supertype that do not supply a useful default implementation instead provide an implementation that generates a run-time error. The error indicates that the operation should have been implemented in a subtype.

We have now introduced the concepts of types, which capture common characteristics of objects; inheritance, which enables types to share various characteristics; and abstract types, which group behavior but have no instances. In the next section we begin looking at how types can be used to structure code.

3 Using types

To the novice user, designing a program using types can be a formidable task. A lack of clear guidelines has compounded the problem, and the fuzzy principles which have been proposed often lead to confusion and misinterpretation. A clearer understanding of how to use types will result in better structured programs and also shorten the time it takes to become adept at using the object-oriented style of programming. We will first discuss a few misconceptions often held by beginning object-oriented programmers, and then will develop some general guidelines for using types effectively.

3.1 Process versus data

As we mentioned earlier, object-oriented programming encourages the programmer to concentrate on data rather than procedures. This emphasis sometimes presents problems for novice object-oriented programmers. Novice programmers generally understand *data abstraction*: that is, they understand grouping data and the operations on that data into a type. But they then believe that modelling a process as an object

goes against the grain of the object-oriented paradigm. By a process we are referring to a procedural mechanism as opposed to a concrete entity. For example, payroll is probably thought of as a process, but employees are considered concrete entities. Should payroll be modelled as an operation on employee entities, or should it be modelled as an independent process defined as a separate type? Both approaches seem plausible: choosing one or the other will depend on the details of payroll and employees.

As another example, consider a lexical scanner which is a front-end process for a parser. The scanner accepts strings, and, according to a set of rules, constructs tokens which it passes to the parser. We could define the scanner process as an operation on the type `String`, since it operates on strings and generates tokens from the strings. But to consider the scanner as an operation on the type `String` does not capture the desired abstraction, nor does it improve the structure of the program. A better design would be to create a distinct type for the scanner: this packages the abstraction properly by focusing on the scanner itself and not the strings it takes as input.

Processes can and should be modelled as separate types. This is not contrary to the object-oriented style of programming. Later in this section we will discuss reasons to separate behavior into distinct types and reasons to keep behavior within an existing type.

3.2 Correspondence to the application domain

Novices encounter another difficulty, related to the common aphorism that objects should be used to model entities in the application domain. This is a useful guideline, but novice users sometimes interpret this guideline to mean that the *only* types that should exist are ones that model the external view of the application.

A program can be viewed as a model of the real world. In moving from real world to computer system, entities are introduced that do not necessarily correspond to visible real world entities. There is nothing wrong with this: it simply reflects deeper levels of implementation. The higher levels of the program still model the application closely.

For example, in a window management system there is often an intangible object called the "window manager." The window manager directs I/O traffic between the physical display device and the windows the user sees on the screen. But the concept of a window manager is artificial, and was introduced only to help us code the application. It does not intuitively correspond to a real world entity; it is merely an artifact of how we model a portion of our application.

3.3 What's in a type?

Code written in an object-oriented language is normally associated with a particular type. When a programmer adds new code, there are two alternatives: the new code can be added to an existing type or a new type can be defined. Usually the choice is obvious, but sometimes it is not. Making this decision correctly, however, is very important: it affects the structure of the program and has important consequences for further evolution of the program.

To decide where to add new code, one must consider whether the behavior being added is better modelled as one or more operations on an existing type, or as an independent concept which can stand on its own. If a new type is created, would the new type be a meaningful abstraction that groups behavior in a useful way? For example, an operation to clear a window on the screen would naturally be provided in the type `Window` itself. But suppose we want to draw a clock in a window. Since clocks are not germane to windows, and since a clock can be thought of as an independent and useful concept, we would probably create a new type, `Clock`, to capture this specialized behavior.

Now, consider the process of enrolling a student in a course. Should this be an operation on the student or the course? The choice depends on our perspective. Another alternative is to create an `Enrollment` type to model the relationship between students and the courses they are taking. This would be important if we later wanted the enrollment relationship to capture other information, such as the grade a particular student received. Enrollment is related to other types (`Student` and `Course`), but it can be modelled as an independent concept and thus can stand on its own as a separate type.

There is no firm rule that can be applied in all cases to decide where to add behavior. But we can suggest criteria that should be considered when associating new behavior with types. The programmer should look at certain aspects of the behavior: its reusability, complexity, applicability, and its knowledge of representation details. Each of these criteria will be examined in detail.

If the behavior can be shared by several types, we suggest creating a separate type. This is analogous to factoring out code into procedures that can be reused by other procedures. For example, consider the concept of a rectangle. Rectangles are useful for describing the boundaries of windows, and are also useful for describing graphical shapes that can be displayed inside windows. By factoring out the concept of a rectangle into a separate type, we can use the abstraction in both windows and graphical shapes. If we had imbedded the behavior of rectangles within `Window`, we would have to duplicate the code within graphical shapes. Reusability is an important consideration when deciding how to structure code. Structuring types for reusability provides leverage when adding new program functionality. This is sometimes referred to as the "building block" approach to system development.

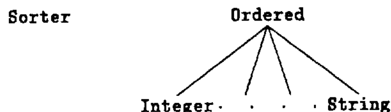
If the algorithm for implementing some particular new behavior is quite complex, we might factor out the code into a separate type, to avoid cluttering up the original type. For instance, calculating the area of a rectangle is a simple and straightforward algorithm, so we would put this operation in type `Rectangle` itself. On the other hand, text hyphenation and justification is a much more complex task, so we would create a new type to do text formatting, instead of putting the behavior in the text type.

One should also consider the general applicability of the behavior: in other words, will most users of the type want to use the behavior? The more specialized the behavior is, the more sensible it is to factor it out into a separate type. For example, the capability of converting strings to uppercase is generally useful, and would likely be included in the `String` type. Conjugating a regular French verb is quite specialized, and even though it is a simple operation on a string, it would make more sense to put the behavior into a different type.

Lastly, one should consider how much the new code needs to know about the representation details of the type. If the behavior needs to know a great deal, it should be included in the type. For instance, an operation that makes a copy of a given object should be implemented in the object's type. When making a copy of an object, one has to know whether the object and its copy can share the values of various fields, or whether the copy will have its own copies of certain field values. Thus, the code implementing the copy operation will need to have access to the representation and should therefore be associated with the type. We should point out that if a behavior does not need to know very much about representation details, this does not necessarily imply that we should create a new type: it merely means that it is less important for the operation to be implemented in the existing type. In all cases, the programmer should try to minimize and localize interdependencies between types.

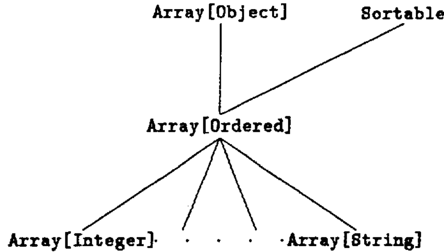
These criteria must be used in concert to decide where to place new behavior. In many cases applying the different criteria will give conflicting results. One should weigh each criterion individually, and then decide which criteria take priority in a particular circumstance.

Consider a more involved example. Suppose we want to sort arrays of objects. We can either define this behavior as an operation on arrays or create a new type. Since sorting is a complex operation, and since it does not need to know very much about the representation of the objects being sorted, we decide to create a type that is a *sorter* object. The primary operation of the sorter would be `sort`, which would take an array and return a sorted version of it. We would also define an abstract type `Ordered` which would define operations like `greater_than`, `less_than`, and `equal`. The `sort` operation would take arrays of `Ordered` objects. Therefore, types that we want to sort, like strings and integers, are declared to be subtypes of `Ordered`. The `Ordered` type is created because it groups behavior that is required by the `sort` operation.



The applicability criterion suggests that the sort behavior should be associated with the array type, since sorting is a generally useful operation which most users of array would want to use. We chose to stress the complexity criterion and thus factored out the sort behavior. A shortcoming of this approach is that the sort algorithm cannot be easily modified based on the type of object being sorted. For example, we might want to sort arrays of integers and arrays of strings using different algorithms. To do this we would have to write code within the sort type to choose a particular algorithm based on the type of object passed to it.

A better way to accomplish this would be to introduce another abstract type, Sortable. The type Array[Ordered][5] would be a subtype of Sortable. Sortable would define the operation sort, and this operation would be inherited by all subtypes of Array[Ordered], such as Array[Integer] and Array[String]. The default implementation of sort would be similar to the sort operation on the Sorter type described earlier.



The advantage of the second approach is that the property of sortability is associated with the objects that possess this property: arrays of Ordered objects. Sortability is not separated out somewhere else. Moreover, each type that inherits the properties of Sortable can specialize the behavior and thus replace the default sort algorithm if desired. By factoring out the sort code into Sortable, we adhere to the complexity rule. But by using inheritance, we can factor the sort behavior back into the array type, and thus adhere to the applicability rule.

This example shows one of several ways that inheritance can be used to improve the structure of code. In the next section, we will take a closer look at the effect that inheritance has on decisions for grouping behavior within types.

4 Using inheritance

In this section, we will discuss how the programmer can turn concepts and implementations into type hierarchies. We will first describe the general design methodology that subtyping promotes, and then cover the different ways of using subtyping.

4.1 Subtyping as a design methodology

One of the major problems in designing and implementing large software systems is organizing the complex relationships that exist among objects in the application domain. Types provide natural criteria for modularization, and thus help shape the strategy for decomposing a program into modules. Subtyping enhances the type system's ability to cope with this inherent complexity. Subtyping affects the strategy used in determining the overall organization of programs, in contrast to techniques such as structured programming, which affect the style of writing small segments of code.

The seminal idea of subtyping is that a program can be constructed by first modelling, in terms of types, the most general concepts in the application domain, and then proceeding to deal with special cases through more specialized subtypes. Subtyping thus provides a general design methodology based on "stepwise refinement by specialization." This methodology can be compared with the well-known methodology of "stepwise refinement by decomposition" [Wirth].

Stepwise refinement by decomposition can be applied to both procedural and data abstractions. At each level of abstraction, the problem is subdivided into smaller components which are in turn subdivided. For procedural abstraction this subdividing process is achieved by creating procedures which call lower-level procedures. The program ends up as a hierarchy of procedures.

When decomposition is used to develop layers of data abstractions, the programmer creates a high-level data type and describes the operations on it. Then the programmer breaks down the top-level data type into its constituent components, breaks down each of those, and so forth. For example, consider a payroll database as a top-level data type. When broken down, one component of the payroll type might

[5] Arrays are defined as *parameterized* types[Schaffert *et al*].

be a collection of employee records. Each employee record could be broken down into fields describing the number of hours worked, the hourly wage, the deductions requested, and so forth. Together, these data types would form an aggregation hierarchy that describes the structure of the top-level payroll data type.

Some early uses of stepwise decomposition did not use procedures to preserve the series of abstractions that was created. Instead, the components of each decomposition were substituted in-line for the higher-level abstraction they implemented. Thus, the final program flattened out the design: it did not preserve the hierarchy of abstractions through which it was created. The result was that maintaining or enhancing a program developed by decomposition was not necessarily simpler than it would have been for a program developed in any other way [Shaw].

Similarly, before the introduction of data abstraction into programming languages, abstract data types were also flattened out. Complex data types were explicitly encoded in terms of simple ones, and only the encodings were visible, not the abstractions. For instance, records with named components were frequently encoded as simple arrays with numbered elements. The introduction of programmer-defined data types helped a great deal by explicitly presenting levels of data abstraction.

Coding a program with data abstractions, but without subtyping, can result in a flattened data type design along another dimension, namely the subtype hierarchy. Natural progressions from general data types to more specialized ones are obscured. Frequently, the programmer ends up overloading a type to represent more than one kind of object. In a language with subtyping, the programmer can factor out the common behavior in two or more types into a common supertype. But without subtyping the programmer must maintain several types with very similar code, or instead must combine several similar types into one, keeping track of the variants using some kind of tag.

For instance, in the payroll example given above, there may be hourly, weekly, and salaried employees whose pay information is quite different, but who otherwise have many properties in common, such as age, badge number, and so forth. In a language with subtyping, one could create an `Employee` type to hold the common information, and create subtypes for each category of employee. Without subtyping, the hierarchy would become encoded in a tag. If there were further divisions, such as temporary and permanent hourly employees, more tags would be needed.

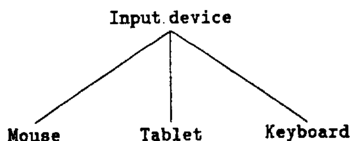
Thus the lack of subtyping makes the data types harder to understand and obscures the natural hierarchy. When subtyping is present, it can be used to implement and preserve data abstraction hierarchies, just as procedures can preserve procedural hierarchies.

With decomposition, one rarely designs strictly in a top-down fashion, even though the methodology in its purest form calls for it. Sometimes one codes certain lower levels of the program first, to see how their design will affect higher-level aspects of the program, such as what data must be passed between parts of the program. Similarly, one does not always design types in a type hierarchy by starting with a supertype and then creating the subtypes. Frequently, the programmer creates several seemingly disparate types, realizes they are related, and then factors out their common characteristics into one or more supertypes. In either case, there is nothing wrong with this bottom-up or middle-out design; several passes up and down are usually required to produce a complete and correct program design.

In the next section we begin to look at different reasons for using subtyping.

4.2 Subtyping for specialization

As we implied above, subtyping is most frequently used for modelling conceptual hierarchies. Sometimes the hierarchy produced is a model of the real world. For instance, `Jet` and `Turboprop` might be subtypes of `Airplane`. `Employee` might be a subtype of `Person`. `Input_device` might be the common supertype for `Mouse`, `Tablet`, and `Keyboard`:



Other times, the programmer will use type hierarchies to model intangible concepts that are nevertheless external to the program. For instance, a hierarchy might model the relationship between algebraic

groups, rings, and fields. Finally, subtyping can be used to describe concepts that are particular to the program itself. For example, the supertype Command might describe behavior common to Batch_command and Interactive_command.

In all these cases, a subtype describes a type that is more specific than that of the supertype. Thus the subtype is a *specialization* of the supertype. Conversely, one could say that a supertype is the *generalization* of its several subtypes. Which term is used depends on one's point of view. A programmer could realize that several types share common characteristics, and factor out the commonality to produce a more general type which would be a supertype. Or, the programmer might need a more specific description of something, and so would create a subtype to specialize an existing type.

Another way of viewing specialization is to think of types as describing sets of values. The supertype defines a possible set of objects, and the subtype restricts the definition in order to define a subset of that set. Thus the term *restriction* could be used to describe specialization.

The supertype in a specialization hierarchy is frequently, but not always, an abstract type. The supertype may have factored out the common characteristics of its subtypes, but standing by itself it is an incomplete description, incapable of describing any useful object. Subtypes will add behavior that turns the abstract type into something useful. For example, the Input_device type described above would be incomplete, and so this type would undoubtedly be abstract.

A supertype is also usually abstract if the subtypes partition the instance space of the supertype. For example, in the Airplane example given above, if all kinds of airplanes of interest can be described using one subtype or another (e.g. Jet or Turboprop), there is no need to create instances of type Airplane itself, even though Airplane might be a perfectly reasonable non-abstract type. If a new kind of airplane needs to be described, a new subtype can be added.

4.3 Subtyping for implementation

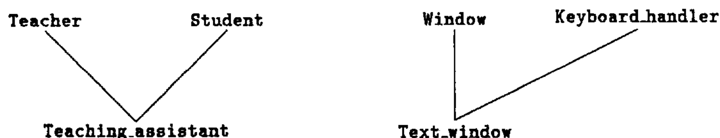
Another common use for subtyping is to guarantee that types present a certain interface, that is, they obey a certain protocol. The programmer can define an abstract supertype that specifies certain operations. Many of the operations may be unimplemented. The programmer will then define several non-abstract subtypes that implement the behavior specified in the abstract supertype. For instance, the programmer may define an abstract type, Dictionary, and then implement this type in two ways, say, Hash_table and Property_list.

Here, the subtypes are not specializations of an existing supertype, but are *realizations* of the abstract supertype. The type hierarchy in this case reflects implementation only: there is no interface change from supertype to subtype.

4.4 Subtyping for combination

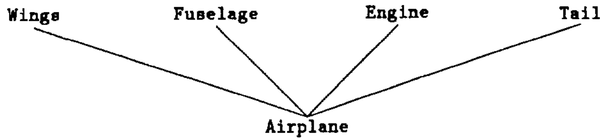
Until now we have been talking about types with single immediate supertypes. When a subtype has multiple supertypes, they combine to form a type with the behavior of all the supertypes.

In some instances, types of relatively equal importance are combined. For example, one might define a type Teaching_assistant with supertypes Teacher and Student. Here the supertypes seem to add together equally to form the subtype. In other cases, one combines a primary supertype with other supertypes that add ancillary functionality. For example, consider a type Text_window, which is a subtype of Window. Text_window is also a subtype of Keyboard_handler, which adds operations to permit keyboard input to the window. Here we would consider Window to be the main supertype. We might even draw these two hierarchies differently:



As with other kinds of subtyping, the supertypes may or may not include complete implementations for the types they define. In the examples above, Teacher and Student probably include complete implementations, but Keyboard handler may be an abstract type that just specifies the interface for the type-in operations, and does not provide any implementations. Thus, as mentioned before, a supertype can be used to guarantee that a subtype provides a certain interface; multiple supertypes just make this capability easier.

Since multiple supertypes can be used to merge implementations together, programmers are tempted to use supertyping even when it is inappropriate. For instance, we could create a type Airplane out of the supertypes Wings, Fuselage, Engine, and Tail:



But a conceptual error is being made here. An airplane is *composed* of these parts: it is not the sum of the behavior of these parts. Multiple supertypes allow us to view an instance of a subtype as being an instance of any of its several supertypes. But we are not trying to view an airplane in such a way: we don't really want to think of an airplane as a more sophisticated way of viewing an engine or a fuselage.

We should not use subtyping for accomplishing this kind of aggregation. We already have an excellent mechanism: object fields. In our example, Airplane should have fields that contain the wings, fuselage, engine, and tail objects. Achieving aggregation through fields instead of through inheritance also allows more than one instance of a particular type of object, since a subtype cannot inherit a supertype more than once. If our airplane had more than one engine, or if we wanted Wings to be broken down into two instances of Wing, we could not use subtyping for aggregation.

To take another example, suppose we would like to keep track of the order in which windows are created. It would make sense to keep the windows on a linked list. We could define a type Linked list node, which defines a "next on the list" field, and make it one of the supertypes of Window. Windows would then have as part of their behavior the ability to be linked together in a list. But we do not necessarily want to view a window sometimes as a window and sometimes as a node in a linked list. Instead, we should make the linked list node a field defined in Window, or even better, we should keep a completely separate list of windows. Then, we could also put a window on more than one linked list, since the list behavior would not have been added by inheritance.

Nevertheless, it is often hard to decide when to combine types using supertyping, and when to use aggregation. For example, in our applications, we have coded the type Rectangle and the type Window. Rectangle implements operations such as overlaps?, includes_point?, upper_left, and so forth. Now, since windows are rectangular areas on the screen, should Window be a subtype of Rectangle or should it have a field that is a Rectangle? Choices like this are a toss-up. We have chosen one and then later the other, based on how Window was used by other code, and not based on anything inherent only to Rectangle and Window.

4.5 Nonstandard subtyping

In the uses for subtyping we have discussed so far, concepts from the abstract design of the program are closely reflected in the type hierarchy. We will now discuss some uses for subtyping that are driven mostly by implementation, not design considerations.

4.5.1 Subtyping for generalization

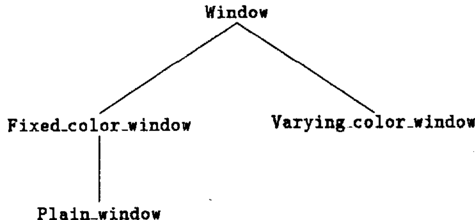
Subtyping for generalization is the opposite of subtyping for specialization. Here, a subtype extends a supertype to create a more general kind of object. For instance, Ellipse could be implemented as a subtype of Circle. A circle would be described by a single center point (focus) and a radius. An ellipse can be described by adding another focus. Or consider a Plain_window, which displays its contents on a

simple white background. The programmer could create a subtype of Plain_window, Color_window, which allows the background color to be other than white, by adding an additional field to store the color, and by changing the inherited window display code so the background was drawn in that color.

In these examples, the conceptual hierarchy we would expect does not correspond to the actual type hierarchy. For instance, circles are special cases of ellipses, not the other way around. Nevertheless, the hierarchy specified by the programmer allows useful implementations to be inherited from supertype to subtype, and can save much recoding.

The programmer should be wary of disassociating the conceptual and implementation hierarchies in this way, since it goes against intuition, and may make the subtypes unduly dependent on the implementation of the supertype. If the hierarchies are dissimilar solely for reasons of implementation expediency, the programmer should consider whether there is a way to rearrange the implementation hierarchy to make it match the conceptual dependencies. For instance, Circle, above, might better be described as an Ellipse in which the two foci are identical.

On the other hand, casually trying to match the conceptual and implementation hierarchies can cause other problems. In the window example above, one alternative is to make Plain_window be a subtype of Color_window, with the background color restricted always to be white. This causes problems, however, if the color of a Color_window can be changed at runtime, since an attempt to change the color of a Plain_window should be detected as an error by code in Plain_window. We can avoid the possibility of such a runtime error by factoring out the common behavior of windows into an abstract type Window, and then creating the new types Fixed_color_window and Varying_color_window, with Plain_window being a subtype of Fixed_color_window:



In some cases, this sort of restructuring may be more trouble than it is worth. Keeping the original implementation hierarchy is easier, even it does not correspond to the conceptual hierarchy.

Subtyping for generalization may also be unavoidable if the programmer does not have complete control of the type hierarchy. For instance, the programmer may wish to create a generalization of some type in the system library, but to do so would mean creating a new type that is a supertype of the system type. If such changes to the library are not permitted, the reasonable alternative is to subtype the system type.

4.5.2 Subtyping for variance

Subtyping for variance is similar to subtyping for generalization: the conceptual hierarchy does not match the implementation hierarchy. In the case of variance, however, there is no direct hierarchical relationship between the supertype and the subtype. Instead, the types are likely to be cousins or siblings in the conceptual hierarchy.

For instance, in the Input_device example given earlier, the implementations of Mouse and Tablet may be nearly identical. One could then make Mouse a subtype of Tablet, or vice versa, to allow all the common code to be reused.

As with subtyping for generalization, there may be a better way to structure the types. In the example above, factoring out the common code into an abstract type, say Pointing_device, is conceptually cleaner. And as with subtyping for generalization, an inability to change an existing type hierarchy might make subtyping for variance necessary, though not the most desirable choice.

5 Types and inheritance: an extended example

To illustrate the ideas and principles we have presented in the previous sections, we will describe a more involved example, and show how our design choices are influenced by these guidelines.

We will consider the design of an inspector facility. An inspector displays debugging information about an object in a window designed for that purpose. The information displayed includes various aspects of the object's state, such as the contents of some or all of its fields, and perhaps some computed values as well. Different types of inspectors are needed for different kinds of objects. For instance, all the interesting information about a point can be displayed at once in a simple format, whereas a large two-dimensional array might best be displayed as a matrix which can be scrolled. Such type-specific data display can greatly speed up the debugging process.

We should first decide where to put the behavior of the inspector. Should we put it in the type of the object to be inspected, or in a new, separate type? An inspector window is a visible, concrete entity, and seems to be a good abstraction in its own right. On the other hand, the information the inspector will display is closely related to the object being inspected. Because we have conflicting indications about where inspector behavior should go, we will consider the guidelines discussed in section 3.3 in more detail.

We will first consider the general applicability of inspector behavior, and also how much the inspector must know about the representation of the object it will display. Since every type of object should be inspectable, and since debugging is a common task, inspector behavior is generally useful, and is applicable to every type. An inspector must also have intimate knowledge about the type of the object being inspected, because it needs to know what values to display, and what kind of display is most suitable for that type of object. Thus, both of these criteria imply that we should define inspector behavior within each type whose objects can be inspected.

But this conclusion conflicts with what the criteria of reusability and algorithm complexity would suggest. Many types of objects will be displayed in the same way. If we implement the display code in each type separately, we would end up with a lot of duplicated code. Instead, we should put the common code in a separate type; then the behavior could be shared by all the types that use it. In addition, handling the display of the debugging information on the screen is a complex task, and will probably require quite a lot of complicated code. Many aspects of this code have nothing to do with the type of the object being inspected. For reasons of modularity, this code should be put in a separate type.

The subtyping guidelines presented in section 4 also suggest that inspector behavior should be implemented in a separate type. Different types of objects will be displayed in different ways, but the variations from one type of display to another may be slight. Different kinds of inspectors will still have a great deal of behavior in common. We can therefore factor out this common behavior into a few supertypes, and create subtypes to handle the variations of different kinds of inspectors.

Since we have concluded that some aspects of object inspection are closely associated with the object itself, and others are not as germane, we should split the behavior for inspectors along this division. Thus, we will implement inspectors as separate types, but we will include an interface in each object type that allows an inspector to look at the internals of the object in a systematic way. We will also provide an operation in every type that creates and displays the proper kind of inspector for objects of that type.

Our new type, `Inspector`, will handle the actual display of debugging information. `Inspector` will have subtypes that define different kinds of inspectors, such as `Matrix_inspector` and `Bitmap_inspector`. An inspector will be given an object to be displayed. But the inspector will get information about the object through an `inspector_data` operation which is defined on the type `Object`. The default implementation of this operation in `Object` will return the names of all the fields in the object and their values. Each type, however, will be able to reimplement this operation to return whatever names and values are appropriate. Special-purpose inspectors may need more information, and will require that the objects they can inspect provide the information through additional operations. Thus we continue to follow our guidelines by associating the knowledge of what is to be displayed with individual types, and yet breaking off the details of the display into separate inspector types.

Next, we need to describe how a specific type of inspector gets associated with a particular kind of object. Recall that we want to specify this behavior on a type by type basis. We can do this by defining another operation on `Object`, called `inspect`. The default implementation for `inspect` will simply create and display a default kind of inspector. Again, types can override this implementation if they want a specific type of inspector to be created. For example, a hash table might display a `Dictionary_inspector` instead.

Finally, we should consider the structure of the type hierarchy used for the different kinds of inspectors. At first glance, we might consider defining the default `Inspector` as the top supertype. Types such as `Matrix_inspector` and `Dictionary_inspector` would be subtypes of this type. But in this type structure, subtyping is used largely for variation. A better structure might be one in which there is an abstract

supertype `Abstract_inspector`, with subtypes `Default_inspector`, `Matrix_inspector`, `Dictionary_inspector`, and so forth.

In this type hierarchy, the common behavior is clearly in `Abstract inspector`. In the earlier hierarchy, the common code is mixed in with the code that is specific for the default inspector. A programmer may end up coding a new inspector in a way that makes the new one unduly dependent on the implementation of `Inspector`. The whole hierarchy would therefore be less maintainable and extensible.

The example we have presented here illustrates applying some of the principles we described earlier in the paper. But these guidelines are not a panacea, and are not hard and fast rules. Creativity and clear-headed thinking are still the most important ingredients of design. Nevertheless, we hope that the ideas we have presented allow the design to proceed in a more methodical way than might otherwise be expected.

6 Conclusion

In this paper, we have looked at typing, subtyping, and inheritance as they are used in object-oriented languages. We have discussed what types should represent, how to distribute program behavior among types, how to design programs that use subtyping and inheritance, and the different ways in which inheritance can be used. We have tried to draw out issues and codify principles which the programmer should consider when making these decisions.

Any one particular object-oriented language will influence how types and inheritance are used in the program to be written. Nevertheless, the programmer should think about the design of the types in the program in more general terms before constraining the design to fit onto a particular language.

When properly applied, object-oriented languages can significantly enhance the maintainability, extensibility, and reusability of the code in a program. But the proper and careful use of types and inheritance is critical to the success of programs built in an object-oriented way. In this paper, we have presented a number of guidelines for using types and inheritance; we hope they will contribute to good object-oriented program design.

References

[Borgida] Alexander Borgida. Features of languages for the development of information systems at the conceptual level. *IEEE Software* **2**(1), January 1985.

[Curry *et al*] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: an approach to multiple-inheritance subclassing. *Proceedings of the SIGOA Conference on Office Information Systems*. Philadelphia, 1982.

[Goldberg & Robson] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[O'Brien] Patrick O'Brien. *Trellis Object-Based Environment: Language Tutorial*. Digital Equipment Corporation technical report DEC-TR-373, November 1985.

[Schaffert *et al*] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Manual*. Digital Equipment Corporation technical report DEC-TR-372, November 1985.

[Schaffert *et al*2] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. Portland, Oregon, 1986.

[Shaw] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software* **1**(4), October 1984.

[Wirth] Niklaus Wirth. Program development by step-wise refinement. *Communications of the ACM* **14**(4): 221-227, April 1971.