24

Using inheritance well

*L*earning all the technical details of inheritance and related mechanisms, as we did in part C, does not automatically mean that we have fully grasped the methodological consequences. Of all issues in object technology, none causes as much discussion as the question of when and how to use inheritance; sweeping opinions abound, for example on Internet discussion groups, but the literature is relatively poor in precise and useful advice.

In this chapter we will probe further into the meaning of inheritance, not for the sake of theory, but to make sure we use it best to benefit our software development projects. We will in particular try to understand how inheritance differs from the other inter-module relation in object-oriented system structures, its sister and rival, the client relation: when to use one, when to use the other, when both choices are acceptable. Once we have set the basic criteria for using inheritance — identifying along the way the typical cases in which it is wrong to use it — we will be able to devise a classification of the various legitimate uses, some widely accepted (subtype inheritance), others, such as implementation or facility inheritance, more controversial. Along the way we will try to learn a little from the experience in taxonomy, or *systematics*, gained from older scientific disciplines.

24.1 HOW NOT TO USE INHERITANCE

To arrive at a methodological principle, it is often useful — as illustrated by so many other discussions in this book — to study first how *not* to do things. Understanding a bad idea helps find good ones, which we might otherwise miss. In too constantly warm a climate, a pear tree will not flower; it needs the jolt of Winter frost to attain full bloom in the Spring.

Extracts from "Software Engineering" by Ian Sommerville, Fourth edition, Addison-Wesley, 1993.

Here the jolt is obligingly provided by a widely successful undergraduate textbook, used throughout the world to teach software engineering to probably more computing science students than any other. Already in its fourth edition, it introduced some elements of object orientation, including a discussion of multiple inheritance. Here is the beginning:

> *Multiple inheritance allows several objects to act as base objects and is supported in object-oriented languages such as* [the notation of the present book] [M 1988].

The bibliographic reference is to the first edition of the present book. Apart from the unfortunate use of "objects" for classes, this is an auspicious start. The extract continues:

> *The characteristics of several different object classes*

(classes, good!)

can be combined to make up a new object.

(no luck). Then comes the example of multiple inheritance:

For example, say we have an object class CAR which encapsulates information about cars and an object class PERSON which encapsulates information about people. We could use both of these to define

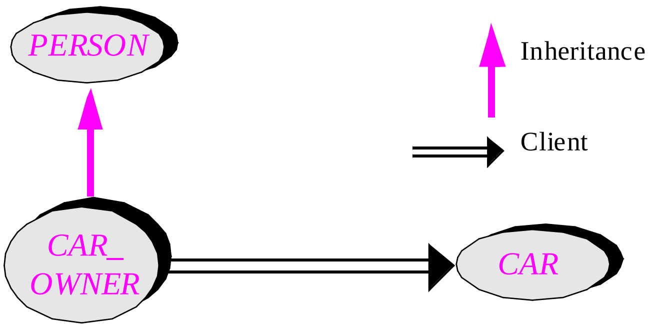
(will our worst fears come out true?)

a new object class CAR-OWNER which combines the attributes of CAR and PERSON.

(They have.) We are invited to consider that every *CAR-OWNER* object may be viewed as not only a person but also a car. To anyone who has studied inheritance even at an elementary level, this will be a surprise.

As you will undoubtedly have figured out, the relation to use in the second case was client, not inheritance: a car owner *is* a person, but *has* a car. In pictures:

A proper model



In formal words:

```
class CAR_OWNER inherit
        PERSON
feature
        my_car: CAR
        …
end -- class CAR_OWNER
```

In the cited text, both links use the inheritance relation. The most interesting twist actually comes a little later in the discussion, when the author advises his reader to treat inheritance with caution:

Adaptation through inheritance tends to lead to extra functionality being inherited, which can make components inefficient and bulky.

Bulky indeed; think of the poor car owner, loaded with his roof, engine and carburetor, not to mention four wheels plus a spare. This view might have been influenced by one of the picturesque phrases of Australian slang, about a car owner who does look as if he also *is* his car:

"He has a head like an Austin Mini with the doors open".

Cartoon by Geoff Hocking; from *The Dictionary of Aussie Slang*, The Five Mile Press, Melbourne, Australia, reprinted with permission.

Inheritance is a non-trivial concept, so we can forgive the author of this extract on the grounds that he was perhaps a little far from his home turf. But the example has an important practical benefit apart from helping us feel smarter: it reminds us of the basic rule on inheritance.

"Is-a" rule of inheritance

Do not make a class *B* inherit from a class *A* unless you can somehow make the argument that one can view every instance of *B* also as an instance of *A*.

In other words, we must be able to convince someone — if only ourselves to start with — that "every *B* is an *A*" (hence the name: "is-a").

In spite of what you may think at first, this is a loose rule, not a strict one. Here is why:

- Note the phrase ''can somehow make the argument". This is voluntarily vague: we do not require a *proof* that every *B* is an *A*. Many cases will leave room for discussion. Is it true that "Every savings account is a checking account"? There is no absolute answer; depending on the bank's policies and your analysis of the properties of the various kinds of account, you may decide to make class *SAVINGS_ACCOUNT* an heir to *BANK_ACCOUNT*, or put it elsewhere in the inheritance structure, getting some help from the other criteria discussed in this chapter. Reasonable people might still disagree on the result. But for this to be the case the "is-a" argument must be sustainable. Once again our counter-example helps: the argument that a *CAR_OWNER* "is-a" *CAR* is *not* sustainable.
- Our view of what "is-a" means will be particularly liberal. It will not, for example, disallow *implementation inheritance* — a form of inheritance that many people view with suspicion — as long as the "is-a" argument can reasonably be made.

These observations define both the usefulness and the limitations of the Is-a rule. It is useful as a *negative* rule in the Popperian style, enabling you to detect and reject inappropriate uses of inheritance. But as a positive rule it is not sufficient; not all suggested uses that pass the rule's test will be appropriate.

Gratifying as the *CAR_OWNER* counter-example may be, then, any feeling of elation that we may have gained from it will be short-lived. It was both the beginning and the end of the unmitigated good news — the news that some proposed uses of inheritance are obviously wrong and easy to spot. The rest of this chapter has to contend with the bad or at least mixed news: that in just about all other cases the decision is a true design issue, that is to say hard, although we will fortunately be able to find some general guidelines.

24.2 WOULD YOU RATHER BUY OR INHERIT?

To choose between the two possible inter-module relations, client and inheritance, the basic rule is deceptively simple: client is *has*, inheritance is *is*. Why then is the choice not easy?

To have and to be

The reason is that whereas to have is not always to be, in many cases *to be is also to have*.

No, this is neither some cheap attempt at existentialist philosophy nor a pitch to make you buy a house if you are currently renting; rather, simple observations on the difficulty of system modeling. We have already encountered an illustration of the first property — to have is not always to be — in the preceding example: a car owner has a car, but by no twist of reasoning or exposition can we assert that he is a car.

What about the reverse situation? Take a simple statement about two object types from ordinary life, such as

Every software engineer is an engineer. [A]

whose truth we accept for its value as an example of the "is-a" relation (whatever our opinion may be as to the statement's accuracy). It seems hard indeed to think of a case which so clearly expresses "to be" rather than "to have". But now consider the following rephrasing of the property:

In every software engineer there is an engineer [B]

which can in turn be restated as

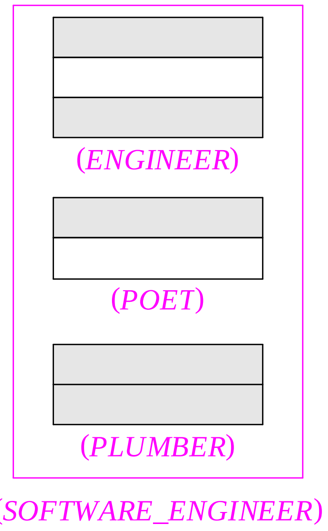
Every software engineer has an "engineer" component. [C]

Twisted, yes, and perhaps a trifle bizarre in its expression; but not fundamentally different from our premise [A]! So here it is: by changing our perspective slightly we can rephrase the "is" property as a "has".

"COMPOSITE OBJECTS AND EXPANDED TYPES", 8.7, page 254

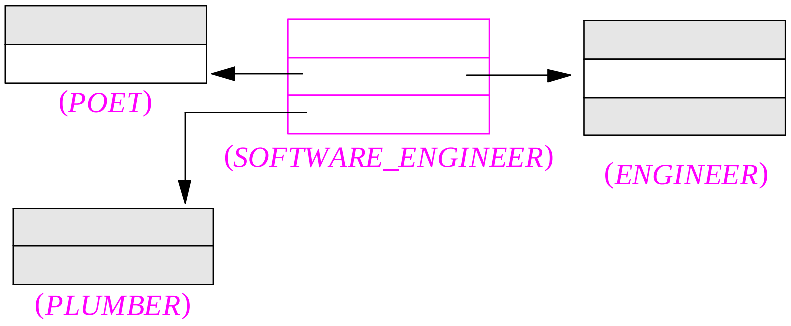
If we look at the picture through the eyes of a programmer, we may summon an object diagram, in the style of those which served to discuss the dynamic model in an earlier chapter, showing a typical instance of a class and its components:

A "software engineer" object as aggregate



This shows an instance of *SOFTWARE_ENGINEER* with various subobjects, representing the various posited aspects of a software engineer's personality and tasks. Rather than subobjects (the expanded view) we might prefer to think in terms of references:

Another possible view



Take both of these representations as ways to visualize the situation as seen from an implementation-oriented mindset, nothing more. Both suggest, however, that a client, or "has", interpretation — every software engineer has an engineer as one of his parts — is faithful to the original statement. The same observation can be made for any similar "is-a" relationship.

So this is why the problem of choosing between client and inheritance is not trivial: when the "is" view is legitimate, one can always take the "has" view instead.

The reverse is not true: when "has" is legitimate, "is" is not always applicable, as the *CAR_OWNER* example shows so clearly. This observation takes care of the easy mistakes, obvious to anyone having understood the basic concepts, and perhaps even explainable to authors of undergraduate texts. But whenever "is" does apply it is not the only contender. So two reasonable and competent people may disagree, one wanting to use inheritance, the other preferring client.

Two criteria fortunately exist to help in such discussions. Not surprisingly (since they address a broad design issue) they may sometimes fail to give a clear, single solution. But in many practical cases they do tell you, beyond any hesitation, which of the two relations is the right one.

Conveniently, one of these two criteria favors inheritance, and the other favors client.

The rule of change

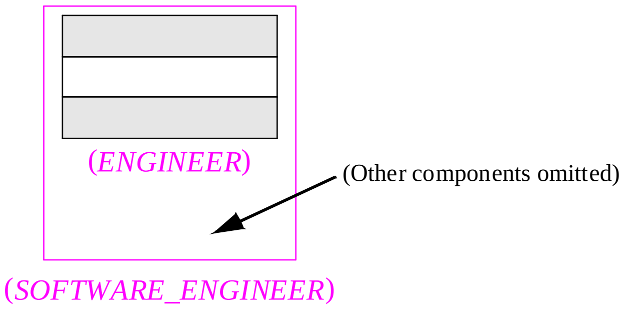
The first observation is that the client relation usually permits change, while the inheritance relation does not. Here we must be careful with our use of the verbs "to be" and "to have" from ordinary language; so far they have helped us characterize the general nature of our two software relations, but software rules are, as always, more precise than their general non-software counterparts.

One of the defining properties of inheritance is that it is a relation between **classes**, not objects. We have interpreted the property "Class *B* inherits from class *A*" as meaning "every *B* object is an *A* object", but must remember that it is not in the power of any such object to change that property: only a change of the class can achieve such a result. The property characterizes the software, not any particular execution.

With the client relation, the constraints are looser. If an object of type *B* has a component of type *A* (either a subobject or an object reference), it is quite possible to change that component; the only restrictions are those of the type system, ensuring provably reliable execution (and governed, through an interesting twist, by the inheritance structure).

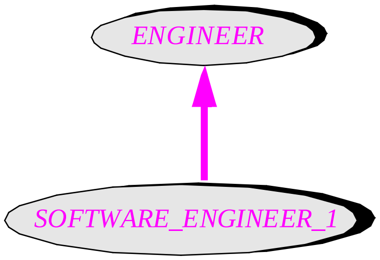
So even though a given inter-object relationship can result from either inheritance or client relationships between the corresponding classes, the effect will be different as to what can be changed and what cannot. For example our fictitious object structure

Object and subobject



could result from an inheritance relationship between the corresponding classes:

```
class SOFTWARE_ENGINEER_1 inherit
    ENGINEER
feature
    …
end -- class SOFTWARE_ENGINEER_1
```

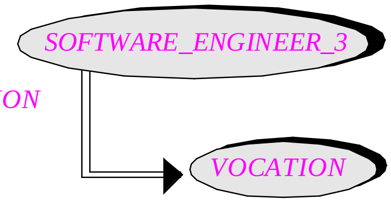


but it could just as well have been obtained through the client relation:

```
class SOFTWARE_ENGINEER_2 feature
    the_engineer_in_me: ENGINEER
    …
end -- class SOFTWARE_ENGINEER_2
```

which could in fact be

```
class SOFTWARE_ENGINEER_3 feature
    the_truly_important_part_of_me: VOCATION
    …
end -- class SOFTWARE_ENGINEER_3
```



provided we satisfy the type rules by making class *ENGINEER* a descendant of class *VOCATION*.

Strictly speaking the last two variants represent a slightly different situation from the first if we assume that none of the given classes is expanded: instead of subobjects, the "software engineer" objects will in the last two cases contain *references* to "engineer" objects, as in the second figure of page 813. The introduction of references, however, does not fundamentally affect this discussion.

With the first class definition, because the inheritance relationship holds between the generating classes, it is not possible to modify the object relationship dynamically: once an engineer, always an engineer.

But with the other two definitions such a modification is possible: a procedure of the "software engineer" class can assign a new value to the corresponding object field (the field for *the_engineer_in_me* or *the_truly_important_part_of_me*). In the case of class *SOFTWARE_ENGINEER_2* the new value must be of type *ENGINEER* or compatible; but with class *SOFTWARE_ENGINEER_3* it may be of any type compatible with *VOCATION*. So our software can model the idea of a software engineer who, after many years of pretending to be an engineer, finally sheds that part of his personality in favor of something that he deems more representative of his work, such as poet or plumber.

This yields our first criterion:

Rule of change

Do not use inheritance to describe a perceived “is-a” relation if the corresponding object components may have to be changed at run time.

Only use inheritance if the corresponding inter-object relation is permanent. In other cases, use the client relation.

The really interesting case is the one illustrated by *SOFTWARE_ENGINEER_3*. With *SOFTWARE_ENGINEER_2* you can only replace the engineer component with another of exactly same type. But in the *SOFTWARE_ENGINEER_3* scheme, *VOCATION* should be a high-level class, most likely deferred; so the attribute can (through polymorphism) represent objects of many possible types, all conforming to *VOCATION*.

This also means that even though this solution uses client as the primary relation, in practice its final form will often use inheritance as a complement. This will be particularly clear when we come to the notion of handle.

The polymorphism rule

Now for a criterion that will require inheritance and exclude client. That criterion is simple: polymorphic uses. In our study of inheritance we have seen that with a declaration of the form

x: *C*

x denotes at run time (assuming class *C* is not expanded) a potentially polymorphic reference; that is to say, *x* may become attached to direct instances not just of *C* but of any proper descendants of *C*. This property is of course a key contribution to the power and flexibility of the object-oriented method, especially through its corollary, the possibility of defining polymorphic data structures, such as a *LIST* [*C*] which may contains instances of any of *C*’s descendants.

In our example, this means that with the *SOFTWARE_ENGINEER_1* solution — the form of the class which inherits from *ENGINEER* — a client can declare an entity

eng: *ENGINEER*

which may become attached at run time to an object of type *SOFTWARE_ENGINEER_1*. Or we can have a list of engineers, or a database of engineers, which includes a few mechanical engineers, a few chemical engineers, and a few software engineers as well.

A reminder on methodology: the use of non-software words is a good help for understanding the concepts, but we should not let ourselves get carried away by such anthropomorphic examples; the objects of interest are software objects. So although we may loosely understand the words “a software engineer” for what they say, they actually denote an instance of *SOFTWARE_ENGINEER_1*, that is to say, a software object somehow modeling a real person.

Such polymorphic effects require inheritance: with *SOFTWARE_ENGINEER_2* or *SOFTWARE_ENGINEER_3* there is no way an entity or data structure of type *ENGINEER* can directly denote "software engineer" objects.

Generalizing these observations — which are not, of course, specific to the example — yields the complement of the rule of change:

Polymorphism rule

Inheritance is appropriate to describe a perceived "is-a" relation if entities or data structure components of the more general type may need to become attached to objects of the more specialized type.

Summary

Although it brings no new concept, the following rule will be convenient as a summary of this discussion of criteria for and against inheritance.

Choosing between client and inheritance

In deciding how to express the dependency of a class *B* on a class *A*, apply the following criteria:

CI1 • If every instance of *B* initially has a component of type *A*, but that component may need to be replaced at run time by an object of a different type, make *B* a client of *A*.

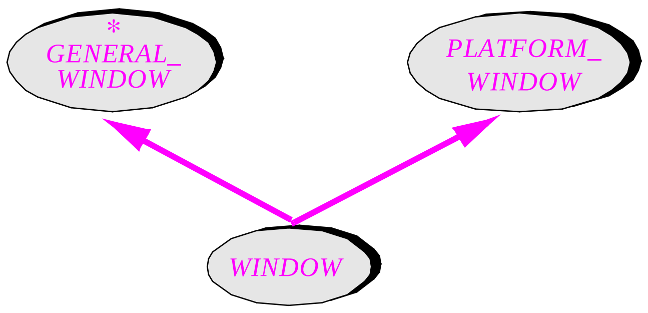
CI2 • If there is a need for entities of type *A* to denote objects of type *B*, or for polymorphic structures containing objects of type *A* of which some may be of type *B*, make *B* an heir of *A*.

24.3 AN APPLICATION: THE HANDLE TECHNIQUE

Here is an example using the preceding rule. It yields a design pattern of wide applicability: *handles*.

The first design of the *Vision* library for platform-independent graphics encountered a general problem: how to account for platform dependencies. The first solution used multiple inheritance in the following way: a typical class, such as the one describing windows, would have a parent describing the platform-independent properties of the corresponding abstraction, and another providing the platform-specific elements.

```
class WINDOW inherit
    GENERAL_WINDOW
    PLATFORM_WINDOW
feature
    …
end -- class WINDOW
```



Platform adaptation through inheritance

Class *GENERAL_WINDOW* and similar ones such as *GENERAL_BUTTON* are deferred: they express all that can be said about the corresponding graphical objects and the applicable operations without reference to a particular graphical platform. Classes such as *PLATFORM_WINDOW* provide the link to a graphical platform such as Windows, OS/2-Presentation-Manager or Unix-Motif; they give access to the platform-specific mechanisms (encapsulated through a library such as WEL or MEL).

On the platform-specific libraries WEL and MEL see "Object-oriented re-architecturing", page 441

A class such as *WINDOW* will then combine its two parents through features which effect (implement) the deferred features of *GENERAL_WINDOW* by using the implementation mechanisms provided by *PLATFORM_WINDOW*.

PLATFORM_WINDOW (like all other similar classes) needs several variants, one for each platform. These identically named classes will be stored in different directories; the Ace for a compilation (the control file) will select the appropriate one.

On the notion of Ace see "Assembling a system", page 198

This solution works, but it has the drawback of tying the notion of *WINDOW* closely to the chosen platform. To transpose an earlier comment about inheritance: once a Motif window, always a Motif window. This may not be too bad, as it is hard to imagine a Unix window which, suddenly seized by middle-age anxiety, decides to become an OS/2 window. The picture becomes less absurd if we expand our definition of "platform" to include formats such as Postscript or HTML; then a graphical object could change representation for purposes of printing or inclusion in a Web document.

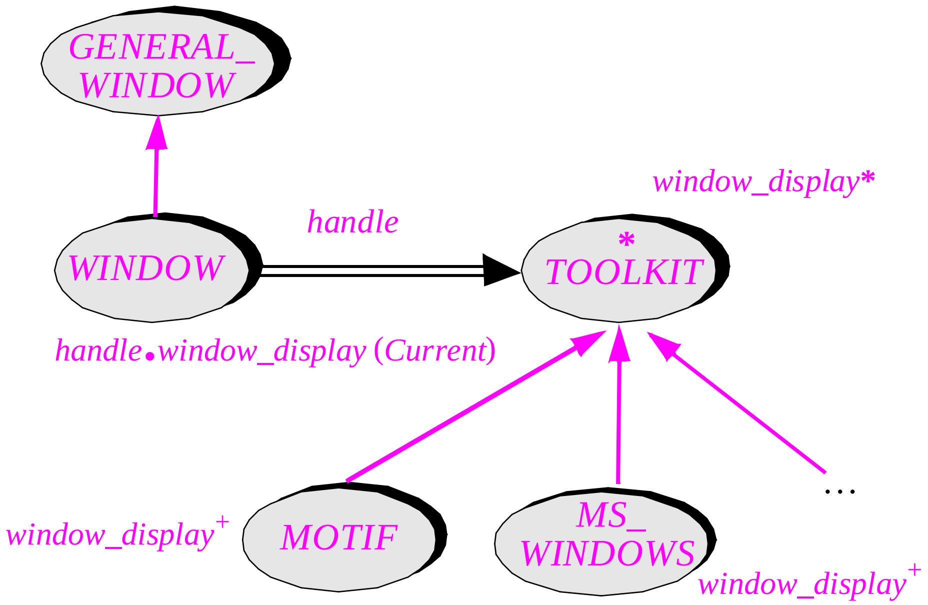
The observation that we might need a looser connection between GUI objects such as a window and the underlying toolkit suggests trying the client relation. An inheritance link will remain, between *WINDOW* and *GENERAL_WINDOW*; but the platform dependency will be represented by a client link to a class *TOOLKIT* representing the underlying "toolkit" (graphical platform). The figure at the top of the facing page illustrates the resulting structure, involving both client and inheritance.

An interesting aspect of this solution is that it recognizes the notion of toolkit as a full-fledged abstraction, represented by a deferred class *TOOLKIT*. Each specific toolkit is then represented by an effective descendant of *TOOLKIT* such as *MOTIF* or *MS_WINDOWS*.

Here is how it works. Each class describing graphical objects, such as *WINDOW*, has an attribute providing access to the underlying platform:

```
handle: TOOLKIT
```

Platform adaptation through a handle



This will yield a field in each instance of the class. It is possible to change the handle:

```
set_handle (new: TOOLKIT) is
            -- Make new the new handle for this object.
    do
        handle := new
    end
```

A typical operation inherited from *GENERAL_WINDOW* in deferred form will be effected through a call to the platform's mechanism:

```
display is
            -- Display window on screen.
    do
        handle.window_display (Current)
    end
```

Through the handle, the graphical object asks the platform to perform the required operation. A feature such as *window_display* is deferred in class *TOOLKIT* and effected variously for its various descendants such as *MOTIF*.

Note that it would be inappropriate to draw from this example the conclusion "Aha! Another case in which inheritance was overused, and the final version stays away from it." The initial version was not wrong; in fact it works quite well, but is less flexible than the second one. And that second version fundamentally relies on inheritance and the consequent techniques of polymorphism and dynamic binding, which it combines with the client relation. Without the *TOOLKIT*-rooted inheritance hierarchy, the polymorphic entity *handle*, and dynamic binding on features such as *window_display*, it would not work. Far from being a rejection of inheritance, then, this technique illustrates a more sophisticated form of inheritance.

The handle technique is widely applicable to the development of libraries supporting multi-platform compatibility. Besides the *Vision* graphical library, we have applied it to the *Store* database library, where the notion of platform covers various SQL-based relational database interfaces such as Oracle, Ingres, Sybase and ODBC.

24.4 TAXOMANIA

For every one of the inheritance categories introduced later in this chapter, the heir redeclares (redefines or effects) some inherited features, or introduces features of its own, or adds to the invariant. (It may of course do several of these things.) A consequence is:

Taxomania rule

Every heir must introduce a feature, redeclare an inherited feature, or add an invariant clause.

This is actually a consequence of the Inheritance rule seen later in this chapter, page 822.

What this rule addresses is a foible sometimes found in newcomers who have been won over to the O-O method, and enthusiastically start seeing taxonomical divisions everywhere (hence the name of the rule, a shortcut for "taxonomy mania"). The result is over-complicated inheritance hierarchies. Taxonomy and inheritance are meant to *help* us master complexity, not to introduce complexity. Adding useless classification levels is self-defeating.

As is so often the case, you can gain the proper perspective — and bring the neophytes back to reason — by keeping in mind the ADT view at all times. A class is the implementation, partial or total, of an abstract data type. Different classes, in particular a parent and an heir, should describe different ADTs. Then, because an ADT is entirely characterized by the applicable features and their properties (captured in the class by assertions), a new class should change an inherited feature, introduce a new feature or change some assertion. Since you can only change a precondition or postcondition by redefining the enclosing feature, the last case means the addition of an invariant clause (as in *restriction inheritance*, one of the categories in our taxonomy).

You may occasionally justify a case of taxomania — a class that does not bring anything new of its own, apart from its existence — on the grounds that the heir class describes an important variant of the notion described by the parent, and that you are introducing it now to pave the way for future introduction or redeclaration of features, even if none has occurred so far. This may be valid when the inheritance structure corresponds to a generally accepted classification in the problem domain. But you should always be wary of such cases, and resist the introduction of new featureless classes unless you can find compelling arguments.

Here is an example. Assume a certain system or library includes a class *PERSON* and that you are considering adding heirs *MALE* and *FEMALE*. Is this justified? You will have to take a closer look. A personnel management system that includes gender-specific features, pertaining for example to maternity leave, may benefit from having heir classes *MALE* and *FEMALE*. But in many other cases the variants, if present, would have no specific features; for example statistical software that just records the gender of individuals may be better off with a single class *PERSON* and a boolean attribute

```
female: BOOLEAN
```

or perhaps

```
Female: INTEGER is unique
Male: INTEGER is unique
```

rather than new heirs. Yet if there is any chance that specific features will be added later on, the corresponding classification is so clearly known in the problem domain that you may prefer to introduce these heirs anyway.

One guideline to keep in mind is the Single Choice principle. We have learned to distrust the use of explicit variant lists, as implemented by **unique** constants, for fear of finding our software polluted with conditional instructions of the form

```
if female then
    …
else
    …
```

or **inspect** instructions. This is, however, not too much of a concern here:

- One of the principal criticisms against this style was that any addition of a variant would cause a chain reaction of changes throughout the software, but in certain cases — such as the above example — we can be confident there will be no new variants.
- Even with a fixed set of variants, the explicit **if** … style is less effective than relying on dynamic binding through calls such as *this_person*•*some_operation* where *MALE* and *FEMALE* have different redeclarations of *some_operation*. But then if we do need to discriminate on a person's gender we violate the premise of this discussion — that there are no features specific to the variants. If such features do exist, inheritance is justified.

The last comment alerts us to the real difficulty. Simple cases of taxomania — in which the patient needlessly adds intermediate nodes all over the inheritance structure — are relatively easy to diagnose (by noticing classes that have no specific features) and cure. But what if the variants *do* have specific features, although the resulting classification conflicts with other criteria? A personnel management system for which we can justify a class *FEMALE_EMPLOYEE* because of a few specific features might have other distinctions as well, such as permanent versus temporary employees, or supervisory versus non-supervisory ones. Then we do not have taxomania any more, but face a general and delicate problem, *multi-criteria classification*, whose possible solutions are discussed later in this chapter.

24.5 USING INHERITANCE: A TAXONOMY OF TAXONOMY

The power of inheritance comes from its versatility. True, this also makes it scary at times, causing many authors to impose restrictions on the mechanism. While understanding these fears and even sometimes sharing them — do the boldest not harbor the occasional doubt and anxiety? — we should overcome them and learn to enjoy inheritance under all of its legitimate variants, which will now be explored.

After recalling some commonly encountered wrong uses of inheritance we will individually review the valid uses:

- Subtype inheritance.
- View inheritance.
- Restriction inheritance.
- Extension inheritance.
- Functional variation inheritance
- Type variation inheritance.
- Reification inheritance.
- Structure inheritance.
- Implementation inheritance.
- Facility inheritance (with two special variants: constant inheritance and machine inheritance).

Some of these categories (subtype, view, implementation, facility) raise specific issues and will be discussed in more detail in separate sections.

Scope of the rules

The relatively broad view of inheritance taken in this book in no way means that "anything goes". We accept and in fact encourage certain forms of inheritance on which some authors frown; but of course there are many ways to misuse inheritance, and not just *CAR_OWNER*. So the inevitable complement of our broad-mindedness is a particularly strict constraint:

Inheritance rule

Every use of inheritance should belong to one of the accepted categories.

This rule is stern indeed: it states that the types of use of inheritance are known and that if you encounter a case that is not covered by one of these types you should just *not* use inheritance.

What are "the accepted categories"? The implicit meaning is "the accepted categories, as discussed in the rest of this section". I indeed hope that all meaningful uses are covered. But the phrasing is a little more careful because the taxonomy may need further thinking. I found precious little in the literature about this topic; the most useful reference is an unpublished Ph. D. thesis [Girod 1991]. So it is quite possible that this attempt at classification has missed some categories. But the rule indicates that if you see a possible use of inheritance that does not fall into one of the following categories, you should give it serious thought. Most likely you should not use inheritance in that case; if after further reflection you are still convinced that inheritance is appropriate, and you are still unable to attach your example to one of the categories of this chapter, then you may have a new contribution to the literature.

Page 820.

We already saw a consequence of the Inheritance rule: the Taxomania rule, which states that every heir class should redeclare or introduce a feature, or change some assertion. It follows directly from the observation that every legitimate form of inheritance detailed below requires the heir to perform at least one of these operations.

The Inheritance rule does not prohibit inheritance links that belong to *more than one* of the inheritance categories. Such practice is, however, not recommended:

Inheritance Simplicity rule

A use of inheritance should preferably belong to just one of the accepted categories.

See "Advisories", page 667.

This is not an absolute rule but what an earlier discussion called an "advisory positive". The rationale for the rule is once again the desire for simplicity and clarity: if whenever you introduce an inheritance link between two classes you apply explicit methodological principles, and in particular decide which one of the approved variants you will be using, you are less likely to make a design mistake or to produce a messy, hard-to-use and hard-to-maintain system structure.

A compelling argument does not seem to exist, however, for making the rule absolute, and once in a while it may be convenient to use a single inheritance link for two of the goals captured by the classification. Such cases remain a minority.

Unfortunately I do not know of a simple criterion that would unambiguously tell us when it is all right to collapse several inheritance categories into one link. Hence the advisory nature of the Inheritance Simplicity rule. The reader's judgment, based on a clear understanding of the methodology of inheritance, should decide any questionable case.

Wrong uses

The preceding two rules confirm the obvious: that it is possible to misuse inheritance. Here is a list of typical mistakes, most of which have already been mentioned. Human ability for mischief being what it is, we can in no way hope for completeness, but a few common mistakes are easy to identify.

The first is **"has" relation with no "is" relation**. *CAR_OWNER* served as an example — extreme but not unique. Over the years I have heard or seen a few similar ones, often as purported examples of multiple inheritance, such as *APPLE_PIE* inheriting from

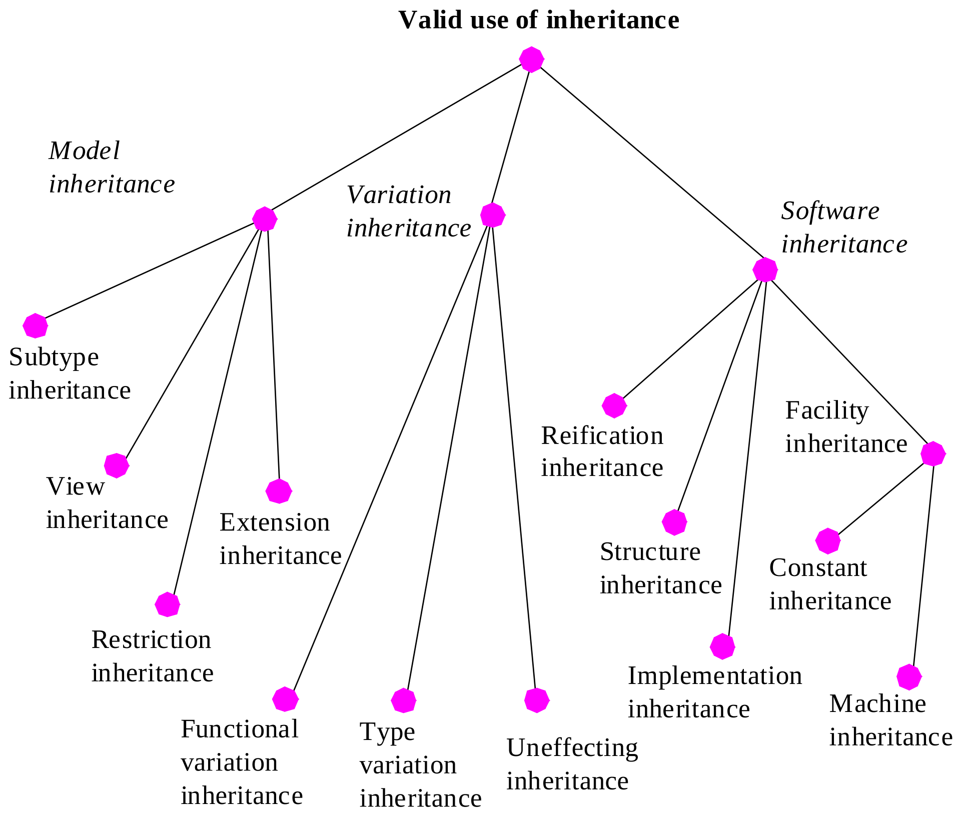
APPLE and from *PIE*, or (this one reported by Adele Goldberg) *ROSE_TREE* inheriting from *ROSE* and from *TREE*.

Another is a typical case of **taxomania** in which a simple boolean property, such as a person's gender (or a property with a few fixed values, such as the color of a traffic light) is used as an inheritance criterion even though no significant feature variants depend on it.

A third typical mistake is **convenience inheritance**, in which the developer sees some useful features in a class and inherits from that class simply to reuse these features. What is wrong here is neither the act of "using inheritance for implementation", nor "inheriting a class for its features", both of which are acceptable forms of inheritance studied later in this chapter, but the use of a class as a parent *without the proper is-a relationship between the corresponding abstractions* — or in some cases without adequate abstractions at all.

General taxonomy

On now to the valid uses of inheritance. The list will include twelve different categories, conveniently grouped into three broad families:



Classification of the valid categories of inheritance

The classification is based on the observation that any software system reflects a certain external model, itself connected with some outside reality in the software's application domain. Then we may distinguish:

- Model inheritance, reflecting "is-a" relations between abstractions in the model.
- Software inheritance, expressing relations within the software, with no obvious counterpart in the model.
- Variation inheritance — a special case that may pertain either to the software or to the model — serving to describe a class through its differences with another class.

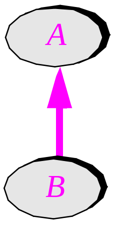
These three general categories facilitate understanding, but the most important properties are captured by the final categories (the tree leaves on the preceding figure).

Exercise E24.2, page 869.

Since the classification is itself a taxonomy, you may want to ask yourself, out of curiosity, how the identified categories apply to it. This is the topic of an exercise.

The definitions which follow all use the names *A* for the parent class and *B* for the heir.

Naming convention for definitions of inheritance categories



Each definition will state which of *A* and *B* is permitted to be deferred, and which effective. A table at the end of the discussion recalls the applicable categories for each deferred-effective combination.

Subtype inheritance

We start with the most obvious form of model inheritance. You are modeling some external system where a category of (external) objects can be partitioned into disjoint subcategories — as with closed figures, partitioned into polygons, ellipses etc. — and you use inheritance to organize the corresponding classes in the software. A bit more formally:

Definition: subtype inheritance

Subtype inheritance applies if *A* and *B* represent certain sets *A'* and *B'* of external objects such that *B'* is a subset of *A'* and the set modeled by any other subtype heir of *A* is disjoint from *B'*. *A* must be deferred.

A' could be the set of closed figures, *B'* the set of polygons, *A* and *B* the corresponding classes. In most practical cases the "external system" will be non-software, for example some aspect of a company's business (where the external objects might be checking and savings accounts) or some part of the physical world (where they might be planets and stars).

Subtype inheritance is the form of inheritance that is closest to the hierarchical taxonomies of botany, zoology and other natural sciences (*VERTEBRATE* ← *MAMMAL* and the like). A typical software example (other than closed figures and polygons) is *DEVICE* ← *FILE*. We insist that the parent, *A*, be deferred, so that it describes a non-completely specified set of objects. *B*, the heir, may be effective, or it may still be deferred. The next two categories cover the case in which *A* may be effective.

A later section will explore in more detail this inheritance category, not always as straightforward as it would seem at first.

“SUBTYPE INHERITANCE AND DESCENDANT HIDING”, 24.7, page 835.

Restriction inheritance

Definition: restriction inheritance

Restriction inheritance applies if the instances of *B* are those instances of *A* that satisfy a certain constraint, expressed if possible as part of the invariant of *B* and not included in the invariant of *A*. Any feature introduced by *B* should be a logical consequence of the added constraint. *A* and *B* should be both deferred or both effective.

Typical examples are *RECTANGLE* ← *SQUARE*, where the extra constraint is *side1 = side2* (included in the invariant of *SQUARE*), and *ELLIPSE* ← *CIRCLE*, where the extra constraint is that the two focuses (or *foci*) of an ellipse are the same point for a circle ; in the general case an ellipse is the set of points such that the sum of their distances to the two focuses is equal to a certain constant. Many mathematical examples indeed fall into this category.

The last part of the definition is meant to avoid mixing this form of inheritance with others, such as extension inheritance, which may add completely new features in the heir. Here to keep things simple it is preferable to limit new features, if any, to those that directly follow from the added constraint. For example class *CIRCLE* will have a new feature *radius* which satisfies this property: in a circle, all points have the same distance from the merged center, and this distance deserves the status of a feature of the class, whereas the corresponding notion in class *ELLIPSE* (the average of the distances to the two focuses) was probably not considered significant enough to yield a feature.

Because the only conceptual change from *A* to *B* is to add some constraints, the classes should be both deferred or both effective.

Restriction inheritance is conceptually close to subtype inheritance; the later discussion of subtyping will for the most part apply to both categories.

Extension inheritance

Definition: extension inheritance

Extension inheritance applies when *B* introduces features not present in *A* and not applicable to direct instances of *A*. Class *A* must be effective.

The presence of both the restriction and extension variants is one of the paradoxes of inheritance. As noted in the discussion of inheritance, extension applies to features whereas restriction (and more generally specialization) applies to instances, but this does not completely eliminate the paradox.

The problem is that the added features will usually include attributes. So if we take the naïve interpretation of a type (as given by a class) as the set of its instances, then it seems the subset relation is the wrong way around! Assume for example

```
class A feature a1: INTEGER end

class B inherit
    A
feature
    b1: REAL
end
```

Non-mathematical readers may skip this one paragraph.

Then if we view each instance of *A* as representing a singleton, that is to say a set containing one integer (which we can write as <*n*> where *n* is the chosen integer) and each instance of *B* as a pair containing an integer and a real (such as the pair <*1*, *–2.5*>), the set of pairs *MB* is not a subset of the set of singletons *MA*. In fact, if we absolutely want a subset relation, it will be in the reverse direction: there is a one-to-one mapping between *MA* and the set of all pairs having a given second element, for example *0.0*.

This discovery that the subset relation seems to be the wrong way may make extension inheritance look suspicious. For example an early version of a respected O-O library (not from ISE) had *RECTANGLE* inheriting from *SQUARE*, not the other way around as we have learned. The reasoning was simple: *SQUARE* has a *side* attribute; *RECTANGLE* inherits from *SQUARE* and adds a new feature, *other_side*, so here is an inheritance link for you! Several people criticized the design and it was soon reversed.

But we cannot dismiss the general category of extension inheritance. In fact its equivalent in mathematics, where you specialize a certain notion by adding completely new operations, is frequently used and considered quite necessary. A typical example is the notion of *ring*, specializing the notion of *group*. A group has a certain operation, say $+$, with certain properties. A ring is a group, so it also has $+$ with these properties, but it adds a new operation, say $*$, with extra properties of its own. This is not fundamentally different from introducing a new attribute in an heir software class.

The corresponding scheme is frequent in O-O software too. In most applications, of course, *SQUARE* should inherit from *RECTANGLE*, not the reverse; but it is not difficult to think of legitimate examples. A class *MOVING_POINT* (for kinematics applications) might inherit from a purely graphical class *POINT* and add a feature *speed* describing the speed's magnitude and direction; or, in a text processing application, a class *CHAPTER* might inherit from *DOCUMENT*, adding the specific features of a document which is a chapter in a book, such as its current position in the book and a procedure that will reposition it.

A proper mathematical model

(Non-mathematically-inclined readers should skip this section.)

For peace of mind we must resolve the apparent paradox noted earlier (the discovery that *MB* is not a subset of *MA*) since we do want some subset relation to hold between instances of an heir and instances of the parent. That relation does exist in the case of extension inheritance; what the paradox shows is that it is inappropriate to use cartesian product of the attribute types to model a class. Given a class

```
class C feature
    c1: T1
    c2: T2
    c3: T3
end
```

we should *not* take, as a mathematical model *C'* for the set of instances of *C*, the cartesian product $T'1 \times T'2 \times T'3$, where the prime signs *'* indicate that we recursively use the model sets; this would lead to the paradox (among other disadvantages).

The functions of interest are not only partial but finite.

Instead, we should consider any instance as being a partial function from the set of possible attribute names *ATTRIBUTE* to the set of all possible values *VALUE*, with the following properties:

A1 • The function is defined for *c1*, *c2* and *c3*.

A2 • The set *VALUE* (the target set of the function) is a superset of $T'1 \cup T'2 \cup T'3$.

A3 • The function's value for *c1* is in *T'1*, and so on.

Then if we remember that a function is a special case of a relation, and that a relation is a set of pairs (for example an instance of class *A* may be modeled by the function {<*a1*, *25*>}, and the instance of *B* cited on the preceding page by {<*a1*, *1*>, <*b1*, *–2.5*>}), then we do have the expected property that *B'* is a subset of *A*.

Note that it is essential to state the property A1 as "The function is defined for…", not "The function's domain is…" which would limit the domain to the set {*c1*, *c2 c3*}, preventing descendants from adding their own attributes. As a result of this approach, every software object is modeled by an infinity of (finite) mathematical objects.

This discussion has only given a sketch of the mathematical model. For more details on using partial functions to model tuples, and the general mathematical background, see [M 1990].

Variation inheritance

(Non-mathematical readers, welcome back!) We now move to the second of our three broad groups of inheritance categories: variation inheritance.

Definition: functional and type variation inheritance

Variation inheritance applies if *B* redefines some features of *A*; *A* and *B* are either both deferred or both effective, and *B* must not introduce any features except for the direct needs of the redefined features. There are two cases:

- Functional variation inheritance: some of the redefinitions affect feature bodies, rather than just their signatures.
- Type variation inheritance: all redefinitions are signature redefinitions.

Variation inheritance is applicable when an existing class *A*, describing a certain abstraction, is already useful by itself, but you discover the need to represent a similar although not identical abstraction, which essentially has the same features with some different signatures or implementations.

The definition requires that both classes be effective (the more common case) or both deferred: variation inheritance does not cover the case of an effecting, where we transform a notion from abstract to concrete. A closely related category is uneffecting, studied next, in which some effective features are made deferred.

The definition stipulates that the heir should introduce no new features, except as directly needed by the redefined features. This clause distinguishes variation inheritance from extension inheritance.

In **type** variation inheritance you only change the signatures (argument and result types and number) of some features. This form of inheritance is suspect; it is often a sign of taxomania. In legitimate cases, however, it may be a preparation for extension inheritance or implementation variation inheritance. An example of type variation inheritance might be the heirs *MALE_EMPLOYEE* and *FEMALE_EMPLOYEE*.

Type variation inheritance is not necessary when the original signature used anchored (**like** …) declarations. For example in the *SEGMENT* class of an interactive drawing package you may have introduced a function

```
perpendicular: SEGMENT is
            -- Segment of same length and same middle point, rotated 90 degrees
      …
```

and then want to define an heir *DOTTED_SEGMENT* to provide a graphical representation with a dotted line rather than a continuous one. In that class, *perpendicular* should return a result of type *DOTTED_SEGMENT*, so you will need to redefine the type. None of this would be needed if the original returned a result of type **like** *Current*, and if you have access to the source of the original and the authority to modify it you may prefer to update that type declaration, normally without any adverse effect on existing clients. But if for some reason you cannot modify the original, or if an anchored declaration is not appropriate in that original (perhaps because of the needs of other descendants), then the ability to redefine the type can save the day.

In **functional** variation inheritance we change some of the features' bodies; if, as is usually the case, the features were already effective, this means changing their implementation. The features' specification, as given by assertions, may also change. It is also possible, although less common, to have functional variation inheritance between two deferred classes; in that case the assertions will change. This may imply changes in some functions, deferred or effective, used by the assertions, or even the addition of new features as long as this is for the "direct needs of the redefined features" as the definition states.

Functional variation inheritance is the direct application of the Open-Closed principle: we want to adapt an existing class without affecting the original (of which we may not even have the source code) and its clients. It is subject to abuses since it may be a form of hacking: twisting an existing class so as to fit a slightly different purpose. At least this will be *organized* hacking, which avoids the dangers of directly modifying existing software, as analyzed in the discussion of the Open-Closed principle. But if you do have access to the source code of the original class, you should examine whether it is not preferable to reorganize the inheritance hierarchy by introducing a more abstract class of which both *A* (the existing variant) and *B* (the new one) will both be heirs, or proper descendants with peer status.

See "The Open-Closed principle", page 57.

Uneffecting

Definition: uneffecting inheritance

Uneffecting inheritance applies if *B* redefines some of the effective features of *A* into deferred features.

Uneffecting is not common, and should not be. Its basic idea goes against the normal direction of inheritance, since we usually expect *B* to be more concrete and *A* more abstract (as with the next category, reification, for which *A* is deferred and *B* effective or at least less deferred). For that reason beginners should stay away from uneffecting. But it may be justified in the following two cases:

- In multiple inheritance, you may want to merge features inherited from two different parents. If one is deferred and the other is effective, this will happen automatically: as soon as they have the same name (possibly after renaming), the effective version will serve as implementation. But if both are effective, you will need to uneffect one of them; the other's implementation will take precedence.

See "Rules on names", page 562.

- You may find a reusable class that is **too concrete** for your purposes, although the abstraction it describes serves your needs. Uneffecting will remove the unwanted implementations. Before using this solution, consider the alternatives: it is preferable to reorganize the inheritance hierarchy to make the more concrete class an heir of the new deferred class, rather than the reverse. But this is not always possible, for example if you do not have the authority to modify *A* and its inheritance hierarchy. Uneffecting may, in such cases, provide a useful form of generalization.

For a link of the uneffecting category, *B* will be deferred; *A* will normally be effective, but might be partially deferred.

Reification inheritance

We now come to the third and last general group, software inheritance.

Definition: reification inheritance

Reification inheritance applies if *A* represents a general kind of data structure, and *B* represents a partial or complete choice of implementation for data structures of that kind. *A* is deferred; *B* may still be deferred, leaving room for further reification through its own heirs, or it may be effective.

An example, used several times in earlier chapters, is a deferred class *TABLE* describing tables of a very general nature. Reification leads to heirs *SEQUENTIAL_TABLE* and *HASH_TABLE*, still deferred. Final reification of *SEQUENTIAL_TABLE* leads to effective classes *ARRAYED_TABLE*, *LINKED_TABLE*, *FILE_TABLE*.

The term "reification", from Latin words meaning "making into a thing", comes from the literary criticism of Georg Lukács. In computing science it is used as part of the VDM specification and development method.

Structure inheritance

Definition: structure inheritance

Structure inheritance applies if *A*, a deferred class, represents a general structural property and *B*, which may be deferred or effective, represents a certain type of objects possessing that property.

Usually *A* represents a mathematical property that a certain set of objects may possess; for example *A* may be the class *COMPARABLE*, equipped with such operations as **infix** "<" and **infix** ">=", representing objects to which a total order relation is applicable. A class that needs an order relation of its own, such as *STRING*, will inherit from *COMPARABLE*.

See "Numeric and comparable values", page 522.

It is common for a class to inherit from several parents in this way. For example class *INTEGER* in the Kernel Library inherits from *COMPARABLE* as well as from a class *NUMERIC* (with features such as **infix** "+" and **infix** "∗") representing its arithmetic properties. (Class *NUMERIC* more precisely represents the mathematical notion of ring.)

What is the difference between the structure and reification categories? With reification inheritance *B* represents the same notion as *A*, with more implementation commitment; with structure inheritance *B* represents an abstraction of its own, of which *A* covers only one aspect, such as the presence of an order relation or of arithmetic operations.

Waldén and Nerson note that novices sometimes believe they are using a similar form of inheritance when they are in fact mistaking a "contains" relation for "is" — as with *AIRPLANE* inheriting from *VENTILATION_SYSTEM*, a variant of the "car-owner" scheme, and just as wrong. They point out that it is easy to avoid this mistake through a criterion of the "absolute" kind, leaving no room for hesitation or ambiguity:

With the inheritance scheme, although the inherited properties are secondary, they are still properties of the **whole objects** *described by the class. If we make AIRPLANE inherit COMPARABLE to take account of an ordering relation on planes, the inherited features apply to each airplane as a whole; but the features of VENTILATION_SYSTEM* do not. *Feature stop of VENTILATION_SYSTEM is not supposed to stop the plane.*

Cited (with change of example) from [Waldén 1995], pages 193-194.

The conclusion in this example is clear: *AIRPLANE* must be a client, not an heir, of *VENTILATION_SYSTEM*.

Implementation inheritance

Definition: implementation inheritance

Structural inheritance applies if *B* obtains from *A* a set of features (other than constant attributes and once functions) necessary to the implementation of the abstraction associated with *B*. Both *A* and *B* must be effective.

Implementation inheritance is discussed in detail later in this chapter. A common case is the "marriage of convenience", based on multiple inheritance, where one parent provides the specification (reification inheritance) and the other provides the implementation (implementation inheritance).

"IMPLEMENTATION INHERITANCE", 24.8, page 844.

The case of inheriting constant attributes or once functions is covered by the next variant.

Facility inheritance

Facility inheritance is the scheme in which the parent is a collection of useful features meant only for use by descendants:

Definition: facility inheritance

Facility inheritance applies if *A* exists solely for the purpose of providing a set of logically related features for the benefit of heirs such as *B*. Two common variants are:

- *Constant inheritance* in which the features of *A* are all constants or once functions describing shared objects.
- *Machine inheritance* in which the features of *A* are routines, which may be viewed as operations on an abstract machine.

An example of facility inheritance was provided by class *EXCEPTIONS*, a utility class providing a set of facilities for detailed access to the exception handling mechanism.

See "ADVANCED EXCEPTION HANDLING", 12.6, page 431.

Sometimes, as in the examples given later in this chapter, a link of the facility kind uses only one of the two variants, constant or machine; but in others, such as *EXCEPTIONS*, the parent class provides both constants (such as the exception code

Incorrect_inspect_value) and routines (such as *trigger* to raise a developer exception). Since this discussion is meant to introduce disjoint inheritance categories, we should treat facility inheritance as a single category — with two (non-disjoint) variants.

With constant inheritance, both *A* and *B* are effective. With machine inheritance, there is more flexibility, but *B* should be at least as effective as *A*.

"FACILITY INHERITANCE", 24.9, page 847.

Facility inheritance is discussed in detail later in this chapter.

Using inheritance with deferred and effective classes

Each of the various categories reviewed places some requirements on which of the heir and the parent may be deferred and which may be effective. The following table summarizes the rules. "Variation" covers type variation and functional variation. Items marked • appear in more than one entry.

De
eff
an

Parent → ***Heir*** ↓	**Deferred**	**Effective**
Deferred	Constant• Restriction• Structure• Subtype• Uneffecting• Variation• View	Extension• Uneffecting•
Effective	Constant• Reification Structure• Subtype•	Constant• Extension• Implementation Restriction• Variation•

24.6 ONE MECHANISM, OR MORE?

"The dual perspective", page 494; *"The two styles", page 609*.

(Note: this discussion assumes as background the earlier presentation of "The meaning of inheritance", especially its section entitled "The dual perspective", and the presentation of descendant hiding, especially its section entitled "The two styles" with its summary table.)

The variety of uses of inheritance, evidenced by the preceding discussion, may lead to the impression that we should have several language mechanisms to cover the underlying notions. In particular, a number of authors have suggested separating between *module* inheritance, essentially a tool to reuse existing features in a new module, and *type* inheritance, essentially a type classification mechanism.

Such a division seems to cause more harm than good, for several reasons.

First, recognizing only two categories is not representative of the variety of uses of inheritance, reflected by the preceding classification. Since no one will advocate introducing ten different language mechanisms, the result would be too restrictive.

The practical effect would be to raise useless methodological discussions: assume you want to inherit from an iterator class such as *LINEAR_ITERATOR*; should you use module inheritance or type inheritance? One can find arguments to support either answer. You will waste your time trying to decide between two competing language mechanisms; the contribution of such reflections to the only goals that count — the quality of your software and the speed at which you produce it — is exactly zero.

An exercise asks you to analyze our categories to try to see for each of them whether it relates more to the "module" or "type" kind.

Exercise E24.8, page 870.

It is also interesting to think of the consequences that such a division will have on the complexity of the language. Inheritance comes with a number of auxiliary mechanisms. Most of them will be needed on both sides:

• *Redefinition* is useful both for subtyping (think of *RECTANGLE* redefining *perimeter* from *POLYGON*) and for module extension (the Open-Closed principle demands that when we inherit a module we keep the flexibility of changing what is not adapted any more to our new context — a flexibility without which we would lose one of the main attractions of the object-oriented method).

• *Renaming* is definitely useful for module inheritance. To present it as inappropriate for type inheritance (see [Breu 1995]) seems too restrictive. In the modeled external system, variants of a certain notion may introduce specific terminology, which it is often desirable for the software to respect. A class *STATE_INSTITUTIONS* in a geographical or electoral information system might have a descendant class *LOUISIANA_INSTITUTIONS* reflecting the peculiarities of Louisiana's political structures; it is not unreasonable to expect that the feature *counties*, giving the list of counties in a state, would be renamed *parishes* in the descendant, since parish is what Louisianians call what the rest of the US knows as a county.

• *Repeated inheritance* may occur with either form. Since we may expect that module-only inheritance will preclude polymorphic substitution, the problem of disambiguating dynamic binding, and hence the need for a **select** clause, will only arise for type inheritance; but all the other questions, in particular when to share repeatedly inherited features and when to replicate them, still arise.

• As always when we introduce new mechanisms into a language, they interact with the rest, and with each other. Do we prohibit a class from both module-inheriting and type-inheriting the same class? If so, we may be just vexing developers who have a good reason to use the same class in two different ways; if not, we open up a whole Pandora's box of new language issues — name conflicts, redefinition conflicts etc.

On Dijkstra's advice see "The need for methodology guidelines", page 664.

All this for the benefit of a purist's view of inheritance — restrictive and controversial. Not that there is anything wrong with defending controversial views; but one should be careful before imposing their consequences on language users — that is to say, on everyone. When in doubt, abstain. Once again, the contrast with Dijkstra's original **goto** excommunication is striking: Dijkstra took great care to explain in detail the drawbacks of the **goto** instruction, based on a theory of software construction and execution, and to explain what replacements were available. In the present case, no compelling argument — at least none that I have seen — shows why it is "bad" to use a single mechanism to cover both module and type inheritance.

Aside from blanket condemnations based on preconceived ideas of what inheritance should be, there is only one serious objection to the use of a single mechanism: the extra complication that this approach imposes on the task of **static type checking**. This issue was discussed at length in chapter 17; it places an extra burden on *compilers*, which is always justifiable (when the burden is reasonable, as here) if the effect is to facilitate the *developer*'s task.

In the end what all this discussion shows is that the ability to use only one inheritance mechanism for both module and type inheritance is not — as partisans of separate mechanisms implicitly consider — the result of a confusion of genres. It is the result of the *very first decision* of object-oriented software construction: the unification of module and type concepts into a single notion, the class. If we accept classes as both modules and types, then we should accept inheritance as both module accumulation and subtyping.

24.7 SUBTYPE INHERITANCE AND DESCENDANT HIDING

The first category on our list is probably the only form on which everyone agrees, at least everyone who accepts inheritance: what we may call pure subtype inheritance.

Most of the discussion will also apply to restriction inheritance, whose principal difference with subtype inheritance is that it does not require the parent to be deferred.

Defining a subtype

As was pointed out in the introduction of inheritance, part of the power of the idea comes from its fusion of a type mechanism, the definition of a new type as a special case of existing types, with a module mechanism, the definition of a module as extension of existing modules. Many of the controversial questions about inheritance come from perceived conflicts between these two views. With subtype inheritance there is no such question — although, as we shall see, this does not mean that everything becomes easy.

Subtype inheritance is closely patterned after the taxonomical principles of natural and mathematical sciences. Every vertebrate is an animal; every mammal is a vertebrate; every elephant is a mammal. Every group (in mathematics) is a monoid; every ring is a group; every field is a ring. Similar examples, of which we saw many in earlier chapters, abound in object-oriented software:

- *FIGURE* ◀— *CLOSED_FIGURE* ◀— *POLYGON* ◀— *QUADRANGLE* ◀— *RECTANGLE* ◀— *SQUARE*
- *DEVICE* ◀— *FILE* ◀— *TEXT_FILE*
- *SHIP* ◀— *LEISURE_SHIP* ◀— *SAILBOAT*
- *ACCOUNT* ◀— *SAVINGS_ACCOUNT* ◀— *FIXED_RATE_ACCOUNT*

and so on. In any one of these subtype links, we have clearly identified the set of objects that the parent type describes; and we have spotted a subset of these objects, characterized by some properties which do not necessarily apply to all instances of the parent. For example a text file is a file, but it has the extra property of being made of a sequence of characters — a property that some other files, such as executable binaries, do not possess.

A general rule of subtype inheritance is that the various heirs of a class represent disjoint sets of instances. No closed figure, for example, is both a polygon and an ellipse.

Several of the examples, such as *RECTANGLE* ◀— *SQUARE*, will most likely involve an effective parent, and so are cases of restriction inheritance.

Multiple views

Subtype inheritance is straightforward when a clear criterion exists to classify the variants of a certain notion. But sometimes several qualities vie for our attention. Even in such a seemingly easy example as the classification of polygons, doubt may arise: should we use the number of sides, leading to heirs such as *TRIANGLE*, *QUADRANGLE* etc., or should we divide our objects into regular polygons (*EQUILATERAL_POLYGON*, *SQUARE* and so on) and irregular ones?

Several strategies are available to address such conflicts. They will be reviewed as part of the study of view inheritance later in this chapter.

Enforcing the subtype view

A type is not just as a set of objects, of course: it is also characterized by the applicable operations (the features), and their semantic properties (the assertions: preconditions, postconditions, invariants). We expect the fate of features and assertions in the heir to be compatible with the concept of subtype — meaning that it must allow us to view any instance of the heir also as an instance of the parent.

The rules on assertions indeed support the subtype view:

- The parent's invariant is automatically part of the heir's invariant; so all the constraints that have been specified for instances of the parent also apply to instances of the heir.
- A routine precondition applies, possibly weakened, to any redeclaration of the routine: so any call which satisfies the requirement specified for instances of the parent will also satisfy the (equal or weaker) requirement specified for instances of the heir.

- A routine postcondition applies, possibly strengthened, to any redeclaration of the routine: so any property of the routine's outcome that has been specified for instances of the parent will be guaranteed to hold as a result of the (equal or stronger) properties specified for instances of the heir.

For features, the situation is a little more subtle. The subtype view implies that all operations applicable to an instance of the parent should be applicable to an instance of the heir. Internally, this is always true: even in the inheritance of *ARRAYED_STACK* from *ARRAY*, which seems far from subtype inheritance, the features of *ARRAY* were still available to the heir, and indeed were essential to the implementation of its *STACK* features. But in that case we had hidden all these *ARRAY* features from the heir's clients, and for good reason (we do not want a client of a stack class to perform arbitrary operations on the representation, such as directly modifying an array element, since this would be a violation of the class interface).

For pure subtype inheritance we might expect a much stronger rule: that *every* feature that a client can apply to instances of the parent class also be applicable, by that same client, to instances of the heir. In other words, no descendant hiding: if *B* inherits *f* from *A*, then the export status of *f* in *B* is at least as generous as in *A*. (That is to say: if *f* was generally exported, it still is; and if it was selectively exported to some classes, it is still exported to them, although it may be exported to more.)

The need for descendant hiding

In a perfect world we could indeed enforce the no-descendant-hiding rule; but not in the real world of software development. Inheritance must be usable even for classes written by people who do not have perfect foresight; some of the features they include in a class may not make sense in a descendant written by someone else, later and in a completely different context. We may call such cases **taxonomy exceptions**. (In a different context the word "exception" would suffice, but we do not want any confusion with the software notion of exception handling as studied in earlier chapters.)

Should we renounce inheriting from an attractive and useful class simply because of a taxonomy exception, that is to say because one or two of its features are inapplicable to our own clients? This would be unreasonable. We just hide the features from our clients' view, and proceed with our work.

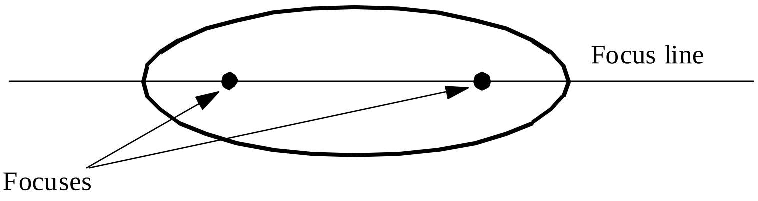
The alternatives have been studied as part of one of the founding principles of object technology — **Open-Closed principle** — and they are not attractive:

- We might modify the original class. This means we may invalidate myriads of existing systems that relied on it — no, thanks. In most practical cases, anyway, the class will not be ours to modify; we may not even have access to its source form.
- We might write a new version of the class (or, if we are lucky and do have access to its source code, make a copy), and modify it. This approach is the reverse of everything that object technology promotes; it defeats any attempt at reusability and at an organized software process.

Avoiding descendant hiding

Before probing further why and when we may need descendant hiding, it is essential to note that most of the time we do not. Descendant hiding should remain a technique of last resort. When you have a full grasp of the inheritance structure sufficiently early in the design process, *preconditions* are a better technique to handle apparent taxonomy exceptions.

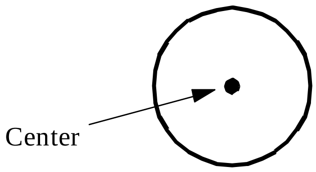
Consider class *ELLIPSE*. An ellipse has two focuses through which you can normally draw a line:



An ellipse and its focus line

Class *ELLIPSE* might correspondingly have a feature *focus_line*.

It is quite normal to define class *CIRCLE* as an heir to *ELLIPSE*: every circle is also an ellipse. But for a circle the two focuses are the same point — the circle's center — so there is no focus line. (It is perhaps more accurate to say that there is an infinity of focus lines, including any line that passes through the center, but in practice the effect is the same.)



A circle and its center

Is this a good example of descendant hiding? In other words, should class *CIRCLE* make feature *focus_line* secret, as in

```
class CIRCLE inherit
    ELLIPSE
        export {NONE} focus_line end
    …
```

Probably not. In this case, the designer of the parent class has all the information at his disposal to determine that *focus_line* is not applicable to all ellipses. Assuming the feature is a routine, it should have a precondition:

```
focus_line is
        -- The line through the two focuses
    require
        not equal (focus_1, focus_2)
    do
        …
    end
```

(The precondition could also be abstract, using a function *distinct_focuses*; this has the advantage that *CIRCLE* can redefine that function once and for all to yield false.)

See page 61.

Here the need to support ellipses without a focus line follows from a proper analysis of the problem. Writing an ellipse class with a function *focus_line* that has no precondition would be a design error; addressing such an error through descendant hiding would be attempting to cover up for that error. As was pointed out at the end of the presentation of the Open-Closed principle, erroneous designs must be fixed, not patched in descendants.

Applications of descendant hiding

The *focus_line* example is typical of taxonomy exceptions arising in application domains such as mathematics which can boast a solid theory with associated classifications, patiently refined over a long period. In such a context, the proper answer is to use a precondition, concrete or abstract, at the place where the original feature appears.

But that technique is not always applicable, especially in domains that are driven by human processes, with their attendant capriciousness that often makes it hard to foresee all possible exceptions.

Consider as an example a class hierarchy, rooted in a class *MORTGAGE*, in a software system for managing mortgages. The descendants have been organized according to various criteria, such as fixed rate versus variable rate, business versus personal or any other that was found appropriate; we may assume for simplicity that this is a taxonomy of the pure subtype kind. Class *MORTGAGE* has a procedure *redeem*, which handles the mechanisms for paying off a mortgage at a certain time earlier than maturation.

Now assume that Congress, in a fit of generosity (or under the pressure of construction lobbies), introduces a new form of government-backed mortgage whose otherwise advantageous conditions carry a provision barring any early redemption. We have found a proper place in the hierarchy for the corresponding class *NEW_MORTGAGE*; but what about procedure *redeem*?

We could use the technique illustrated with *focus_line*: a precondition. But what if there has never before in banker's memory existed a mortgage that could not be redeemed? Then procedure *redeem* probably does not have a precondition. (The situation is the same if the precondition existed but was concrete, so that it cannot be redefined.)

So if we decide to use a precondition we must modify class *MORTGAGE*. As usual, this assumes that we have access to its source code and the right to modify it — often not true. Suppose, however, that this is not a problem. We will add to *MORTGAGE* a boolean-valued function *redeemable* and to *redeem* a clause

```
require
    redeemable
```

But now we have changed the interface of the class. All the clients of the class and of its numerous descendants have instantly been made potentially incorrect; to observe the specification all calls *m*.*redeem* (…) should now be rewritten as

```
if m.redeemable then
    m.redeem (…)
else
    … (What in the world do we say here?) …
end
```

Initially this change is not urgent, since the incorrectness is only potential: existing software will only use the existing descendants of *MORTGAGE*, so no harm can result. But not fixing them means leaving a time bomb — unprotected calls to a precondition-equipped routine — ticking in our software. As soon as a client developer has the clever idea of using a polymorphic attachment with a source of type *NEW_MORTGAGE* but forgets the test we have a bug. And the compiler will not produce any diagnostic.

The absence of a precondition in the original version of *redeem* was not a design mistake on the part of the original designers: in their view of the world, until now correct, no precondition was needed. Every mortgage was redeemable. We cannot require every feature to have a precondition; imagine a world in which for every useful *f* there would be an accompanying boolean-valued function *f_feasible* serving as its bodyguard; then we would never be able to write a simple *x.f* for the rest of our lives; each call would be in an **if** … or equivalent as illustrated above for *m.redeem*. Not fun.

The *redeem* example is typical of taxonomy exceptions which, unlike *focus_line* and other cases of perfect-foresight classification, cannot be addressed through careful *a priori* precondition design. The observation made earlier fully applies: it would be absurd to renounce inheritance — the reuse of a rich class structure, lovingly developed and carefully validated — because a feature or two, out of dozens of useful ones, do not apply to our goal of the moment. We should just use descendant hiding:

```
class NEW_MORTGAGE inherit
    MORTGAGE
        export {NONE} redeem end
    …
```

No error or anomaly will be introduced in existing software — the existing class structure or its clients. If someone modifies a client class to include a polymorphic attachment with source type *NEW_MORTGAGE*, and the target of that attachment is also used with *redeem*, as in

```
m: MORTGAGE; nm: NEW_MORTGAGE
…
m := nm
…
m.redeem (…)
```

then the call becomes a catcall, and the potential error will be caught statically by the extended mechanism described in our discussion of typing.

"BEWARE OF POLYMORPHIC CATCALLS!", 17.7, page 637.

Taxonomies and their limitations

Taxonomy exceptions are not specific to software examples. Even — or perhaps especially — in the most established areas of natural science, it sometimes seems impossible to find a statement of the form "*members of the ABC phylum* [or genus, species etc.] *are characterized by property XYZ*" that is not prefaced by "*most*", qualified by "*usually*" or followed by "*except in a few cases*".This is true at all levels of the hierarchy, even the most fundamental categories, which a layman might naïvely believe to be established on indisputable criteria!

If you think for example that the distinction between the animal and plant kingdoms is simple, just ponder its definition in a popular reference text (italics added):

From: *the New York Public Library Science Desk Reference, ed. Patricia Barnes-Svarney, 1995.*

DISTINGUISHING PLANTS FROM ANIMALS

There are several *general* factors that distinguish plants from animals, *though* there are *numerous exceptions.*

Locomotion *Most* animals move about freely, *while* it is *rare* to find plants that can move around in their surrounding environments. *Most* plants are rooted in the soil, or attached to rocks, wood or *other materials.*

Food Green plants that contain chlorophyll manufacture food themselves, but *most* animals obtain nutrients by eating plants or other animals. […]

Growth Plants *usually* grow from the tips of their branches and roots, and at the outer layer of their stems, for their entire life. Animals *usually* grow in all parts of their bodies and stop growing after maturity.

Chemical regulation Though both plants and animals *generally* have hormones and other chemicals that regulate *certain* reactions within the organism, the chemical composition of these hormones differ[s] in the two kingdoms.

The same comments apply to another area of study, cultural rather than natural, which has also contributed to the development of systematic taxonomy: the historical classification of human languages.

In zoology a common example, so famous in Artificial Intelligence circles as to have become a cliché, still provides a good illustration of taxonomy exceptions. (Remember, however, that this is only an *analogy*, not a software example, and so cannot prove anything; it can only help us understand ideas whose relevance has been demonstrated otherwise.) Birds fly; in software terms class *BIRD* would have a procedure *fly*. Yet if we wanted a class *OSTRICH* we would have to admit that ostriches, although among the birdest of birds, do not fly.

We could think of classifying birds into flying and non-flying categories. But this would conflict with other possible criteria including, most importantly, the commonly retained one, shown on the next page.

- **Kingdom: Animalia** — multicellular organisms without chlorophyll
  - **Phylum: Chordata** —coelemic cavity, 3 germ layers, a notocord, an endoskeleton and a closed circulatory system
    - **Class: Aves** birds (there are 30
      - **Order: Anseriformes** — waterfowl
      - **Order: Apodiformes** — swifts and hummingbirds
      - **Order: Caprimulgiformes** — nightjars, potoos, frogmouths, owlet- frogmouths and oilbirds
      - **Order: Casuariiformes** cassowaries and emu
      - **Order: Chardriiformes** — shorebirds
      - **Order: Ciconiiformes** — long-legged wading birds
      - **Order: Coliiformes** — mousebirds
      - **Order: Columbiformes** — pigeons and doves
      - **Order: Coraciiformes** — kingfishers
      - **Order: Cuciliformes** — cuckoos
      - **Order: Dinornithiformes** — kiwis and moas
      - **Order: Falconiformes** — raptors
      - **Order: Galliformes** — gallinaceous birds (chickens, grouse, quail and pheasant)
      - **Order: Gaviiformes** — loons
      - **Order: Gruiformes** — terrestrial and marsh birds
      - **Order: Musophagiformes** — turacos
      - **Order: Passeriformes** — perching birds, songbirds and passerines
      - **Order: Pelecaniformes** — waterbirds with webbed feet
      - **Order: Phoenicopteriformes** — flamingos
      - **Order: Piciformes** — woodpeckers
      - **Order: Podicipediformes** — grebes
      - **Order: Procellariiformes** — tube-nosed seabirds
      - **Order: Psittaciformes** — parrots, macaws
      - **Order: Pteroclidiformes** — sandgrouse
      - **Order: Rheiformes** — rheas, nandus
      - **Order: Sphenisciformes** — penguins
      - **Order: Strigiformes** — owls
      - **Order: Struthioniformes** — ostrich
      - **Order: Tinamiformes** — tinamous
      - **Order: Trogoniformes** — trogons and quetzals

General classification of birds

(*Data from Ed Everham, at* www.runet.edu/~eeverham.) *Reproduced with the author's permission. Associated comments are reproduced in* "The arbitrariness of classifications", page 859.

The *OSTRICH* example has an interesting twist. Although regrettably most of them do not seem to be aware of it, ostriches really should fly. Younger generations lost this ancestral skill through an accident of evolutionary history, but anatomically ostriches have retained most of the aeronautical machinery of birds. This property, which makes the job of the professional taxonomist a little harder (although it may facilitate that of his colleague, the professional taxidermist), will not in the end prevent him from classifying ostriches among birds.

In software terms *OSTRICH* will simply inherit from *BIRD* and hide the inherited *fly* feature.

Using descendant hiding

All our efforts [at classification] *are powerless against the multiple relations which from everywhere affect the living beings around us. This is the fight, described by the great botanist Goethe, between Man and Nature in her infinity. One can be sure that Man will always be defeated.*

Henri Baillon, *General Study of the Euphorbiaceous Family* (1850). Quoted (in French) in Peter F. Stevens, *The Development of Biological Systematics*: *Antoine-Laurent de Jussieu, Nature, and the Natural System*, Columbia University Press, New York, 1994.

The preceding evidence, from both software practice and non-software analogies, suggests that even with a careful design some taxonomy exceptions may remain. Hiding *redeem* from *NEW_MORTGAGE* or *fly* from *OSTRICH* is not necessarily a sign of sloppy design or insufficient foresight; it is the recognition that other inheritance hierarchies that would not require descendant hiding could be more complex and less useful.

Such taxonomy exceptions have the precedent of centuries of effort by intellectual giants (including Aristotle, Linné, Buffon, Jussieu and Darwin). They may even signal some intrinsic limitation of the human ability to comprehend the world. Could they be related to the indeterminacy results that shook scientific thought in the twentieth century, uncertainty in physics and undecidability in mathematics?

All this assumes that descendant hiding remains, as already noted, a rare occurrence. If you design a taxonomy with taxonomy exceptions all over — well, they are not exceptions any more, so you do not really have much of a taxonomy.

In software, for those few cases in which conflicting classification criteria or massive previous work precludes the production of a perfect subtype hierarchy, descendant hiding is more than a convenient facility: it will save your neck.

24.8 IMPLEMENTATION INHERITANCE

A form of inheritance that has often been criticized but is in fact both convenient and conceptually valid is the use of an inheritance link between a class describing a certain implementation of an abstract data structure and the class providing the implementation.

The marriage of convenience

In the discussion of multiple inheritance we saw an example of the "marriage of convenience" kind, which combines a deferred class with a mechanism to implement it. The example was *ARRAYED_STACK*, of the general form

"The marriage of convenience", page 530.

```
class ARRAYED_STACK [G] inherit
    STACK [G]
        redefine change_top end
    ARRAY [G]
        rename
            count as capacity, put as array_put
        export
            {NONE} all
        end
feature
    … Implementation of the deferred routines of STACK, such as put, count, full,
      and redefinition of change_top, in terms of ARRAY operations…
end
```

It is interesting to compare *ARRAYED_STACK*, as sketched here, with the class *STACK2* of an earlier discussion — an array implementation of stacks defined without any use of inheritance. Note in particular how avoiding the need for the class to be a client of *ARRAY* simplifies the notation (the previous version had to use *implementation*.*put* where we can now just write *put*).

STACK2 appeared on page 350.

In the above inheritance part for *ARRAY* all features have been made secret. This is typical of marriage-of-convenience inheritance: all the features from the specification-providing parent, here *STACK*, are exported; all the features from the implementation-providing parent, here *ARRAY*, are hidden. This forces clients of class *ARRAY_STACK* to use the corresponding instances through stack features only; we do not want to let them perform arbitrary array operations on the representation, such as changing the value of an element other than the top one.

It feels so good, but is it wrong?

Implementation inheritance is not without its critics. That we hide many inherited features seems to some people a violation of the "is-a" principle of inheritance.

It is not. There are different forms of "is-a". By its behavior, an arrayed stack is a stack; but internally it is an array. In fact the representation of an instance of *ARRAYED_STACK* is exactly the same as that of an instance of *ARRAY*, enriched with one attribute (*count*).

Being made in the same way is a rather strong form of "is-a". And it is not just the representation: all the features of *ARRAY*, such as *put* (renamed *array_put*), **infix** "@" and *count* (renamed *capacity*) are available to *ARRAYED_STACK*, although not exported to its clients; the class needs them to implement the *STACK* features.

So there is nothing conceptually wrong with such implementation-only inheritance. The comparison with the counter-example studied at the beginning of this chapter is striking: for *CAR_OWNER* we had a gross misunderstanding of the concept; with *ARRAYED_STACK* we have a well-identified form of the "is-a" relationship.

There is one drawback: permitting the inheritance mechanism to restrict the export availability of an inherited feature — that is to say, permitting the **export** clause — makes static type checking more difficult, as we have studied in detail. But this difficulty is largely for the compiler writer, not for the software developer.

Doing without inheritance

Page 350.

Let us probe further and see what it would take to work without implementation inheritance in our example case. This has been seen already: class *STACK2* of an earlier chapter. It has an attribute *representation* of type *ARRAY [G]* and stack procedures implemented in the following style (assertions omitted):

```
put (x: G) is
        -- Add x on top.
    require
        …
    do
        count := count + 1
        representation.put (count, x)
    ensure
        …
    end
```

Every manipulation of the representation requires a call to a feature of *ARRAY* with *representation* as the target. There is a performance penalty: minor for space (the *representation* attribute), more serious for time (going through *representation*, that is to say adding an indirection, for each operation).

Assume we can ignore the efficiency issue. Tediousness is another, with all the "*representation*." prefixes that you must add before every array operation. This will be true in all the classes that implement various data structures — stacks, but also lists, queues and others — through arrays.

The object-oriented designer hates tedious, repetitive tasks. "Encapsulate repetition" is our motto. If we see such a pattern occurring repeatedly throughout a set of classes, the natural and healthy reaction is to try to understand the common abstraction, and encapsulate it in a class. The abstraction here is something like "data structure that has access to an array and its operations". The class could be:

```
indexing
    description: "Objects that have access to an array and its operations"
class
    ARRAYED [G]
feature -- Access
    item (i: INTEGER): G is
            -- The representation's element at index i
        require
            …
        do
            Result := representation.item (i)
        ensure
            …
        end
feature -- Element change
    put (x: G; i: INTEGER) is
            -- Replace by x the representation's element at index i.
        require
            …
        do
            representation.put (x, i)
        ensure
            …
        end
feature {NONE} -- Implementation
    representation: ARRAY [G]
end -- class ARRAYED
```

The features *item* and *put* have been exported. Since *ARRAYED* only describes internal properties of a data structure, it does not really need exported features. So someone who disagrees with the very idea of letting a descendant hide some of its parents' exported features may prefer to make all the features of *ARRAYED* secret. They will then by default remain secret in descendants.

With this class definition it becomes quite uncontroversial to make classes such as *ARRAYED_STACK* or *ARRAYED_LIST* inherit from *ARRAYED*: they indeed describe "arrayed" structures. These classes can now use *item* instead of *representation.item* and so on; we have rid ourselves of the tediousness.

But wait a minute! If it is right to inherit from *ARRAYED*, why can we not inherit directly from *ARRAY*? We gain nothing from the further layer or encapsulation that we have thrown over *ARRAY* — a form of encapsulation that starts looking more like obfuscation. By going through *ARRAYED* we are just pretending to ourselves that we are not using implementation inheritance, but for all practical purposes we are. We have just made the software more complex and less efficient.

There is indeed no reason in this example for class *ARRAYED*. Direct implementation inheritance from classes such as *ARRAY* is simpler and legitimate.

24.9 FACILITY INHERITANCE

With facility inheritance we are even less coy than with implementation inheritance about why we want the marriage: pure, greedy self-interest. We see a class with advantageous features and we want to use them. But there is nothing to be ashamed of: the class has no other *raison d'être*.

Using character codes

The Base Libraries include a class *ASCII*:

```
indexing
    description:
        "The ASCII character set. %
        %This class may be used as ancestor by classes needing its facilities."
class ASCII feature -- Access
    Character_set_size: INTEGER is 128; Last_ascii: INTEGER is 127
    First_printable: INTEGER is 32; Last_printable: INTEGER is 126
    Letter_layout: INTEGER is 70
    Case_diff: INTEGER is 32
        -- Lower_a – Upper_a
    …
    Ctrl_a: INTEGER is 1; Soh: INTEGER is 1
    Ctrl_b: INTEGER is 2; Stx: INTEGER is 2
    …
    Blank: INTEGER is 32; Sp: INTEGER is 32
    Exclamation: INTEGER is 33; Doublequote: INTEGER is 34
    …
    …
    Upper_a: INTEGER is 65; Upper_b: INTEGER is 66
    …
    Lower_a: INTEGER is 97; Lower_b: INTEGER is 98
    … etc. …
end -- class ASCII
```

This class is a repertoire of constant attributes (142 features in all) describing properties of the ASCII character set. As the *description* entry states, it is meant to be inherited by classes needing access to such properties.

Consider for example a lexical analyzer — the part of a language analysis system that is responsible for identifying the basic elements, or *tokens*, of an input text; these tokens may be (assuming the input is a text in some programming language) integer constants, identifiers, symbols and so on. One of the classes of the system, say *TOKENIZER*, will need access to the character codes, to classify the input characters into digits, letters etc. Such a class will inherit these codes from *ASCII*:

```
class TOKENIZER inherit ASCII feature
        … Routines here may use such features as Blank, Case_diff etc. …
end
```

Classes such as *ASCII* have been known to raise a few eyebrows; before going into the methodological discussion of whether they are a proper application of inheritance, we will look at another example of facility inheritance.

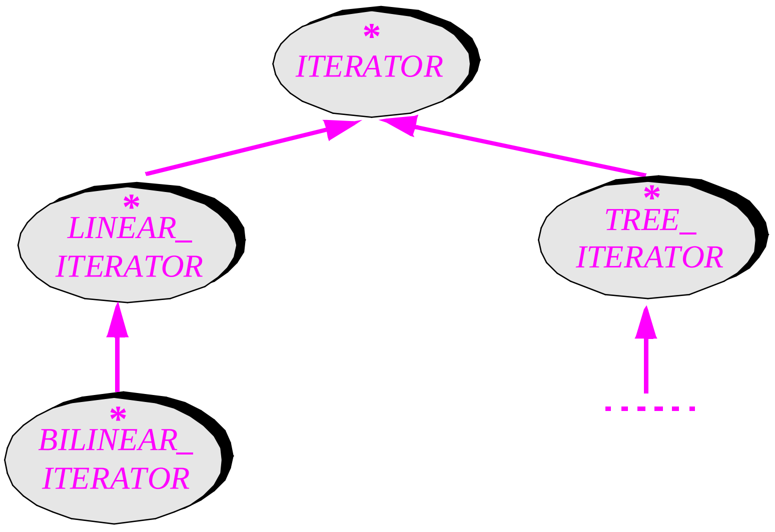
Iterators

The second example will show a case in which the inherited features are not just constant attributes (as with *ASCII*) but routines of the most general kind.

Assume that we want to provide a general mechanism to iterate over data structures of a certain kind, for example linear structures such as lists. "Iterating" means performing a certain procedure, say *action*, on elements of such a structure, taken in their sequential order. We are asked to provide a number of iteration mechanisms, including: applying *action* to all the elements; applying it to all the elements that satisfy a certain criterion given by a boolean-valued function *test*; applying it to all the elements up to the first one that satisfies *test*, or the first one that does not satisfy this condition; and so on. A system that uses the mechanism must be able to apply it to any *action* and *test* of its choice.

Exercise E24.7, page 870.

At first it might seem that the iterating features should belong to the data structure classes themselves, such as *LIST* or *SEQUENCE*; but as an exercise invites you to determine for yourself this is not the right solution. It is preferable to introduce a separate hierarchy for iterators:



Class *LINEAR_ITERATOR*, the one of interest for this discussion, looks like this:

```
indexing
    description:
        "Objects that are able to iterate over linear structures"
    names: iterators, iteration, linear_iterators, linear_iteration
deferred class LINEAR_ITERATOR [G] inherit
    ITERATOR [G]
        redefine target end
feature -- Access
    invariant_value: BOOLEAN is
            -- The property to be maintained by an iteration (default: true).
        do
            Result := True
        end

    target: LINEAR [G]
            -- The structure to which iteration features will apply

    test: BOOLEAN is
            -- The boolean condition used to select applicable elements
        deferred
        end

feature -- Basic operations
    action is
            -- The action to be applied to selected elements.
        deferred
        end

    do_if is
            -- Apply action in sequence to every item of target that satisfies test.
        do
            from start invariant invariant_value until exhausted loop
                if test then action end
                forth
            end
        ensure then
            exhausted
        end
    … And so on: do_all, do_while, do_until etc. …

end -- class LINEAR_ITERATOR
```

Now assume a class that needs to perform a certain operation on selected elements of a list of some specific type; for example a command class in a text processing system may need to justify all paragraphs in a document, excepted for preformated paragraphs (such as program texts and other display paragraphs). Then:

```
class JUSTIFIER inherit
    LINEAR_ITERATOR [PARAGRAPH]
        rename
            action as justify,
            test as justifiable,
            do_all as justify_all
        end
feature
    justify is
        do … end
    justifiable is
            -- Is paragraph subject to justification?
        do
            Result := not preformated
        end
    …
end -- class JUSTIFIER
```

The renaming was not indispensable but helps for clarity. Note that there is no need to declare or redeclare the procedure *justify_all* (the former *do_all*): as inherited, it does the expected job based on the effected versions of *action* and *test*.

Procedure *justify*, instead of being described in the class, could be inherited from another parent. In this case multiple inheritance would perform a "join" operation that effects the deferred *action*, inherited from one parent under the name *justify* (here the renaming is essential), with the effective *justify* inherited from the other parent. A form of marriage of convenience, in fact.

LINEAR_ITERATOR is a remarkable example of **behavior class**, capturing common behaviors while leaving specific components open so that descendants can plug in their specific variants.

"Don't call us, we'll call you", page 505.

Forms of facility inheritance

The two examples, *ASCII* and *LINEAR_ITERATOR*, are typical of the two main variants of facility inheritance:

- *Constant* inheritance, in which the parent principally yields constant attributes and shared objects.
- *Operation* inheritance, in which it yields routines.

As noted earlier, it is possible to combine both of these variants in a single inheritance link. That is why facility inheritance is one of our categories, not two.

Understanding facility inheritance

To some people facility inheritance appears to be an abuse of the mechanism — a form of hacking. But that is not necessarily the case.

The main question to consider in these examples is not about inheritance but about the classes that have been defined, *ASCII* and *LINEAR_ITERATOR*. As always when looking at a class design, we must ask ourselves: "Does this indeed describe a meaningful data abstraction?" — a set of objects characterized by their abstract properties.

With the examples the answer is less obvious than with a class *RECTANGLE*, *BANK_ACCOUNT* or *LINKED_LIST*, but it exists all the same:

- Class *ASCII* represents the abstraction: "any object that has access to the properties of the ASCII character set".
- Class *LINEAR_ITERATOR* represents the abstraction: "any object that has the ability to perform sequential iterations on a linear structure". Such objects tend to be of the "machine" kind described in the preceding chapter.

"Objects as machines", page 751.

Once these abstractions have been accepted, the inheritance links do not raise any problem: an instance of *TOKENIZER* does need "access to the properties of the ASCII character set", and an instance of *JUSTIFIER* does need "the ability to perform sequential iterations on a linear structure". In fact, we could classify such examples of inheritance links under the subtype kind. What distinguishes facility inheritance is the nature of the parent.

That the classes themselves are the issue, not the use of inheritance, is reinforced by the observation that an application class could rely on these classes as a client rather than heir. This would make things heavier, especially for *ASCII*: with

```
charset: ASCII
…
!! charset
```

On iterator objects see exercise E15.4, page 567.

every use of a character code would have to be written *charset*.*Lower_a* and the like. The object attached with *ASCII* does not play any useful role. With *LINEAR_ITERATOR* the same comments apply as long as a given class needs only one kind of iteration. If several are required, it becomes interesting to create iterator objects, each with its own version of *action* and *test*; then you can have as many iteration schemes as you need.

If it is appropriate to have iterator objects, we need iterator classes, and there is no reason to deny such classes the right to join the inheritance club.

24.10 MULTIPLE CRITERIA AND VIEW INHERITANCE

Perhaps the most difficult problem of using inheritance arises when alternative criteria are available to classify the abstractions of a certain application area.

Classifying through multiple criteria

The traditional classifications of the natural sciences use a single criterion (possibly involving several qualities) at each level: vertebrate versus invertebrate, leaves renewed each year or not, and so on. The result is what we would call single inheritance hierarchies, whose main advantage is their great simplicity. But there are problems too, since nature is definitely not single-criterion. This will be obvious to anyone who has ever tried to take a nature walk armed with a botanical book meant to enable plant recognition through the official Linnaean criteria. Species A is deciduous and species B is not, the book says; how long can you afford to wait, if this is July, to find out whether the leaves remain? You are told that June will bring bright purple flowers, but how can you tell in the midst of January? The roots of A are at most 7 meters deep, versus at least 9 for B — must you dig?

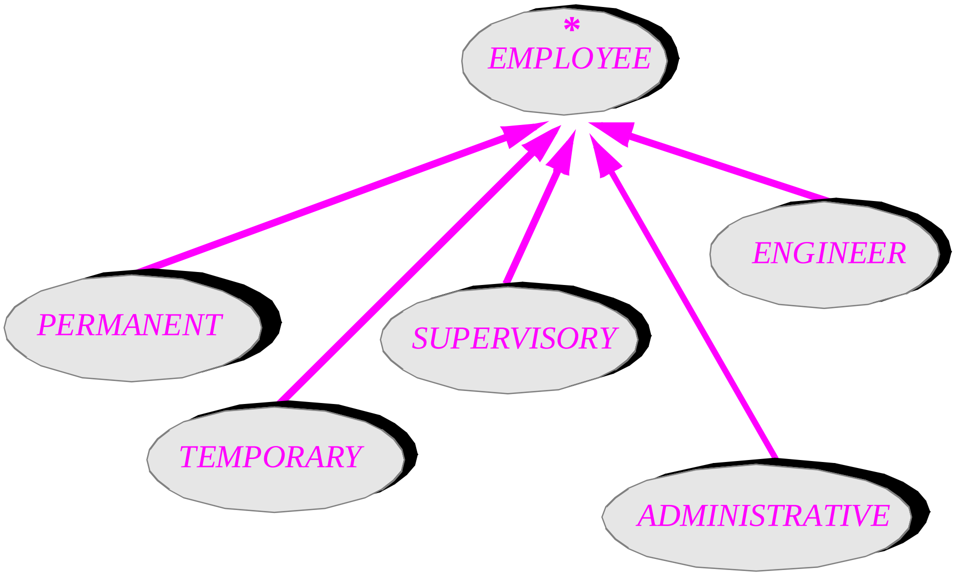
In software, when a single criterion seems too restrictive, we can use all the techniques of multiple and especially repeated inheritance that we have learned to master in earlier chapters. Assume for example a class *EMPLOYEE* in a personnel management system. Assume further that we have two separate criteria for classifying employees:

- By contract type, such as permanent *vs*. temporary.
- By job type, such as engineering, administrative, managerial.

and that both of these criteria have been recognized to lead to valid descendant classes; in other words you are not engaging in taxomania, since the classes that you have identified, such as *TEMPORARY_EMPLOYEE* for the first criterion and *MANAGER* for the second, are truly characterized by specific features not applicable to the other categories. What do you do?

A first attempt might introduce all the variants at the same level:

A messy classification



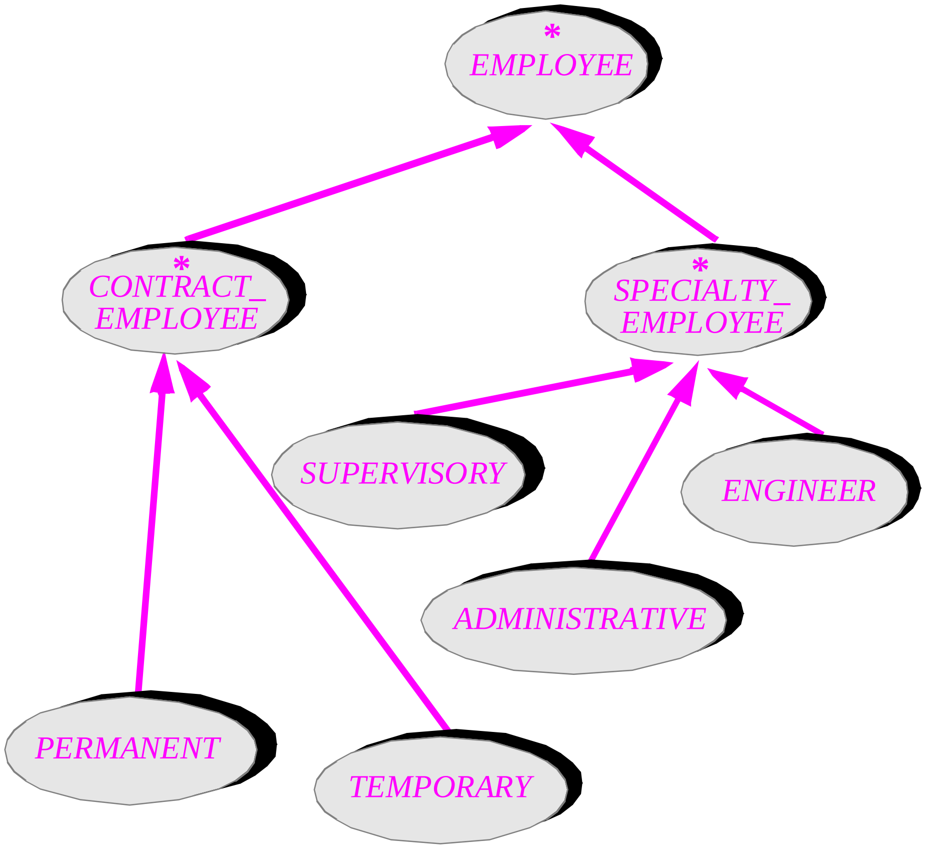
To keep this sketched example small and the figure simple, the class names have been abbreviated. To go from this example to a real system we would have to apply the usual naming guidelines, which suggest longer and more accurate names such as *PERMANENT_EMPLOYEE*, *ENGINEERING_EMPLOYEE* and so on.

This inheritance hierarchy is not satisfactory since widely different concepts are represented by classes at the same level.

View inheritance

Classification through views

If you retain the idea of using inheritance for the classification used in the example under discussion, you should introduce an intermediate level to describe the competing classification criteria:



Note that the name *CONTRACT_EMPLOYEE* does not mean “employee that has a contract” (as opposed to employees who might not have one!), but “employee as characterized by his contract”. The name of the sibling class similarly means “employee as characterized by his specialty”.

That these names seem far-fetched reflects a certain uneasiness, typical of this kind of inheritance. In subtype inheritance we encountered the rule that the sets of instances represented by the various heirs to a class be disjoint. Here the rule does not apply: a permanent employee, for example, may be an engineer too. This means that such a classification is meant for repeated inheritance: some proper descendants of the classes shown in the figure will have both *CONTRACT_EMPLOYEE* and *SPECIALTY_EMPLOYEE* as ancestors — not directly, but for example by inheriting from both *PERMANENT* and *ENGINEER*. Such classes will be repeated descendants of *EMPLOYEE*.

This form of inheritance may be called view inheritance: various heirs of a certain class represent not disjoint subsets of instances (as in the subtype case) but various ways of classifying instances of the parent. Note that this only makes sense if both the parent and the heirs are deferred classes, that is to say, classes describing general categories rather than fully specified objects. Our first attempt at *EMPLOYEE* classification by views (the one that had all descendants at the same level) violated that rule; the second one satisfies it.

Is view inheritance appropriate?

View inheritance is relatively far from the more common uses of inheritance and is subject to criticism. The reader will be judge of whether to use it for his own purposes, but in any case we should examine the pros and cons.

It should be clear that — like repeated inheritance, which it requires — view inheritance is **not a beginner's mechanism**. The rule of prudence that was introduced for repeated inheritance holds here: if you have less than a few months' hands-on experience with O-O development of significant projects, better stay away from view inheritance.

The alternative to view inheritance is to choose one of the classification criteria as primary, and use it as the sole guide for devising the inheritance hierarchy; to address the other criteria, you will use specific features. It is interesting to note that many modern zoologists and botanists use this approach: their basic classification criterion is the reconstructed evolutionary history of the genera and species involved. Would it that we always had such a single, indisputable standard to guide us in devising software taxonomies.

To stick to a single primary criterion in our example we could decide that the job type is the factor of principal interest, and represent the employment status by a feature. As a first attempt, the feature (in class *EMPLOYEE*) could be

is_permanent: *BOOLEAN*

but this is dangerously constraining; to extend the possibilities, we could have

Permanent: *INTEGER* **is unique**

Temporary: *INTEGER* **is unique**

Contractor: *INTEGER* **is unique**

…

but then we have learned to be wary, for good reasons, of explicit enumerations. A better approach is to introduce a class *WORK_CONTRACT*, most likely deferred, with as many descendants as necessary to account for specific kinds of work contract. Then we can stay away from loathed explicit discriminations of the form

if *is_permanent* **then** … **else** … **end**

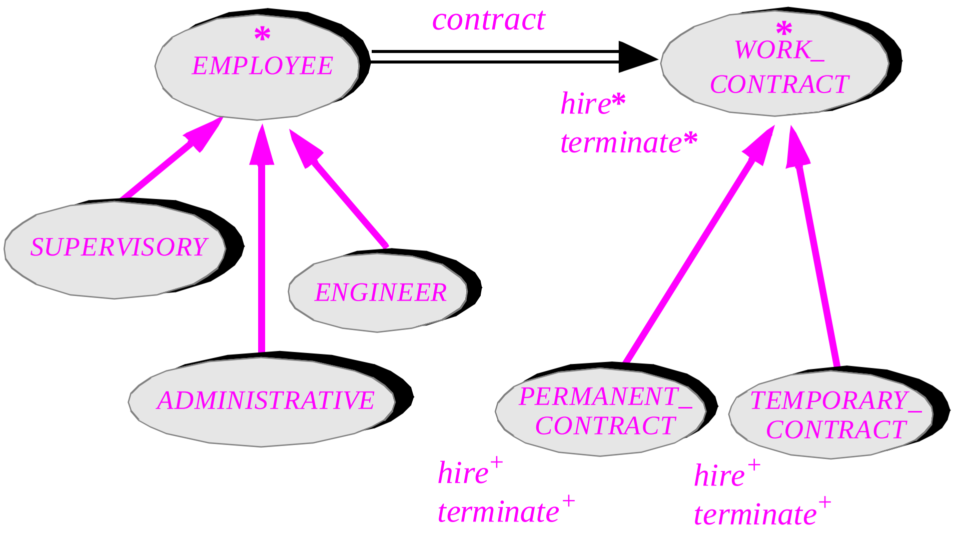
or

```
inspect
        contract_type
when Permanent then
        …
when …
        …
end
```

with their contingent of future extendibility troubles (stemming from their violation of just about every modularity principle: continuity, single choice, open-closedness); instead, we will equip class *WORK_CONTRACT* with deferred features representing contract-type-dependent operations, which will then be effected differently in descendants. Most of these features will need an argument of type *EMPLOYEE*, representing the employee to which the operation is being applied; examples might include *hire* and *terminate*.

The resulting structure will look like this:

Multi-criteria classification through separate, client-related hierarchies



See "AN APPLICATION: THE HANDLETECHNIQUE", 24.3, page 817.

This scheme, as you may have noted, is almost identical to the **handle**-based design pattern described earlier in this chapter.

Such a technique may be used in place of view inheritance. It does complicate the structure by introducing a separate hierarchy, a new attribute (here *contract*) and the corresponding client relations. It has the advantage that the abstractions in such a hierarchy are beyond question (work contract, permanent work contract); with the view inheritance solution, the abstractions are clear too but a little trickier to explain ("employee seen from the perspective of his work contract", "employee seen from the perspective of his specialty").

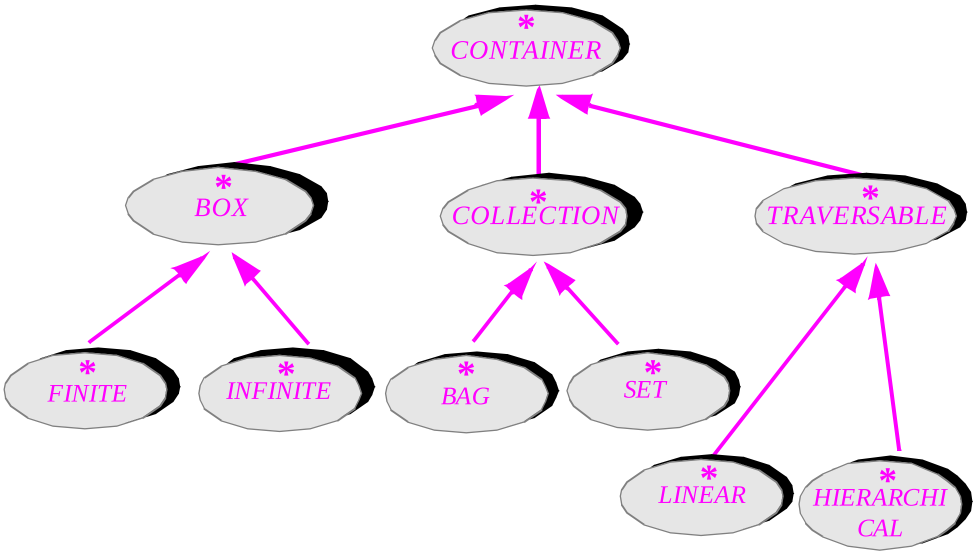
Criteria for view inheritance

It is not uncommon to think of view inheritance early in the analysis of a problem domain, while you are still struggling with the fundamental concepts and considering several possible classification criteria, all of which vie for your attention. As you improve your understanding of the application area, it will often happen that one of the criteria starts to dominate the others, imposing itself as the primary guide for devising the inheritance structure. In such cases, the preceding discussion strongly suggests that you should renounce view inheritance in favor of more straightforward techniques.

I still find view inheritance useful when the following three conditions are met:

- The various classification criteria are equally important, so any choice of a primary one would be arbitrary.
- Many possible combinations (such as, in the earlier example, permanent supervisor, temporary engineer, permanent engineer and so on) are needed.
- The classes under consideration are so important as to justify spending significant time to get the best possible inheritance structure. This applies in particular when the classes are part of a **reusable library** with large reuse potential.

An example of application of these criteria is the uppermost structure of the Base libraries, in the environment described in the last chapter of this book. The resulting classes followed from an effort, described in detail in the book [M 1994a], of applying taxonomical principles to the systematic classification of computing science's basic structures, in the tradition of the natural scientists. The highest part of the "container" structure looks like this:



A view-based classification of fundamental computing structures

The first-level classification (*BOX*, *COLLECTION*, *TRAVERSABLE*) is view-based; the level below it (and many of those further below, not shown) is a subtype classification. A container structure is characterized through three criteria:

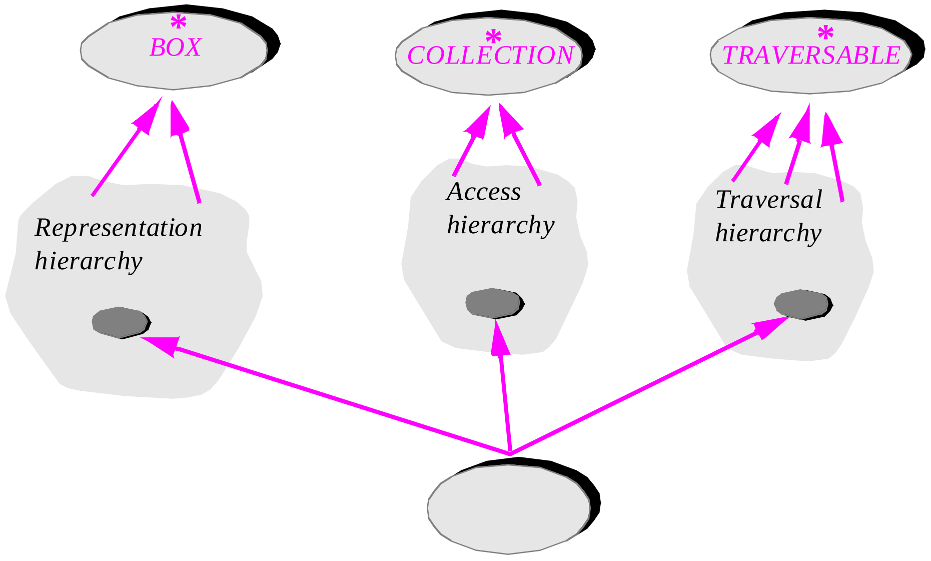
- How items will be accessed: *COLLECTION*. A *SET* makes it possible to find out whether an item is present, whereas a *BAG* also enables the client to find out the number of occurrences of a given element. Further refinements include such access abstractions as *SEQUENCE* (items are accessed sequentially), *STACK* (items are accessed in the reverse order of their insertion) and so on.

- How items will be represented: *BOX*. Variants include finite and infinite structures. A finite structure can be bounded or unbounded; a bounded structured can be fixed or resizable.

- How the structure can be traversed: *TRAVERSABLE*.

It is interesting to note that the hierarchy did not start out as view inheritance. The initial idea was to define *BOX*, *COLLECTION* and *TRAVERSABLE* as unrelated classes, each at the top of a separate hierarchy; then, when describing any particular data structure implementation, to use multiple inheritance to pick one parent from each of the three parts. For example a linked list is finite and unbounded (representation), sequentially accessed (access), and linearly traversable (traversal):

Building a data structure class by combination of abstractions through multiple inheritance



But then we realized that it was inappropriate to keep *BOX*, *COLLECTION* and *TRAVERSABLE* separate: they all needed a few common features, in particular *has* (membership test) and *empty* (test for no elements). This clearly indicated the need for a common ancestor — *CONTAINER*, where these common features now appear. Hence a structure that was initially designed as pure multiple inheritance, with three disjoint hierarchies at the top, turned out to be a view inheritance hierarchy with a considerable amount of repeated inheritance.

Although initially difficult to get right, this structure has turned out to be useful, flexible and stable, confirming both of the conclusions of this discussion: that view inheritance is not for the faint of heart; and that when applicable it can play a key role for complex problem domains where many criteria interact — if the effort is justified, as in a fundamental library of reusable components, which simply has to be done right.

24.11 HOW TO DEVELOP INHERITANCE STRUCTURES

Some of the material in this section is from [M 1995].

When you read a book or pedagogical article on the object-oriented method, or when you discover a class library, the inheritance hierarchies that you see have already been designed, and the author does not always tell you how they got to be that way. How then do you go about designing your own structures?

Specialization and abstraction

Voluntarily or not, many pedagogical presentations tend to create the impression that inheritance structures should be designed from the most general (the upper part) to the most specific (the leaves). This is in part because this is often the best way to *describe* a good structure once it exists: from the general to the particular; from the figures to the closed figures to the polygons to the rectangles to the squares. But the best way to describe a structure is not necessarily the best way to *produce* it.

A similar comment, due to Michael Jackson, was mentioned in the discussion of top-down design.

See "Production and description", page 114.

In an ideal world populated with perfect people, we would always recognize the proper abstractions right away, and then draw the categories, their subcategories and so on. In the real world, however, we often see a specific case before we discover the general abstraction of which it is but a variant.

In many cases the abstraction is not unique; how best to generalize a certain notion depends on what you or your clients will most likely want to do with the notion and its variants. Consider for example a notion that we have often encountered in earlier discussion: points in a two-dimensional space. At least four generalizations are possible:

- Points in arbitrary-dimension space — leading to an inheritance structure where the sisters of class *POINT* will be classes *POINT_3D* and so on.
- Geometrical figures — the other classes in the structure being the likes of *FIGURE*, *RECTANGLE*, *CIRCLE* and so on.
- Polygons — with other classes such as *QUADRANGLE* (four vertices), *TRIANGLE* (three vertices) and *SEGMENT* (two vertices), *POINT* being the special polygon with just one vertex.
- Objects that are entirely determined by two coordinates — the other contenders here being *COMPLEX* and *VECTOR_2D*.

Although some of these generalizations may intuitively be more appealing than others, it is impossible to say in the absolute which one of them is the best. The answer will depend on how your software base evolves and what it will need. So a prudent process in which you sometimes abstract a bit too late, because you waited until you were sure that you had found the most useful path of generalization, may be preferable to one in which you might get too much untested abstraction too soon.

The arbitrariness of classifications

The *POINT* example is typical. When presented with two competing classifications of a certain set of abstractions, you will often be able to determine, based on rational arguments, which one is *better*; but seldom is one in a position to determine that a certain inheritance structure is the *best* possible one.

This situation is not specific to software. Do not believe, for example that the Linnaean classifications of natural science are universally accepted or eternal. The maintainers of the "Tree of Life" Internet archive mentioned earlier (see also the bibliographical notes) state at the outset that the project's classification — however collaborative and interdisciplinary — is controversial. And this is not just for weird smallish creatures too viscous to be discussed at lunch; Dr. Everham's Web classification of birds cited earlier comes with the comment

See page 842. Bibliographic references removed.

There are 174 Families, 2044 Genera and 9021 species of birds in the world! The most abundant species are in the order Passeriformes with 5276 species. The least number of species in an order is 1: the Ostrich in Struthioniformes. (I would have thought the Ostrich would be in an order with the Emus, Kiwis and Moas, all extinct, because they all are flightless with stout legs and longish necks.) The Linnaeus system groups organisms based on morphological similarities. Another classification of animals is based on DNA-DNA hybridization. This is highly complex; for example an American Cuckoo would be classified as: Kingdom, Animalia; Phylum, Chordata; Class, Aves; Subclass, Neornithes; Infraclass, Neoaves; Parvclass, Passerae; Superorder, Cuculimorphae; Order, Cuculiformes; Infraorder, Cuculides; Parvorder, Coccyzida; Family, Coccyzidae.

More on competing classification methods at the end of this chapter.

This shows the competition between two systems: the traditional one, based on morphology (and evolution); and a more inductive one based on DNA analysis. They lead to radically different results. Also note, as an aside, that here we see a zoologist who does think that flightlessness should be a significant taxonomical criterion — but the official classification disagrees.

Induction and deduction

To design software hierarchies, the proper process is a combination of the deductive and the inductive, of specialization and generalization: sometimes you see the abstraction first and then infer the special cases; sometimes you first build or find a useful class and then realize that there is a more abstract underlying concept.

If you find yourself not always using the first scheme, but once in a while discovering the abstract only after you have seen the concrete, *maybe there is nothing wrong with you*. You are simply using a normal "yoyo" approach to classification.

As you accumulate experience and insight, you should find that the share of (correct) a priori decisions grows. But an a posteriori component will always remain.

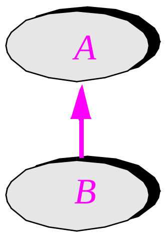
Varieties of class abstraction

This principle of Reversion is the most wonderful of all the attributes of inheritance.

Charles Darwin

Two forms of a posteriori parent construction are common and useful.

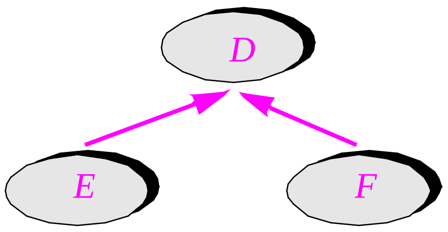
Abstracting is the late recognition of a higher-level concept. You find a class *B* which covers a useful notion, but whose developer did not recognize that it was actually a special case of a more general notion *A*, justifying an inheritance link:



Abstraction

That this insight was initially missed — that is to say, that *B* was built without *A* — is not a reason to renounce the use of inheritance in this case. Once you recognize the need for *A*, you can, and in most cases should, write this class and adapt *B* to become one of its heirs. It is not as good as having written *A* earlier, but better than not writing it at all.

Factoring is the case in which you detect that two classes *E* and *F* actually represent variants of the same general notion:



Factoring

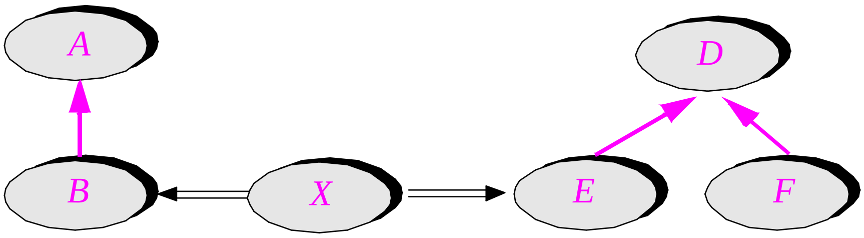
If you recognize this commonality belatedly, the generalization step will enable you to add a common parent class *D*. Here again it would have been preferable to get the hierarchy right the first time around, but late is better than never.

Client independence

Abstracting and factoring may in many cases proceed without negative effects on the existing clients (an application of the Open-Closed principle).

This property results from the method's use of information hiding. Consider again the preceding schematic cases, but with a typical client class *X* added to the picture:

Abstraction, factoring and clients



When *B* gets abstracted into *A*, or the features of *E* get factored with those of *F* into *D*, a class *X* that is a client of *B* or *E* (in the figure it is a client of both) will in many cases not feel any effect from the change. The ancestry of a class does not affect its clients if they are simply applying the features of the class on entities of the corresponding type. In other words, if *X* uses *B* and *E* as suppliers under the scheme

```
b1: B; e1: E
…
b1.some_feature_of_B
…
e1.some_feature_of_E
```

then *X* is unaffected by any re-parenting of *B* or *E* arising from abstracting or factoring.

Elevating the level of abstraction

Abstracting and factoring are typical of the process of continuous improvement that characterizes a successful object-oriented software construction process. In my experience this is one of the most elating aspects of practicing the method: knowing that even though you are not expected to reach perfection the first time around, you are given the opportunity to improve your design continually, until it satisfies everyone.

In a development group that applies the method well, this regular elevation of the *level of abstraction* of the software, and as a corollary of its quality, is clearly perceptible to the project members, and serves as constant incentive and motivation.

24.12 A SUMMARY VIEW: USING INHERITANCE WELL

Inheritance will never cease to surprise us with its power and versatility. In this chapter we have tried to get a better handle at what inheritance really means and how we can use it to our best advantage. A few central conclusions have emerged.

First, we should not be afraid of the variety of ways in which we can use inheritance. Prohibiting multiple inheritance or facility inheritance achieves no other aim than to hurt ourselves. The mechanisms are there to help you: use them well, but use them.

Next, inheritance is for the most part a *supplier*'s technique. It is one weapon in our arsenal of techniques for fighting our adversaries (in particular complexity, the software developer's relentless foe). Inheritance may matter to *client* software as well, especially in the case of libraries, but its main goal is to help us building the thing in the first place.

Of course, all software is designed for its clients, and the clients' needs drive the process. A set of classes is good if it will offer excellent service to client software: interfaces and associated implementations that are complete, free from bad surprises (such as unexpected performance penalties), simple to use, easy to learn, easy to remember, extendible. To achieve these goals, the designer is free to use inheritance and other object-oriented techniques in any way he pleases.The end justifies the means.

Also remember, when designing an inheritance structure, that the goal is software construction, not philosophy. Seldom is there a single solution, or even a best one in the absolute. "Best" means best for the purposes of a certain class of client applications. This is particularly true as we move away from areas such as mathematics and fundamental computing science, where a widely accepted body of theory exists, towards business-driven application domains. To find out what class hierarchy best addresses the notion of company share, you probably need to know whether the software caters to individual investors, to a publicly traded company, to a stock broker, or to the Stock Exchange.

In a way, this is comforting. The naturalist who classifies a certain set of plants and animals must devise absolute categories. In software the equivalent only happens if you are in the business of producing general-purpose libraries (such as those covering fundamental data structures, graphics, databases). Most of the time, your aims will be more modest. You will need to design a *good* hierarchy, one that will satisfy the needs of a certain kind of client software.

The final lesson of this chapter generalizes a comment made in the discussion of facility inheritance: the principal difficulty of building class structures is not inheritance per se; it is the search for abstractions. If you have identified valid abstractions, their inheritance structure will follow. To find the abstractions, the guide you will use is the guide that we follow throughout this book: the theory of abstract data types.

24.13 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Every use of inheritance should reflect some form of “is” relation between two categories of object, either in an external modeled domain or in the software itself.
- Do not use inheritance to model a “has this kind of component” relation; this is the province of the client relation. (Remember *CAR_OWNER*.)
- When inheritance is applicable, client is often potentially applicable too. If the corresponding view can change, use the client relation; if you foresee polymorphic uses, use inheritance.
- Do not introduce intermediate inheritance nodes unless they describe a well-identified abstraction, characterized by specific features.
- A classification of inheritance was defined, based on twelve kinds divided into three general categories: model inheritance (describing relations existing in the modeled domain), software inheritance (describing relations in the software itself), and variation inheritance (for class adaptation in either the model or the software).
- The power of inheritance comes from its combination of a type specialization and a module extension mechanism. It seems neither wise nor useful to use different language mechanisms.
- Implementation and facility inheritance require some care but can be powerful supplier-side techniques.
- View inheritance, a delicate technique involving repeated inheritance, allows classifying object types along several competing criteria. It is useful for professional libraries. In many cases a simpler handle technique is preferable.
- Although not theoretically ideal, the actual process of designing inheritance hierarchies is often yoyo-like — from the abstract to the concrete and back.
- Inheritance is primarily a supplier technique.

24.14 BIBLIOGRAPHICAL NOTES

The principal reference on the taxonomy of inheritance is [Girod 1991]. A book on O-O methodology [Page-Jones 1995], one of a very small number that provide useful methodological advice on object-oriented design, includes precious advice on uses and misuses of inheritance. Another useful reference is [McGregor 1992]; John McGregor has particularly explored the technique called view inheritance in this chapter.

[Breu 1995] also provides interesting concepts, based on a view of proper inheritance usage more restrictive than the one in this chapter.

A technique similar to this chapter’s “handles” is described in [Gil 1994].

The preparation of this chapter benefited from the comments of several biologists who maintain Web-accessible resources on the taxonomy of living beings, in particular: the “tree of life” at the University of Arizona (*phylogeny.arizona.edu/tree/life.html*), courtesy of Professors David Maddison and, for birds, Michel Laurin (the latter from Berkeley). Professor Edwin Everham from Radford University was also very helpful.

General references on the theory of classification, or *systematics*, appear at the end of the next section.

24.15 APPENDIX: A HISTORY OF TAXONOMY

This Appendix is supplementary material, not used in the rest of this book. The study of taxonomic efforts in other disciplines is full of potential lessons for us object-oriented software developers. I hope to spur further interest in this fascinating area — possibly a topic for an inter-disciplinary Master's or Ph. D. thesis.

From Aristotle to Darwin

The classification of species began at least as early as Aristotle (384-322 B.C.E.), whose taxonomy of animals, *Historia Animalium*, continued for plants under the title *Historia Plantarum* by his student Theophrastus of Eresos (ca. 370-288 B.C.E.), was accepted as definitive for many centuries. Aristotle's criteria for classifying animals include both how they reproduce and where they live; from a modern viewpoint, only the first would be considered relevant, as we have come to accept that regardless of habitat considerations a dolphin is closer to a llama than to a shark. Theophrastus's classification was more systematically structural. Modern botanical terminology comes largely from Aristotle and Theophrastus through the Latin translation of the latter's terms in the *Natural History of* Pliny the Elder (23-79 C.E.) (Pliny was well aware of the need to avoid being misled by appearances: "*It was the plan* [of some Greek naturalists] *to delineate the various plants in colors, and then to add in writing a description of the properties which they possessed. Pictures, however, are very apt to mislead*; … *besides, it is not sufficient to delineate a plant as it appears at one period only, as it presents a different appearance at each of the four seasons of the year*.") A later important contributor was Dioscorides of Anazarbus (1st century C.E.), Nero's doctor, who classified plants according to their medicinal properties.

Several scholars took up the work at the time of the Renaissance, in particular Conrad Gessner, who was to influence Linné and Cuvier through his *Opera Botanica* and *Historia Plantarum* (1541-1571), distinguishing genus from species and order from class, and Caspar Bauhin, who devised a binomial system for the classification of plants in his *Pinax* (1596). In the next century, John Ray (1628-1705) removed some of the arbitrariness of prevailing classifications by taking into account several properties of plants' morphology, rather than just one feature. He established the basic division of flowering plants into monocots and dicots (foreseen by Theophrastus). That division, still in use today, is another example of the fuzziness of even some of the fundamental classification criteria of biology; the UC Berkeley Museum of Paleontology (see the bibliographical references at the end of this section) gives a list of seven factors distinguishing monocots from dicots — one *vs*. two cotyledons in the embryo, flower parts in multiples of three *vs*. multiples of four or five, etc. — but adds that no single factor in that list will infallibly identify a given flowering plant as a monocot or dicot.

Only in the eighteenth century, with the development of biology as a science and the fast growth in known species, did the problem of biological classification start to acquire a character of urgency. Whereas Theophrastus had identified five hundred plant species, Bauhin knew six thousand, and Linnaeus catalogued eighteen thousand; less than a century later Cuvier listed over fifty thousand! The philosopher-scientists of the Age of Enlightenment, aroused by Newton's classification of heavenly bodies in his *Principia Mathematica* (1687), were not content any more to list the species, but started to look for meaningful principles of grouping them into categories — for the proper abstraction mechanisms, as we software people would say. The roots of modern taxonomy can be traced to that collective effort of the early modern era.

The key contributor was the Swedish botanist Carl Linné (1701-1778), also known by the Latin name Carolus Linnaeus, who in 1737 published his taxonomic system, still the basis of all taxonomic systems used today. One of his major innovations, was — using software engineering terminology again — to discard the *top-down* approach used by previous taxonomists (who posited basic abstract categories and successively divided them into smaller groups) in favor of a *bottom-up* approach, well in line with the emphasis on pragmatism and experimentation that marked the beginnings of the scientific method; he started from the species themselves and grouped them into categories.

Both Ray and Linné were in search of a "natural system", that is to say an ideal classification that would reveal divine intentions.

Progress between Linné and Darwin was largely due to an astonishing succession of naturalists at the Paris *Jardin des Plantes*:

- Georges-Louis de Buffon (1707-1788) wrote the magnificent 44-volume *Histoire Naturelle*, bold enough to suggest a common ancestry for humans and apes.
- Antoine-Laurent de Jussieu (1748-1836) looked for a more natural and comprehensive system of plant classification than Linné's. Modern taxonomies of plants actually follow from Jussieu's work, itself based on Ray's. (Although modern classification systems are based on Linné's ideas, his actual taxonomy has largely been discarded — initially in part because of moral reasons, since he gave such importance to sexual features.)
- Jean-Baptiste Lamarck (1744-1829), whose theory of evolution announced Darwin's, published his *Flore française* in 1778 and almost single-handedly originated the classification of "invertebrates", a term he coined. In his *Histoire naturelle des Animaux sans Vertèbres* he was the first to separate the crustaceans from the insects.
- Georges Cuvier (1769-1832) did for vertebrates what Lamarck did for invertebrates. He was famous for his ability to reconstruct complete organisms from fossil fragments. He classified animals into four branches.
- Étienne Geoffroy Saint-Hilaire (1772-1844), another great taxonomist, was the adversary of Cuvier (whom he had brought to Paris) in a famous public debate about unity *vs*. diversity of life forms. The dispute reflected deeper questions: evolutionary *vs*. fixed views of species, and the issue, still open today, of formalism *vs*. functionalism. When we see Cuvier writing "*If there are resemblances between the organs of fishes and those of the other vertebrate classes, it is only insofar as there are resemblances between their functions*" in 1828, and Geoffroy responding "*Animals have no habits but those that result from the structure of their organs*" in 1829, it is hard for a software professional to avoid thinking "abstract data type" and "implementation".

The next revolution in taxonomical thought came with Charles Darwin (1809-1882), whose *Origin of Species* (1859) suggested a simple basis for taxonomy: use evolutionary history. The classification of organisms according to their origin in evolution is known as **cladistics**. For some biologists, this is the *only* criterion. The Berkeley Museum of Paleontology again:

> *For many years, since even before Darwin, it has been popular to tell "stories" about how certain traits of organisms came to be. With cladistics, it is possible to determine whether these stories have merit, or whether they should be abandoned in favor of a competing hypothesis. For instance, it was long said that the orb-weaving spiders, with their intricate and orderly webs, had evolved from spiders with cobweb-like webs. The cladistic analysis of these spiders showed that, in fact, orb-weaving was the primitive state, and that cobweb-weaving had evolved from spiders with more orderly webs.*

Biologists who use to this single, unimpeachable criterion, are in a way more fortunate than us poor software modelers: they can assume, or pretend, that there is a single taxonomical truth, and that the only problem is to reconstruct it. (In other words they have fulfilled Ray's, Linné's and Jussieu's quest for a single Natural System.) In software modeling we cannot postulate, let alone discover, such an underlying truth.

The modern scene

You would think that biological taxonomy, with its long and prestigious history, from Aristotle to Darwin and Huxley, would by now be a sedate field. Think again. Since the sixties, controversy has been raging. There are three main schools, the ardor of whose debates will seem thoroughly familiar to anyone who has heard software engineers debate their favorite programming languages. Here is — after the taxonomy of taxonomy which occupied our efforts at the beginning of this chapter — the taxonomy of taxonomists:

- The *numerical pheneticists* draw their classifications from the study of organisms' individual characters, using numerical measures of distance (and relying generously on computer algorithms) to group organisms that have the most characters in common. Sokal and Sneath are recognized as the founders of this approach.
- The *cladists* use evolutionary history as the sole criterion. The Berkeley extract reflected this view (more details below). Cladistics draws its inspiration from work by the German scientist Willi Hennig, first published in German in 1950 and in English in 1965.
- The *evolutionary taxonomists*, led by G.G. Simpson and Ernst Mayr, who claim Darwin's direct heritage, "*base* [their] *classifications on observed similarities and differences among groups of organisms, evaluated in the light of their inferred evolutionary history*" (as stated by Mayr, 1981, reference below).

It is next to impossible to find neutral accounts of the arguments for each approach in the literature. (Perhaps this sounds familiar.) It falls on the outsider to try to develop an impartial view. In this brief survey we will try to remain as close as possible to the software analogies.

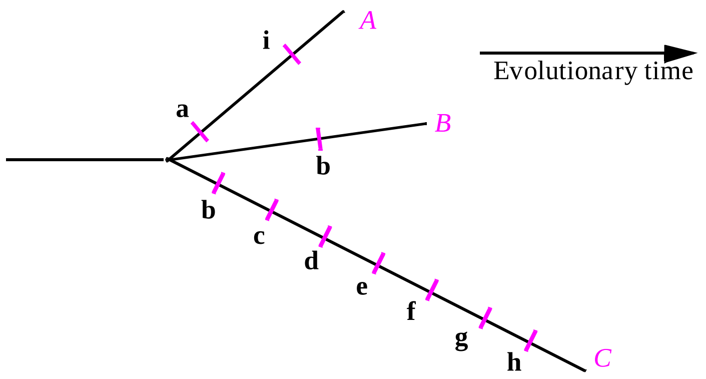
Numerical phenetics — what we would call the bottom-up approach — has the advantage of being based on precise, repeatable measures. But the choice of measured characters and their weighting is subjective. And a purely external measure risks being influenced by chance factors; it is well known since Darwin that evolution involves not only divergence (species evolving from a common ancestor by developing different characters) but convergence (completely distinct species developing similar features to adapt to similar environments or by sheer coincidence). So there is a great danger of arbitrariness. One can also fear instability: the discovery of new species — which occurs all the time in biology — could, more than with the other approaches, put into question classifications drawn from the statistical analysis of the previously known species.

On the surface the other two schools would seem to be very close to each other. Why then do they keep arguing with each other from their respective journals and conferences? The reason is that the cladists are particularly rigorous, as they would see it, or dogmatic, as the other two schools might put it. They take evolution, and evolution only, as the classification criterion. The method is particularly strict: it examines the evolutionary history, as given by the fossil record, and decides which characters are *synapomorhic* and which ones *plesiomorphic*. A feature is plesiomorphic if it was already present in a common ancestor; then for the cladist it is not interesting at all! The useful features as the synapomorphic ones, which hold for two organisms but not their ancestors. Synapomorphies are the primary tool for positing new groups (*taxa*, the plural of *taxon*).

In the following situation, then, the cladists will see only two taxa:

A cladogram

After Mayr, 1961.



This is a *cladogram*, or record of the appearance of characters in the evolutionary history. The marks indicate new characters. *B* and *C* have a synapomorphy, character **b**, which was not in the ancestor and is not in *A*; so for a cladist *B* and *C* will form a taxon, and *A* another. For an evolutionary taxonomist, there would be three taxa, since *C* differs from *B* in many other characters (**c** to **h**). In its pure form cladistics is even more restrictive: like Roman Jakobson's phonology, it only considers *binary* characters; and it posits that when taxa evolve from a common ancestor the ancestor disappears.

Evolutionary taxonomy seems a more moderate approach, trying to draw from both cladistics and phenetics: evolution is the classification basis, but complemented by analysis of other characters, not necessarily synapomorphic.

Why then the restrictiveness of cladistics? The principal argument is epistemological: an attempt to satisfy Karl Popper's rules of falsifiability. Cladists argue that their approach is the only non-circular one; whereas the other two more or less assume (according to this view) what they are trying to deduce, a cladistic hypothesis can be refuted, in the same way that a single experiment can disprove a theory of physics, although no amount of experimentation will *prove* a theory.

The debate between these approaches is not closed. The progress of molecular biology will certainly affect it; in particular, by providing a link between observed characters and the evolutionary record, it may help achieve some reconciliation between phenetics and the other two methods.

We will stop here, with regret (more mundane software engineering topics are claiming our attention). For an O-O software developer, reading the taxonomy literature, although requiring a fair deal of attention in some cases ("*A phylogenetic definition of homology may be considered more falsifiable than a phenetic definition and therefore preferable if it leads to a hypothesis of homology which includes all the potential falsifiers provided by phenetic comparisons as well as the potential falsifiers provided by phylogeny…*") is rich in rewards. Our own work constantly subjects us, like our friends from the Biology department or the Herbarium, to two siren songs from opposite sides: the a priori form of classification, top-down, deductive and based on a "natural" order of things, coming to us through the cladists from Linné; and the empirical, inductive, bottom-up view of the pheneticists, telling us to observe and gather. Perhaps, like the evolutionary taxonomists, we will want a bit of both.

Bibliography on taxonomy

The following references — which have been separated from the main bibliography of this book to avoid too much *mélange des genres* — will be useful as a starting point on the subject of taxonomy history:

- The on-line material on evolution at the University of California Museum of Paleontology in Berkeley: **http://www.ucmp.berkeley.edu/clad/clad4.html** (authors: Allen G. Collins, Robert Guralnick, Brian R. Speer). Resolutely cladist. Some of the above presentation draws from the UCMP pages and from suggestions by their authors.

- A biography of Jussieu: *Antoine-Laurent de Jussieu, Nature and the Natural System* by Peter F. Stevens, Columbia University Press, New York, 1994. (I am grateful to Prof. Stevens for several important suggestions.)

- A collection of papers on cladistics: *Cladistic Theory and Methodology*, edited by Thomas Duncan and Tod F. Stuessy, Van Nostrand Reinhold, 1985. Quite cladist, but the end of the volume adds some interesting critical articles, one in particular by Ernst Mayr (*Cladistic analysis or cladistic classifications*?, pages 304-308, originally in *Zeitung Zool. Syst. Evolut.-Forsch.*, 19:94-128, 1974).

- Another volume of contributions: *Prospects in Systematics*, ed. D.L. Hawksworth, Systematics Association, Clarendon Press, Oxford, 1988.

- A textbook: *Biological Systematics* by Herbert H. Ross, Addison-Wesley, Reading (Mass.), 1973.

- The founding book of cladistics: *Phylogenetic Systematics* by Willi Hennig, English translation, University of Illinois Press, Urbana (Ill.), 1966. See also a shorter presentation by Hennig (adapted from his original 1950 article) in Duncan and Stuessy.

- A cladistic treatise, starting with the picture of Hennig: *Phylogenetics — The Theory and Practice of Phylogenetic Systematics* by E.O. Wiley, published by John Wiley and Sons, New York, 1981. By the same author, a Popperian argument for cladistics, *Karl R. Popper, Systematics, and Classification: A Reply to Walter Bock and Other Evolutionary Taxonomists*, pages 37-47 of Duncan and Stuessy, originally in *Syst. Zool* 24:233-243, 1975.

- A clear article by Ernst Mayr, leaning to evolutionary taxonomy but discussing the other approaches with some sympathy: *Biological Classification: Towards a Synthesis of Opposing Methodologies*, in *Science*, vol. 214, 1961, pages 510-516.

- The foundational text of the pheneticists: *Principles of Numerical Taxonomy*, by Robert P. Sokal and Peter H.A. Sneath, Freeman Publishing, San Francisco, 1963, revised edition 1973.

- A short and more recent book advocating *Transformed Cladistics* (subtitle: *Taxonomy and Evolution*) by N.R. Scott-Ram, Cambridge University Press, 1990.

EXERCISES

E24.1 Arrayed stacks

Write in full the *STACK* class and its heir *ARRAYED_STACK* sketched in this chapter, using the "marriage of convenience" technique.

E24.2 Meta-taxonomy

See "General taxonomy", page 824.

Imagine this chapter's classification of the forms of inheritance were an inheritance hierarchy. What kind or kinds would it involve?

E24.3 The stacks of Hanoï

The Towers of Hanoï problem, used in many computing science texts as an example of recursive procedure, comes from Édouard Lucas, "Récréations Mathématiques", Paris, 1883, reprinted by Albert Blanchard, Paris, 1975.

(This exercise comes from an example used by Philippe Drix on the French GUE electronic mailing list, late 1995 and early 1996.)

Assume a deferred class *STACK* with a procedure *put* to push an element onto the top, with a precondition involving the boolean-valued function *full* (which could also be called *extendible*; as you study the exercise you will note that the choice of name may affect the appeal of various possible solutions).

Now consider the famous problem of the Towers of Hanoï, where disks are stacked on piles — the towers — with the rule that a disk may only be put on a larger disk.

Is it appropriate to define the class *HANOÏ_STACK*, representing such piles, as an heir to *STACK*? If so, how should the class be written? If not, can *HANOÏ_STACK* still make use of *STACK*? Write the class in full for the various possible solutions; discuss the pros and cons of each, state which one you prefer, and explain the rationale for your choice.

E24.4 Are polygons lists?

"POLYGONS AND RECTANGLES", 14.1, page 460.

The implementation of our first inheritance example, class *POLYGON*, uses a linked list attribute *vertices* to represent the vertices of a polygon. Should *POLYGON* instead inherit from *LINKED_LIST* [*POINT*]?

E24.5 Functional variation inheritance

Provide one or more examples of functional variation inheritance. For each of them, discuss whether they are legitimate applications of the Open-Closed principle or examples of what the discussion called "organized hacking".

E24.6 Classification examples

For each of the following cases, indicate which one of the inheritance kinds applies:

- *SEGMENT* from *OPEN_FIGURE*.
- *COMPARABLE* (objects equipped with a total order relation) inheriting from *PART_COMPARABLE* (objects with a partial order relation).
- Some class from *EXCEPTIONS*.

"ADVANCED EXCEPTION HANDLING", 12.6, page

E24.7 Where do iterators belong?

Would it be a good idea to have iterator features (*while_do* and the like) included in classes describing the data structures on which they iterate, such as *LIST*? Consider the following points:

- The ease of applying iterations to arbitrary *action* and *test* routines, chosen by the application.
- Extendibility: the possibility of adding new iteration schemes to the library.
- More generally, respect of object-oriented principles, in particular the idea that operations do not exist by themselves but only in relation to certain data abstractions.

E24.8 Module and type inheritance

Assume we devise a language with two kinds of inheritance: module extension and subtyping. Where would each of the inheritance kinds identified in this chapter fit?

E24.9 Inheritance and polymorphism

Of the kinds of inheritance reviewed in this chapter between a parent *A* and an heir *B*, which ones do you expect in practice to be used for polymorphic attachment, that is to say assignments *x* := *y* or the corresponding argument passing with *x* of type *A* and *y* of type *B*?