17

Typing

*E*ffective use of object technology requires that we clearly specify, in the texts of our systems, the types of all objects that they will manipulate at run time. This rule, known as static typing — a notion defined precisely in the next sections — makes our software:

- More *reliable*, by enabling compilers and other tools to suppress discrepancies before they have had time to cause damage.
- More *readable*, by providing precious information to authors of client systems, future maintainers of our own software, and other readers.
- More *efficient*, since this information helps a good compiler generate better code.

Although the typing issue has been extensively discussed in non-O-O contexts, and static typing applied to many non-O-O languages, the concepts are particularly clear and relevant in object technology since the approach as a whole is largely based on the idea of type, merged with the idea of module to yield the basic O-O construct, the class.

The desire to provide static typing has been a major influence on the mechanisms discussed in earlier chapters. Here we need to take a comprehensive look at typing and devise solutions to the remaining difficulties raised by this concept.

17.1 THE TYPING PROBLEM

One nice thing can be said about the typing issue in object-oriented software construction: it may not be an easy problem, but it is a *simple* problem — simple, that is, to state.

The Basic Construct

The problem's simplicity comes from the simplicity of the object-oriented model of computation. If we put aside some of the details, only one kind of event ever occurs during the execution of an object-oriented system: feature call, of the general form

x.*f* (*arg*)

which executes on the object attached to *x* the operation *f*, using the argument *arg*, with the understanding that in some cases *arg* stands for several arguments, or no argument at all. Smalltalk programmers would say "pass to the object *x* the message *f* with argument *arg*", and use another syntax, but those are differences of style, not substance.

That everything relies on this Basic Construct accounts in part for the general feeling of beauty that object-oriented ideas arouse in many people.

From the Basic Construct follows the basic kind of abnormal event that might occur at execution time:

Definition: type violation

A run-time type violation (or just type violation for short) occurs in the execution of a call $x.f(arg)$, where x is attached to an object OBJ, if either:

V1 • There is no feature corresponding to f and applicable to OBJ.

V2 • There is such a feature, but *arg* is not an acceptable argument for it.

The typing problem is the need to avoid such events:

Object-oriented typing problem

When do we know whether the execution of an object-oriented system may produce a type violation?

The key word is *when*. If the feature or arguments do not match, you will find out sooner or later: applying the feature "raise salary" to an instance of *SUBMARINE* or "fire the torpedoes" to an instance of *EMPLOYEE* will not work; somehow the execution will fail. But you may prefer to find out sooner rather than later.

Static and dynamic typing

Although intermediate variants are possible, two main approaches present themselves:

- *Dynamic typing*: wait until the last possible moment, the execution of each call.
- *Static typing*: rely on a set of rules that determine, from the text of a system, whether its executions may cause type violations. Only execute systems for which the rules guarantee that no violation will ever occur.

The names are easy to explain: with dynamic typing, type verification occurs at execution time (dynamically); with static typing, it is performed on the text of the software (statically, that is to say before any execution).

The terms "typed" and "untyped" are sometimes used for "statically typed" and "dynamically typed". To avoid any confusion we will stick to the full names.

Static typing is only interesting if the rules can be checked automatically. Since software texts are usually processed by a compiler before being executed, it is convenient to have the compiler, rather than a separate tool, take care of these checks. The rest of the discussion will indeed assume for simplicity that the compiler and the type checker are the same tool. This assumption yields a simple definition:

Definition: statically typed language

An object-oriented language is statically typed if it is equipped with a set of consistency rules, enforceable by compilers, whose observance by a system text guarantees that no execution of the system can cause a type violation.

In the literature you will encounter the term "*strong* typing". It corresponds to the all-or-nothing nature of this definition, which demands rules that guarantee the absence of type violations. *Weak* forms of static typing, whose rules eliminate certain type violations but not all, are also possible, and some O-O languages are indeed weakly-statically-typed in this sense. We shall strive, however, for the strongest possible form.

Some authors also talk about strong forms of dynamic typing. But this is a contradiction.

In a dynamically typed language (also known as an "untyped" language), there are no type declarations; entities simply become associated with whatever values the execution of the software attaches to them. No static type checking is possible.

Typing rules

Our object-oriented notation is statically typed. Its type rules have been introduced in earlier chapters; they boil down to three simple constraints:

- Every entity or function must be declared as being of a certain type, as in *acc*: *ACCOUNT*; every routine declares zero or more formal arguments, with a type for each, as in *put* (*x*: *G*; *i*: *INTEGER*).
- In any assignment $x := y$, and in any routine call using y as the actual argument for the formal argument x, the type of the source y must conform to the type of the target x. The definition of conformance is based on inheritance — B conforms to A if it is a descendant of A — complemented by rules for generic parameters.

Type Conformance rule, page 474.

- In a call of the form $x.f\ (arg)$, f must be a feature of the base class of x's type, and must be available to the class in which the call appears.

Feature Call rule, page 473.

Realism

Although the definition of "statically typed language" is precise, it also highlights the need for informal criteria in devising type rules. Consider the following two extreme cases:

- An *all-valid language* in which every syntactically correct system is also typewise-valid, with no need for type rules. Such languages are possible (imagine for example a small notation for Polish-style additions and subtractions with integers); unfortunately, as readers familiar with the theory of computation will know, no useful general-purpose language can meet that criterion.
- An *all-invalid language*, easy to devise: just take any existing language and add a type rule that makes *any* system invalid! This makes the language typed according to the definition: since no system passes the rules, no system that passes the rules can cause a type violation.

We may say that an all-valid language is **usable**, but not **useful** for general-purpose development; an all-invalid language may be useful, but it is not usable.

What we need in practice is a type system that makes the language both useful and usable: powerful enough to express the computations we need; convenient enough not to force us into undue complications to satisfy the type rules.

We will say that a language is **realistic** if it is both useful and usable. Unlike the definition of static typing, which always yields an indisputable answer to the question "*Is language X statically typed*?", the definition of realism is partly subjective; reasonable people may disagree on whether a language, equipped with certain type rules, is still useful and usable.

In this chapter we will check that the typed notation defined in the preceding chapters is realistic.

Pessimism

In discussing approaches to O-O typing we should keep in mind another general property of static typing: it is always, by nature, a pessimistic policy. Trying to guarantee that *no computation shall ever fail*, you disallow *some computations that might succeed*.

To see this, consider a trivial non-O-O language, Pascal-like, with distinct types *INTEGER* and *REAL*. With the declaration *n*: *INTEGER*, the assignment *n* := *r* will be rejected as violating the type rules. So all the following will be considered type-invalid and rejected by the compiler:

n := *0.0* [A]
n := *1.0* [B]
n := *—3.67* [C]
n := *3.67 — 3.67* [D]

Of these invalid operations, [A], if permitted to execute, would always work since any number system will provide an exact representation for the floating-point number 0.0, which can be transformed unambiguously to the integer 0. [B] would almost certainly work too. [C] is ambiguous (do we want the rounded version, the truncated version of the number?) But [D] would work. So would

if *n* ^ *2* < *0* **then** *n* := *3.67* **end** [E]

because the assignment will never be executed (*n* ^ *2* denotes the square of *n*). If we replace *n* ^ *2* by just *n*, where *n* is read from user input just before the test, some executions would work (those for which *n* is non-negative), others would not. Assigning to *n* a very large real number, not representable as an integer, would not work.

In a typed language, all these examples — those which would always work, those which would never work, and those which would work some of the time — are equally and mercilessly considered violations of the type rules, and any compiler will reject them.

The question then is not *whether* to be pessimistic but *how* pessimistic we can afford to be. We are back to the realism requirement: if the type rules are so pessimistic as to bar us from expressing in a simple way the computations that we need, we will reject them. But if they achieve type safety with little loss of expressive power, we will accept them and enjoy the benefits. For example making *n* := *r* invalid turns out to be good news if the environment provides functions such as *round* and *truncate*, enabling you to convert a real into an integer in exactly the way you want, without the ambiguity of an implicit conversion.

17.2 STATIC TYPING: WHY AND HOW

Although the advantages of static typing seem obvious, it is necessary to review the terms of the debate.

The benefits

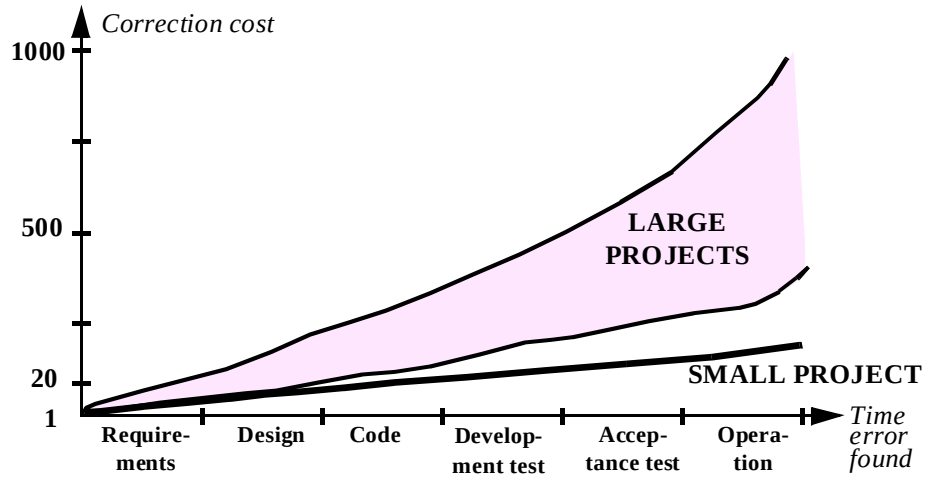
The reasons for using a statically typed form of object technology were listed at the very beginning of this chapter: reliability, readability and efficiency.

The **reliability** value comes from the use of static typing to detect errors that would otherwise manifest themselves only at run time, and only in certain runs. The rule that forces you to declare entities and functions — the first of our three type rules above — introduces redundancy into the software text; this enables the compiler, through the other two rules, to detect inconsistencies between the purpose and actual use of an entity, feature or expression.

Catching errors early is essential, as correction cost grows quickly with the detection delay. This property, intuitively clear to all software professionals, is confirmed quantitatively, for specification errors, by Boehm's well-known studies, plotting the cost of correcting an error against the time at which it is found (base 1 if found at requirements time), for both a set of large industrial projects and a controlled small project experiment:

Relative cost of correcting errors

After [Boehm 1981]. Reproduced with permission.



The **readability** benefit is also appreciable. As the examples appearing throughout this book should show convincingly, declaring every entity and function with a certain type is a powerful way of conveying to the software reader some information about its intended uses. This is particularly precious for maintainers of the software.

> If readability were not part of the goal we might be able to obtain some of the other benefits of typing without explicit declarations. It is possible indeed, under certain conditions, to use an implicit form of typing in which the compiler, instead of requiring software authors to declare entity types, attempts to determine the type of each entity automatically from its uses. This is known as *type inference*. But from a software engineering perspective explicit declarations are a help, not a penalty; types should be clear not just to the compiler but to the human reader.

Finally, the **efficiency** benefit can make the difference between success and failure of object technology in practice. Without static typing, the execution of *x.f (arg)* can take an arbitrary long time: as we saw in the discussion of inheritance, the basic algorithm looks for a feature *f* in the base class *C* of *x*'s type; if it does not find it, it looks in *C*'s parents, and so on. This is a fatal source of inefficiency. It can be mitigated by improvements to the basic algorithm, and the authors of the Self language have done extensive work to enable better code generation for a dynamically typed language. But it is through static typing that O-O software has been able to approach or equal the efficiency of traditional software.

For more details on the implementation techniques discussed in this section see "Dynamic binding and efficiency", page 508. On Self, see the bibliographical notes.

The key idea was explained in the earlier discussion. When the compiler generates the code for *x.f (arg)*, it knows the type of *x*. Because of polymorphism, this is not necessarily the type of the attached run-time object OBJ, and so does not uniquely determine the proper version of *f*. But the declaration restricts the set of possible types, enabling the compiler to generate tables providing run-time access to the right *f* at minimum — and **constant-bounded** — expense. Further optimizations of *static binding* and *inlining*, also facilitated by typing, eliminate the expense altogether in applicable cases.

Arguments for dynamic typing

In spite of these benefits of static typing, dynamic typing keeps its supporters, found in particular in the Smalltalk community. Their argument mainly follows from the realism issue cited above: they contend that static typing is too constraining, preventing the unfettered expression of software ideas. Terms such as "stranglehold" and "chastity belt" are often heard in such discussions.

This argument can be correct, but only for a statically typed language that misses some important facilities. It is indeed remarkable that all the type-related concepts introduced in preceding chapters are necessary; remove any of them, and the straitjacket comment becomes valid in at least some cases. But by including them all we obtain enough flexibility to make static typing both practical and pleasurable.

The ingredients of successful typing

Let us review the mechanisms which permit realistic static typing. They have all been introduced in earlier chapters, so that we only need a brief reminder for each; listing them all together shows the consistency and power of their combination.

Our type system is entirely based on the notion of **class**. Even basic types such as *INTEGER* are defined by classes. So we do not need special rules for predefined types. (Here the notation departs from "hybrid" languages such as Object Pascal, Java and C++, which retain the type system of an older language along with the class-based system of object technology.)

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

Expanded types give us more flexibility by allowing types whose values denote objects along with types whose values denote object references.

Crucial flexibility is afforded by **inheritance** and the associated notion of **conformance**. This addresses the major limitation of traditional typed languages such as

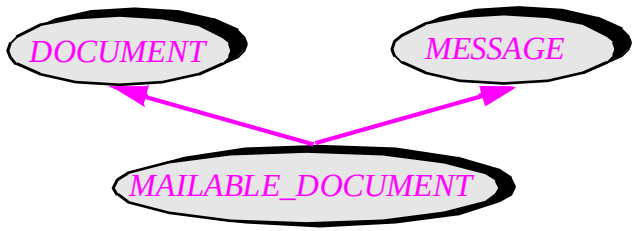
"Limits to polymorphism", page 474.

Pascal and Ada, where an assignment *x* := *y* requires the types of *x* and *y* to be identical. This rule is too strict: it prevents you from using an entity that may denote objects of various related types, such as a *SAVINGS_ACCOUNT* and a *CHECKING_ACCOUNT*. With inheritance, all we require is that the type of *y* conform to the type of *x*; this is the case if *x* is of type *ACCOUNT*, *y* of type *SAVINGS_ACCOUNT*, and the latter class is a descendant of the former.

Chapter 15.

To be practical, a statically typed language requires its inheritance scheme to support **multiple inheritance**. A principal part of common objections against static typing is that it prevents you from looking at objects in different ways. For example an object of type *DOCUMENT* might need to be transmitted over a network, and so will need the features associated with objects of type *MESSAGE*. But this is only a problem with a language that is restricted to single inheritance; with multiple inheritance you can introduce as many viewpoints as you need.

Multiple inheritance



Chapter 10.

We also need **genericity**, to define flexible yet type-safe container data structures. For example a list class will be defined as **class** *LIST* [*G*] … Without this mechanism, static typing would force us to declare a different class for each type of list element — an obviously unsustainable solution.

"CONSTRAINED GENERICITY", 16.4, page 585.

Genericity needs in some cases to be **constrained**, allowing us to apply certain operations to entities of a generic type. For example if a generic class *SORTABLE_LIST* has a sort operation, it requires a comparison operation on entities of type *G*, the generic parameter. This is achieved by associating with *G* a generic constraint *COMPARABLE*:

class *SORTABLE_LIST* [*G* –> *COMPARABLE*] …

meaning that any actual generic parameter used for *SORTABLE_LIST* must be a descendant of class *COMPARABLE*, which has the required comparison features.

"ASSIGNMENT ATTEMPT", 16.5, page 591.

Another indispensable mechanism is **assignment attempt**, to access objects whose type the software does not control. If *y* denotes an object obtained from a database or a network, you cannot be sure it has the expected type; the assignment attempt *x* ?= *y* will assign to *x* the value of *y* if it is of a compatible type, but otherwise will make *x* void. Without assignment attempt we could not abide by the type rules in such cases.

Chapter 11.

Assertions — associated, as part of the idea of Design by Contract, with classes and features in the form of preconditions, postconditions and class invariants — allow you to describe semantic constraints which cannot be captured by type specifications. Although with the "interval types" of such languages as Pascal and Ada you can declare, for

example, that a certain entity takes its values between 10 and 20, no type mechanism will enable you to state that *i* must be either in that interval or negative, and always twice as much as *j*. Here class invariants come to the rescue, by letting you specify exactly what you need, however sophisticated the constraint.

Anchored declarations are essential in practice to avoid redeclaration avalanche. By declaring *y*: **like** *x* you make sure that *y* will follow any redeclaration of the type of *x* in a descendant. Without this mechanism developers would be endlessly redeclaring routines for type purposes only.

"ANCHORED DECLARATION", 16.7, page 599.

Anchored declarations are a specific case of our last required language mechanism: **covariance**, which will be discussed in more detail later in this chapter.

A practical property of the environment is also essential: **fast incremental recompilation**. When you write a system or (more commonly) modify an existing system, you will want to see the effect soon. With static typing you must first let the compiler re-typecheck the system. Traditional compiling techniques require recompiling the whole system (and going through a *linking* process); the time may be painfully long, especially for a proportionally small change to a large system. This phenomenon has been a major *a contrario* argument for **interpreted** approaches, such as those of early Lisp and Smalltalk environments, which execute systems with no or little processing, hence no type checking. But modern compiler technology removes this argument. A good compiler will detect what has changed since the last compilation, and reprocess only that part, keeping the recompilation time small — and proportional to the size of the change, not of the system.

> The *Melting Ice Technology* described in the last chapter of this book achieves this goal, typically permitting recompilation in a matter of seconds after a small change even to a large system.

"A little bit typed"?

It was noted above that we should aim for a *strong* form of static typing. This means that we should avoid any loopholes in the static requirements — or, if any such loopholes remain, identify them clearly, if possible providing tools to flag any software using them.

The most common loophole, in languages that are otherwise statically typed, is the presence of conversions that disguise the type of an entity. In C and its derivatives, conversions are called "casts" and follow a simple syntax: *(OTHER_TYPE) x* denotes the value of *x* presented to the compiler as if it were of type *OTHER_TYPE*; there are few limitations on what that type may be, regardless of *x*'s actual type.

Such mechanisms evade the constraints of type checking; casting is indeed a pervasive feature of C programming, including in the ANSI C variant (which is "more" typed than its precursor, the so-called Kernighan and Ritchie version). Even in C++, examination of published software shows that casts, although less frequent, remain an accepted and possibly indispensable occasional practice.

It seems difficult to accept claims of static typing if at any stage the developer can eschew the type rules through casts. Accordingly, the rest of this chapter will assume that the type system is strict and allows no casts.

> You may have noted that assignment attempts, mentioned above as an essential component of a realistic type system, superficially resemble casts. But there is a fundamental difference: an assignment attempt does not blindly force a different type; it *tries* a candidate type, and enables the software to check whether the object actually matches that type. This is safe, and indispensable in some circumstances. The C++ literature sometimes includes assignment attempts ("downcasts") in its definition of casts; clearly, the above prohibition of casts only covers the harmful variant, and does not extend to assignment attempts.

Typing and binding: avoiding the confusion

Although as a reader of this book you will have no difficulty distinguishing static typing from static *binding*, you may meet people who confuse the two notions. This may be due in part to the influence of Smalltalk, whose advocacy of a dynamic approach to both typing and binding may leave the inattentive observer with the incorrect impression that the answer to both questions must be the same. (The analysis developed in this book suggests that to achieve reliability and flexibility it is preferable to combine dynamic binding with static typing.) Let us carefully compare the two concepts.

Both have to do with the semantics of the Basic Construct $x.f(arg)$; they cover the two separate questions that it raises:

Typing and binding

- **Typing question**: When do we know for sure that at run time there will be an operation corresponding to f and applicable to the object attached to x (with the argument arg)?
- **Binding question**: Which operation will the call execute?

Typing addresses the existence of **at least one** operation; binding addresses the choice of **the right one** among these operations, if there is more than one candidate.

In object technology:

- The typing question follows from *polymorphism*: since x may denote run-time objects of several possible types, we must make sure that an operation representing f is available in all cases.
- The binding question follows from *redeclaration*: since a class can change an inherited feature — as with *RECTANGLE* redefining *perimeter* inherited from *POLYGON* — there may be two or more operations all vying to be the one representing f for a particular call.

Both answers can be dynamic, meaning at execution time, or static, meaning before execution. All four possibilities appear in actual languages:

- Some non-O-O languages, such as Pascal and Ada, have both static typing and static binding. In these languages each entity represents objects of only one type, specified statically; the approach yields reliability at the expense of flexibility.
- Smalltalk and other O-O languages influenced by it have dynamic binding and dynamic typing. This is the reverse choice: favoring flexibility at the expense of reliability enforcement.
- Some non-O-O languages are untyped (really meaning, as we have seen, dynamically typed) and statically bound. They include assembly languages and some scripting languages.
- The notation developed in this book supports static typing and dynamic binding.

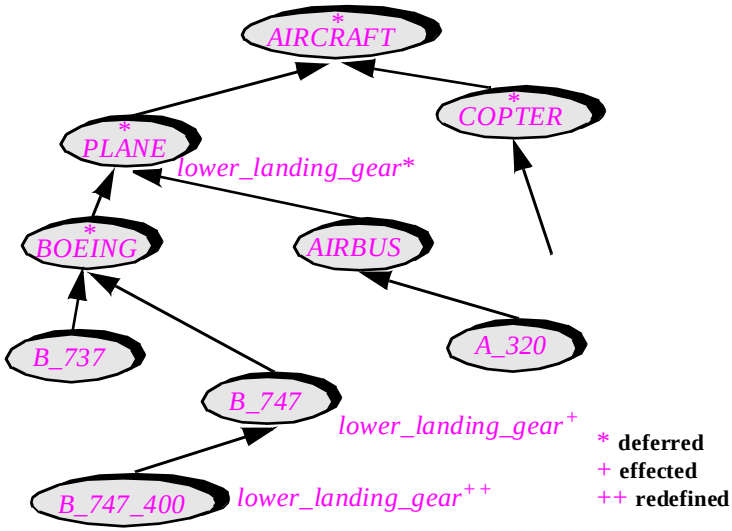
Note the peculiarity of C++ which supports static typing (although in a non-strong form because of the presence of casts) and, for binding, a static policy by default, while permitting dynamic binding at the price of explicit **virtual** declarations.

The C++ policy was discussed in "The C++ approach to binding", page 514.

The reason choosing static typing and dynamic binding is clear. To the first question, "when do we know we have a feature?", the most attractive answer for reliable software engineering is the static one: "*at the earliest possible time*" — compilation time, to catch errors before they catch you. To the second question, "what feature do we use?", the most attractive answer is the dynamic one: "*the right feature*" — the feature directly adapted to the object's type. As discussed in detail in the presentation of inheritance, this is the only acceptable solution unless static and dynamic binding have the same effect.

The following fictitious inheritance hierarchy helps make these notions more vivid.

Kinds of flying object



For a call of the form

my_aircraft.*lower_landing_gear*

the typing question is when to ascertain that there will be a feature *lower_landing_gear* applicable to the object (for a *COPTER* there would not be any); the binding question is which version to choose (since we have several versions, as shown).

Static binding would mean that we disregard the object type and believe the entity declaration, leading us for example to apply to a Boeing 747-400 the version of a feature, such as *lower_landing_gear*, that has been defined for the standard Boeing 747 planes, instead of the version specially redefined for the 747-400 variant; this is clearly wrong if the object is of the latter type. Dynamic binding will apply the operation that the object demands, based on its type; this is the right approach.

With static typing we will refuse the call at compile time unless we can guarantee that whatever happens to *my_aircraft* at run time the type of the attached object will be equipped with a feature corresponding to *lower_landing_gear*. The basic technique for obtaining this guarantee is simple: since we must declare *my_aircraft*, we require that its type's base class include such a feature. This means that the declared type cannot be *AIRCRAFT* since there is no *lower_landing_gear* at that level; helicopters, for example, have no landing gears, for the purpose of this example at least. With such a declaration the compiler would reject our software with no possibility of appeal. But if we declare the entity as being of type *PLANE*, which has the required feature, all is well.

Smalltalk-style ynamic typing would mean waiting until execution to find out if there is an applicable feature; acceptable perhaps for prototypes and experimental software, but not for production systems. Run time is a little late to ask whether you have a landing gear.

17.3 COVARIANCE AND DESCENDANT HIDING

In a simple world a discussion of typing would stop here: we have defined the goals and advantages of static typing; examined the constraints that a realistic type system must meet; and reviewed the typing techniques of the object-oriented framework developed in the preceding chapters, checking that they satisfy the stated criteria.

The world is not simple. The combination of static typing with some of the software engineering requirements of object technology makes the issues more difficult than they appear at first. Two techniques raise difficulties: covariance, the change of argument types in redefinitions; and descendant hiding, the ability for a class to restrict the export status of an inherited feature.

Covariance

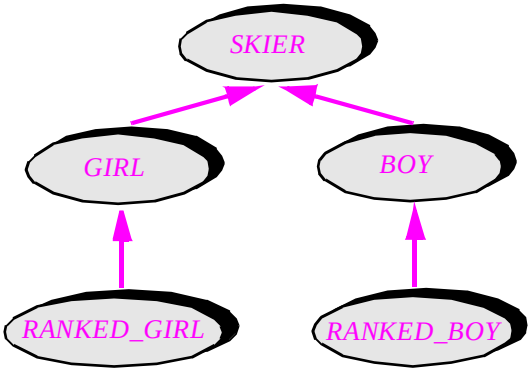
See the original discussions in "TYPING AND REDECLARATION", 16.6, page 596 and "ANCHORED DECLARATION", 16.7, page 599.

The principal problem is what happens to arguments when we redefine a feature's type. We have encountered several cases already: devices and printers, linkable and bi-linkable elements, points and their conjugates.

To understand the general nature of the issue let us use a fresh example. Being non-technical, it carries the usual risks of metaphors; but the closeness to software schemes is obvious, and we will frequently come back to actual software examples.

The example involves a high-school ski team preparing for a trip to a minor-league championship, and the team members' concerned parents. For brevity and simplicity it uses the class names *GIRL* as an abbreviation for "member of the girls' ski team" and *BOY* as an abbreviation for "member of the boys' ski team". Some skiers on each team are ranked, that is to say have already recorded good results in earlier championships. This is an important notion: ranked skiers will start first in a slalom, thus gaining a considerable advantage over the others since a slalom run is much harder to negotiate after too many competitors have already worked it. (This rule that ranked skiers go first is a way to privilege the already privileged, and may be the reason why skiing exerts such a fascination over many people: that it serves as an apt metaphor for life itself.) We get two new classes, *RANKED_GIRL* and *RANKED_BOY*.

Kinds of skier



Some rooms are reserved for boys only, girls only, ranked girls only; we may use a class hierarchy parallel to the one above: *ROOM*, *GIRL_ROOM*, *RANKED_GIRL_ROOM* etc. The discussion will omit *RANKED_BOY* which is parallel to *RANKED_GIRL*.

Here is an outline of class *SKIER*:

```
class SKIER feature
        roommate: SKIER
                -- This skier's roommate
        share (other: SKIER) is
                -- Choose other as roommate.
            require
                other /= Void
            do
                roommate := other
            end
        … Other possible features omitted in this class and the following ones …
end -- class SKIER
```

We have two features of interest: the attribute *roommate*; and the procedure *share*, which assigns a certain skier as roommate to the current skier, as in

```
s1, s2: SKIER
…
s1.share (s2)
```

Rather than *SKIER*, you may have thought of using for *other* the anchored type **like** *roommate* (or **like** *Current* for both *roommate* and *other*). If so, you are most likely right, but let us forget for a while that we know about anchored types: this will enable us to understand the covariance problem in its bare form; anchored types will soon come back.

How does type redefinition get into the picture? Assume the rules require girls to share rooms only with girls, and ranked girls only with other ranked girls. We will redefine the type of feature *roommate*, as shown below (in this class text and the next, the redefined elements appear underlined).

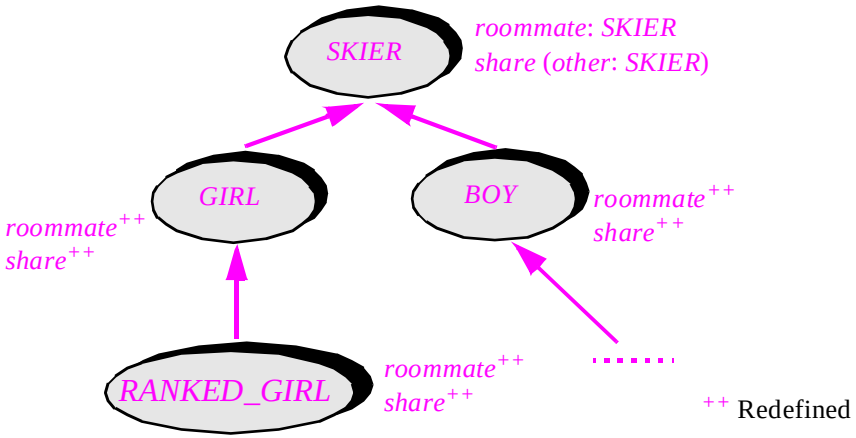
```
class GIRL inherit
        SKIER
                redefine roommate end
feature
        roommate: GIRL
                        -- This skier's roommate.
end -- class GIRL
```

We should correspondingly redefine the argument to procedure *share*, so that a more complete version of the class text is:

```
class GIRL inherit
        SKIER
                redefine roommate, share end
feature
        roommate: GIRL
                        -- This skier's roommate.
        share (other: GIRL) is
                        -- Choose other as roommate.
                require
                        other /= Void
                do
                        roommate := other
                end
end -- class GIRL
```

All proper descendants must be adapted in this way (remember, we are depriving ourselves from anchored types for the moment). The general picture is this:

Skier hierarchy and redefinitions



Since inheritance is specialization, the type rules require that if we redefine the result of a feature, here *roommate*, the new type must always be a descendant of the original one. This also applies to the redefined type for the argument *other* of routine *share*. This policy, as we know, is called covariance, where the "co" indicates that the argument and result vary together; the reverse policy is termed *contravariance*.

Type Redeclaration rule, page 599.

Covariance is, according to all available evidence, what we need in practice. Our earlier software examples illustrate this clearly:

- A *LINKABLE* list element may be chained to any other linkable; a *BI_LINKABLE* must be chained to another *BI_LINKABLE*. So the argument of procedure *put_right* should be redefined covariantly.

"Type inconsistencies", page 599.

- In the same example, any routine of *LINKED_LIST* that uses an argument of type *LINKABLE* will most likely need it to be of type *BI_LINKABLE* in *TWO_WAY_LIST*.

Figure "Parallel hierarchies", page 598.

- Procedure *set_alternate* takes a *DEVICE* argument in class *DEVICE*, a *PRINTER* argument in class *PRINTER*.

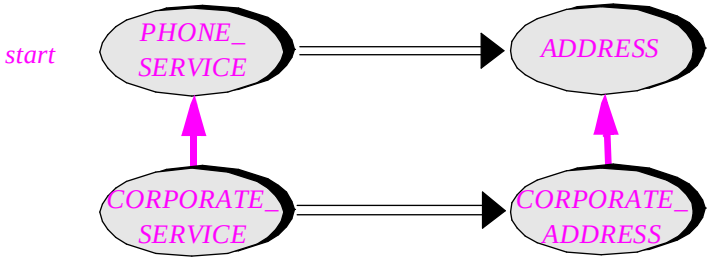
Covariant redefinition is particularly common because of the O-O method's emphasis on information hiding, which leads to procedures of the form

```
set_attrib (v: SOME_TYPE) is
            -- Set attrib to v.
    …
```

with *attrib* of type *SOME_TYPE*; such procedures are naturally covariant (and in practice, as we know, will usually rely on anchored types) since any class that changes the type of *attrib* will need to redefine *set_attrib*'s argument in the same way. The preceding examples mostly belonged to this scheme, but it is by no means the only one requiring covariance. Think for example of a procedure or function for concatenating a *LINKED_LIST* to another: its argument will have to be redefined as a two-way-list in *TWO_WAY_LIST*. The general addition operation, **infix** "+", takes a *NUMERIC* argument in

NUMERIC, a *REAL* argument in *REAL*, an *INTEGER* argument in *INTEGER*. In the parallel hierarchies

Phone service and billing addresses

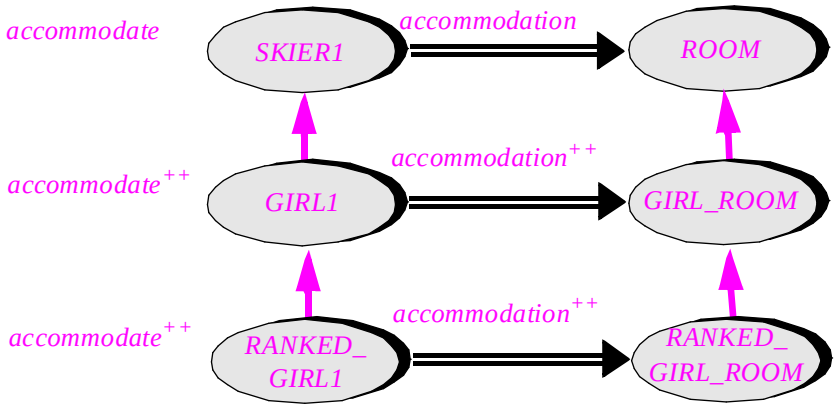


procedure *start*, which starts a phone service, may need an argument of type *ADDRESS* representing the billing address; for a corporate account you will need a corporate address.

What about a contravariant solution? In the skier example, contravariance would mean that if we go to class *RANKED_GIRL*, where the result of *roommate* is redefined to be of type *RANKED_GIRL*, we may for the argument of routine *share* use type *GIRL*, or *SKIER* of the most general kind. One type that is *not* permitted under contravariance is *RANKED_GIRL*! Enough to justify the parents' worst fears.

Parallel hierarchies

To leave no stone unturned, it is useful to consider a variant of the *SKIER* example with two parallel hierarchies, rather than just one. This will model the situation evidenced in software examples already cited: *TWO_WAY_LIST* → *LINKED_LIST* parallel to *BI_LINKABLE* → *LINKABLE*, or the *PHONE_SERVICE* hierarchy. Just assume that we have a *ROOM* hierarchy with descendants such as *GIRL_ROOM* (*BOY* variants omitted):



Then instead of *roommate* and *share*, the skier classes will have features *accommodation* and *accommodate*:

```
indexing
        description: "New variant with parallel hierarchies"
class SKIER1 feature
        accommodation: ROOM
        accommodate (r: ROOM) is … require … do
                        roommate := other
                end
end -- class SKIER1
```

Here too we need covariant redefinition: in class *GIRL1* both *accommodation* and the argument of *accommodate* should be redeclared of type *GIRL_ROOM*, in *BOY1* they should be of type *BOY_ROOM*, and so on. (Remember again that for the time being we are working without anchored types.) A contravariant policy would be as useless as in the preceding form of the example.

Polymorphic perversity

Enough covariant examples. Why would anyone consider contravariance, which goes against what we need in practice (not to mention proper behavior for young people)? To understand, we have to consider the problems that polymorphism may cause under a covariant policy. A harmful scheme is easy to make up, and you may have thought of it yourself already:

```
s: SKIER; b: BOY; g: GIRL
…
!! b; !! g;          -- Creation of a BOY and GIRL objects.
s := b;              -- Polymorphic assignment.
s.share (g)
```

The effect of the last call, although possibly to the boys' liking, is exactly what the type definitions were attempting to exclude. A room assignment makes a boy object, known as *b* but also disguising itself polymorphically under the *SKIER* pseudonym *s*, the roommate of the *GIRL* object attached to *g*. Yet the call appears type-correct, since *share* is an exported feature of class *SKIER*, and *GIRL*, the type of argument *g*, conforms to *SKIER*, the type declared for the formal argument of *share* in *SKIER*.

The corresponding scheme with the parallel hierarchy variant is just as simple: just replace *SKIER* by *SKIER1* etc., and the call to *share* by a a call *s.accommodate* (*gr*), where *gr* is of type *GIRL_ROOM*: at run time this will assign a boy to a girl room.

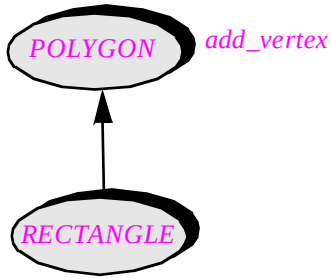
With contravariance one would not have these problems: as you specialize the target of a call (*s* in the example), you would generalize the argument. Contravariance, as a result, leads to simpler mathematical models of the inheritance-redefinition-polymorphism mechanism. For that reason a number of theoretical articles have advocated contravariance. But the argument is not very convincing, since, as we have seen and as the literature readily admits, contravariance is of essentially no practical use.

> An argument often encountered in the programming literature is that one should strive for techniques that have simple mathematical models. Mathematical elegance, however, is only one of several design criteria; we should not forget to make our designs realistic and useful too. In computing science as in other disciplines, it is after all much easier to devise dramatically simple theories if we neglect to make them agree with reality.

So rather than trying to force a covariant body into a contravariant suit, we should accept the reality for what it is, covariant, and study ways to remove the unpleasant effects.

Descendant hiding

Before looking for solutions to the covariance problem, let us examine the other mechanism that can cause type violations through polymorphism. Descendant hiding is the ability for a class not to export a feature that was exported by one of its parents.



A typical example is a feature *add_vertex*, which class *POLYGON* exports but its descendant *RECTANGLE* hides, because it would violate the invariant of the class:

```
class RECTANGLE inherit
        POLYGON
                export {NONE} add_vertex end
feature
        ...
invariant
        vertex_count = 4
end
```

A non-software counterpart is the well-known example of *OSTRICH* inheriting from a class *BIRD* equipped with a feature *fly*, which *OSTRICH* should not export.

"SUBTYPE INHERITANCE AND DESCENDANT HIDING", 24.7, page 835.

Let us for the moment accept this scheme at face value, setting aside the question, discussed in detail later, of whether such forms of inheritance are methodologically legitimate. The modeling power of descendant hiding, like that of covariance, clashes with the tricks made possible by polymorphism. An example is trivial to build:

```
p: POLYGON; r: RECTANGLE
…
!! r;                    -- Creation of a RECTANGLE object.
p := r;                  -- Polymorphic assignment.
p.add_vertex (…)
```

Since *add_vertex* is an exported feature of *POLYGON*, the call appears type-correct; if accepted, it would on execution add a vertex to a rectangle, producing an inconsistent object.

Class and system validity

Some terminology will be useful to discuss the issues raised by covariance and descendant hiding. A system is **class-valid** if it satisfies the type rules summarized at the beginning of this chapter: every entity declared with a type; every assignment and actual-formal argument association satisfies conformance; and every call uses a feature of the target's type, exported to the caller.

The system is **system-valid** if no type violation can occur at run time.

Ideally these two notions (whose names will be justified later in this chapter) should be equivalent. What we have seen through the preceding examples is that with covariance and descendant hiding a system can be class-valid without being system-valid. Such an error — making a system invalid although it is class-valid — will be called a **system validity error**.

For an explanation of the names see end of 17.6, page 636.

Practical scope

The simplicity of the examples of system validity error, resulting from covariance or descendant hiding, makes up what we may call the static typing paradox. On being introduced to object-oriented typing, an inquisitive newcomer can make up such a counter-example in a few minutes; yet in actual development, while violations of class-level validity rules are common (and, caught by the compiler, provide tremendous help in getting the software right), system validity errors are exceedingly rare, even in large, multi-year projects.

This is not an excuse for ignoring them. The rest of this chapter investigates three possible solutions.

An important note: because the problems discussed next are both delicate and infrequent, it is reasonable and indeed suggested, if this is your first reading, that you skip the rest of this chapter unless you are already well-versed in the practical and theoretical aspects of object technology. If you are relatively new to the approach, you will understand the discussion much better after reading the methodological chapters of part D, in particular chapter 24 on the methodology of inheritance.

SUGGESTED SHORTCUT: skip to next chapter.

17.4 FIRST APPROACHES TO SYSTEM VALIDITY

Let us concentrate first on the covariance issue, the more challenging of the two. There is an abundant literature on the subject and we can take a look at various proposed solutions.

Contravariance and novariance

Adopting a contravariant policy removes the theoretical problem of system validity errors. But this approach makes the type system unrealistic, so we need not examine it further.

C++ is original in using a *novariant* policy: when you redefine a routine, you cannot change the types of its arguments! If C++ were a strongly typed language, this would make the type system quite unusable. The easiest solution, as with other such limitations of C++ (such as the absence of constrained genericity), is to use casts, and so to bypass the typing mechanism altogether. This solution is not particularly attractive. Note, however, that some of the proposals discussed next rely on a form of novariance, made meaningful by the introduction of new type mechanisms to replace covariant redefinition.

Using generic parameters

An interesting idea, originally introduced by Franz Weber, relies on genericity. We can declare our class *SKIER1* with a generic parameter representing the room: **class** *SKIER1* [*G*] or rather, using constrained genericity,

```
class SKIER1 [G –> ROOM] feature
        accommodation: G
        accommodate (r: G) is … require… do accommodation := r end
end
```

Then class *GIRL1* will inherit from *SKIER1* [*GIRL_ROOM*] and so on. The same technique may be applied to the variant without parallel hierarchies, although it seems stranger at first: **class** *SKIER* [*G –> SKIER*].

This approach solves the covariance problem. In any use of the class you need to specify an actual generic parameter — such as *ROOM* or *GIRL_ROOM* —, so the invalid combinations become impossible. The language would become novariant, and systems would satisfy their covariance needs entirely through generic parameters.

Unfortunately, the generic parameter technique is not really acceptable as a general solution. It will lead to inflated generic parameter lists, with one parameter for each type of a possibly covariant argument. To use the class, a developer will have to provide as many types as there are parameters; this will make classes hard to understand.

Worse, adding a covariant routine with an argument of a type not yet covered would require adding a generic parameter to the class, and hence changing its interface, thereby invalidating all client classes. This is not acceptable.

Type variables

Several authors (including Kim Bruce, David Shang, Tony Simons) have proposed solutions based on the introduction of type variables. Although it is impossible to summarize these sophisticated proposals without being unfair, the basic idea is simple: instead of covariant redefinition, permit type declarations to use type variables rather than actual types; extend the conformance rules to handle type variables; make the language otherwise novariant; provide a facility to assign a type value to a type variable.

Instead of *ROOM*, the declarations for attribute *accommodation* and for the argument of *accommodate* would use a type variable, to which an actual type value can be assigned separately.

These proposals are worth considering, and the interested reader should consult the corresponding articles, as well as complementary publications by Cardelli, Castagna, Weber and others, starting from the paper and Web references cited in the bibliographical notes to this chapter. We will not, however, pursue this line, for two reasons:

- The type variable mechanism, if designed properly, should subsume genericity and anchored declarations, the two existing mechanisms for using a type without fully specifying it. At first this can be construed as an argument in favor of type variables, as they might enable us to replace two language constructs by one, and solve other problems at the same time. But the result may not be satisfactory in practice since both genericity and anchored types are simple, widely accepted and easy to explain; it is not clear that an all-encompassing type variable mechanism can do as well.
- Assuming we can indeed devise a type variable mechanism that solves the technical difficulties of combining covariance and polymorphism (still ignoring descendant hiding for the moment), it will require *perfect foresight* from the class designer: knowing in advance which features are subject to type redefinition in descendants, and which are not. The following section will further discuss this problem, which arises from a practical software engineering concern and, unfortunately, hampers the credibility of many theoretically satisfying schemes.

These considerations suggest trying a different approach: examining the mechanisms that we already have at our disposal — constrained and unconstrained genericity, anchored types, and of course inheritance — to see how they can be further constrained to remove the possibility of system validity errors.

17.5 RELYING ON ANCHORED TYPES

We can actually find an almost satisfactory solution to the covariance problem by taking a closer look at a mechanism that we already know well: anchored declarations.

You must indeed have been itching, in the *SKIER* and *SKIER1* examples, to use anchored declarations, removing most of the need for type redefinitions. Anchoring is the covariant mechanism *par excellence*: by declaring *y*: **like** *x*, you make *y* vary with *x*

whenever *x* gets redefined to descendant-based types in descendant classes. Our examples become:

Underlining indicates the change from earlier versions.

```
class SKIER feature
        roommate: like Current
        share (other: like Current) is … require … do
                roommate := other
            end
        …
end -- class SKIER

class SKIER1 feature
        accommodation: ROOM
        accommodate (r: like accommodation) is … require … do
                accommodation := r
            end
end -- class SKIER1
```

Then descendants need no redefinition in the *SKIER* version, and in the *SKIER1* version they only need to redefine attribute *accommodation*. The anchored entities — *roommate* and the arguments of *share* and *accommodate* — will automatically follow the anchors' redefinitions. This tremendous simplification, in line with what we saw in the original examples of anchored declaration, confirms that without anchoring (or some alternate mechanism such as type variables) it would be impossible to write realistic typed object-oriented software.

But does this eliminate system validity violations? No! At least not without a further restriction. We can still cheat the type checker into letting pass polymorphic assignments that will cause run-time type violations.

True, the original examples will be rejected. In

```
s: SKIER; b: BOY; g: GIRL
o
!! b;!! g;          -- Creation of a BOY and GIRL objects.
s := b;             -- Polymorphic assignment.
s.share (g)
```

"Rules on anchored types", page 604.

the argument *g* to *share* is not valid, since we need something of type **like** *s*, and *GIRL* does not conform to **like** *s*. The conformance rule for anchored types stated that no type conforms to **like** *s* other than this type itself.

The relief is short-lived, however, The same rule stated that, in the other direction of conformance, **like** *s* conforms to the type of *s*. So we can fool the type checker, although we have to be pretty devious, by using polymorphism not just on the target *s* of the call but on its argument *g*:

```
s: SKIER; b: BOY; g: like s; actual_g: GIRL;
o
!! b; !! actual_g                -- Creation of a BOY and GIRL objects.
s := actual_g; g := s            -- Go through s to attach g to the GIRL object.
s := b                           -- Polymorphic assignment.
s.share (g)
```

The effect is exactly the same as before.

There is a way out. If we are serious about using anchored declarations as the sole covariance mechanism, then we can get rid of system validity errors by prohibiting polymorphism altogether on anchored entities. This requires a language change: we would introduce a new keyword **anchor**, used in such declarations as

```
anchor s: SKIER
```

Warning: hypothetical construct, for purposes of discussion only.

Then we would permit a declaration of the form **like** *s* only if *s* is declared in this form, and adapt the conformance rule to make sure that *s* as well as elements of type **like** *s* can be attached (assigned or argument-passed) only to each other.

> In the original rule there was a notion of **anchor-equivalent** elements: with *x* declared of some non-anchored type *T* and *y* declared **like** *x*, then *x* and *y* are anchor-equivalent to each other and to any other entity anchor-equivalent to either of them. An attachment to an anchored target was valid only if the source was anchor-equivalent to the target (which makes the assignment *g*:= *s* valid even though *g* is anchored and *s* is not); but there was no such restriction the other way around: *z* := *y* was valid for any *z* of type *T*. With the new approach this would not be permitted any more; in any attachment involving an entity that is either anchor or anchored, the source and the target must be anchor-equivalent.

With this approach, we would remove from the language the possibility of redefining the type of any routine argument. (We could also prohibit redefining the result type, but this is not necessary. We must retain, of course, the possibility of redefining an attribute type.) *All* such redefinitions will now be obtained indirectly, through the anchoring mechanism, which enforces covariance. Where with the earlier approach a class *D* redefined an inherited feature as

```
r (u: Y) …
```

from an original version, in a proper ancestor *C* of *D*, that read

```
r (u: X) …
```

with *Y* conforming to *X*, you should now define the original in *C* as

```
r (u: like your_anchor) …
```

and only redefine in *D* the type of *your_anchor*.

This solution to the covariance-polymorphism issue will be called the **Anchoring** approach (short for the more accurate “Covariance through anchoring only”). Its properties make it particularly attractive:

- It is based on a clear concept: strictly separating the *covariant* elements from the *potentially polymorphic* ones (just “polymorphic” for short). Any entity declared as

anchor or as **like** *some_anchor* is covariant; any other is polymorphic. You can have attachments within each category; but no entity or expression will cross the boundary. For example you cannot assign a polymorphic source to a covariant target.

- The solution is simple, elegant, easy to explain even to relative beginners.
- It appears completely tight, removing any possibility of covariance-related system validity violation.
- It retains the framework defined in the preceding chapters, in particular the notions of genericity, constrained or not. (As a result it is, in my opinion, preferable to the introduction of type variables covering both covariance and genericity, since these two mechanisms address clearly distinct practical needs.)
- It entails a small language change — adding one keyword, reinforcing a conformance rule — and no foreseeable implementation difficulty.
- It is, at least in a theoretical sense, **realistic**: any system that was previously possible can be rewritten using the transformation just outlined, replacing covariant redefinitions by anchored redeclarations in the original. True, some attachments will become invalid as a result; but they correspond to cases that could have led to type violations, and can be replaced by assignment attempts, whose result the software can then check to ascertain at run time that everything is fine.

With such arguments we would seem to be at the end of the discussion. Why then is the Anchoring solution not fully satisfactory? First, it still leaves us with the descendant hiding issue. But the fundamental reason is the software engineering concern already voiced during our brief encounter with the notion of type variables. The Yalta-like division of the world into a polymorphic part and a covariant part assumes that the designer of a class always has perfect foresight: for every entity that he introduces, in particular every routine argument, he must decide once and for all between one of two possibilities:

- The entity is potentially polymorphic: now or later, it may become attached (through argument passing if it is a formal argument of a routine, through assignment otherwise) to objects of types other than its declared type. Then no descendant will be permitted to redefine that type.
- The entity is subject to type redefinition: then it is either anchored or an anchor itself.

Page 57.

But how can the designer be sure in each case? Much of the attraction of the object-oriented method, captured at the beginning of this book by the Open-Closed principle, comes from its support for late adaptation of original choices; from the way it accepts that designers of general-purpose modules need *not* have infinite wisdom, since authors of descendants can adapt some of their decisions.

In this imperfection-tolerant approach, both type redefinition and descendant hiding are a safety valve, which enables us to reuse an existing, almost-suitable class:

- With type redefinition, you can adapt the type declaration in the descendant without touching the original (to which, of course, you may lack source access or modification privileges). With the covariance-only solution you would need to change the original, using the transformation outlined earlier.

- Descendant hiding similarly preserves you from suffering too much from the bumps of the design process. True, one may criticize a design which has *RECTANGLE* inherit from *POLYGON* and still want *add_vertex* in *POLYGON*; instead, you may devise an inheritance structure that removes this problem, separating fixed polygons from variable ones. It is indeed preferable to stay away from *taxonomy exceptions* in designing inheritance structures. But can we eliminate them altogether? The discussion of descendant hiding in a later chapter (where we will encounter examples that cannot be restructured as easily as polygons and rectangles) suggests that we cannot, for two reasons. First, various classification criteria may compete: for example we may prefer to classify our polygons into regular and irregular ones. Second, we have to accept that even where an ideal solution is possible some designers will not have seen it, although we may still try to inherit from their classes.

See "SUBTYPE INHERITANCE AND DESCENDANT HIDING", 24.7, page 835, which discusses taxonomy exceptions.

If we want to preserve the flexibility of descendant adaptation, we will need to permit covariant type redefinition — not just through anchoring — and descendant hiding. The next sections describe how.

17.6 GLOBAL ANALYSIS

(This section describes an intermediate approach; readers interested in an overview of the main practical solutions may skip to the next section.)

Skip to "BEWARE OF POLYMORPHIC CAT-CALLS!", 17.7, page 636.

In studying the Anchoring solution we noted that the basic idea was to separate the covariant part from the polymorphic part. Indeed, if you consider the two instructions in

s := *b* …
s.*share* (*g*)

each is a legitimate application of an important O-O mechanism: the first applies polymorphism; the second uses type redefinition. Things start to go wrong when you combine these operations for the same *s*. Similarly, in

p := *r* …
p.*add_vertex* (…)

the problem arises from the combination of two individually blameless operations. Here too you can use either instruction by itself without a hitch; include both and you are in trouble.

The type violations follow from erroneous calls. In the first example, the polymorphic assignment attaches *s* to a *BOY* object, making *g* an illegal argument to *share* since *g* is attached to a *GIRL* object. In the second example the assignment attaches *r* to a *RECTANGLE* object, making *add_vertex* a non-exported feature.

Hence an idea for a new solution: determine in advance — statically, as part of the type checking performed by the compiler or set of tools — the **typeset** of each entity, short for "dynamic type set", comprising the types of all objects to which the entity might become attached at run time. Then verify, still statically, that each call is valid for each element of the typesets of the target and arguments.

In our examples, the assignment *s* := *b* indicates that *BOY* is in the typeset of *s* (because *BOY* is in the typeset of *b* as a result of the creation instruction !! *b*); *GIRL* is in the typeset of *g* because of the instruction !! *g*; but then the call to *share* would not be valid for a target *s* of type *BOY* and an argument *g* of type *GIRL*. Similarly, *RECTANGLE* is in the typeset of *p* because of the polymorphic assignment, but the call to *add_vertex* would not be valid for *p* of type *RECTANGLE*.

These observations lead to what we may call the **Global** approach, based on a new typing rule:

System Validity rule

A call *x*.*f* (*arg*) is system-valid if and only if it is class-valid for *x* having any type in its own typeset, and *arg* having any type in its own typeset.

In this definition a call is "class-valid" if it is valid according to the Feature Call rule recalled at the beginning of this chapter: if *C* is the base class of *x*'s type, *f* must be an exported feature of *C*, and the type of *arg* must conform to the type of the formal argument of *f*. (Remember that for simplicity we assume that each routine has exactly one argument; the rule is trivially transposed to an arbitrary number of arguments.)

System validity is the same thing as ordinary class validity, except that we do not just consider the type declared for the target *x* and the arguments *arg*: we apply class validity to every possible type in their typesets.

Here is the basic rule for determining the typeset of all entities:

T1 • Start out with an empty typeset for every entity.

T2 • For every creation instruction of the form ! *SOME_TYPE* ! *a*, add *SOME_TYPE* to the typeset of *a*. (For simplicity, assume that any instruction !! *a* has been replaced by ! *ATYPE* ! *a*, where *ATYPE* is the type declared for *a*.)

T3 • For every assignment of the form *a* := *b*, add all the elements of the typeset of *b* to the typeset of *a*.

T4 • If *a* is a formal argument of a routine, for every corresponding actual argument *b* in a call, add all the elements of the typeset of *b* to the typeset of *a*.

T5 • Repeat steps T3 and T4 until no typeset changes.

This description does not take genericity into account, but the extension is not hard. The repetition (T5) is necessary because of the possibility of attachment chains (an attachment of *b* to *a*, of *c* to *b* and so on). It is easy to see, however, that the process will terminate after a finite number of steps.

> The number of steps is bounded by the maximum length of attachment chains, that is to say the maximum *n* such that the system contains attachments of x_{i+1} to x_i for $i = 1, 2, \ldots n-1$. The repetition of T3 and T4 is known as a "fixpoint" technique.

As you may have noted, the rule does not consider instruction sequencing. For example, in

```
! TYPE1 ! t;    s := t;    ! TYPE2 ! t
```

we will include both *TYPE1* and *TYPE2* into the typeset of *s*, even though *s* can only, with the instructions given, become attached to an object of type *TYPE1*. Taking instruction sequencing into account would force the compiler to perform extensive flow analysis, leading to undue complexity. Instead, the rules are more pessimistic: they will flag any occurrence of all three operations

```
!! b
s := b
s.share (g)
```

as system-invalid, even if their possible run-time sequencing cannot possibly lead to a type violation.

The global analysis approach was presented (with more details) in chapter 22 of [M 1992]. It solves both the covariance problem and the descendant hiding problem. It suffers, however, from an annoying practical deficiency: although it does not require flow analysis, it assumes that you are checking an **entire system** at once, rather than each class incrementally. The killer rule is T4, which for any call *x.f (b)* corresponding to a routine *f (a: ARG_TYPE)*, adds the typeset of *b* to that of *a*. If *f* is a routine from a library class, this means that adding a call to *f* in a new client can affect the typesets of *f*'s formal arguments, and ripple over to existing calls in other clients.

Although there have been proposals for incremental algorithms [M 1989b], their practicality has not been established. This means that in a development environment supporting incremental compilation the global analysis technique would need to be implemented as a check on an entire system, rather than as part of the local (and fast) operations that the compiler performs each time a user changes a few classes. Even though there are precedents for such an approach — C developers, for example, sometimes rely on a tool called **lint**, separate from the compilation process, to look for inconsistencies — it is not really attractive, especially in today's sophisticated environments whose users expect the tools to provide fast and complete responses.

As a result, the global validity approach has not to my knowledge been implemented. (Another reason is probably that the rule may appear difficult to teach, especially when given with all the details of genericity etc.)

In passing we have seen the reason for some terminology used since the beginning of this discussion. A system was said to be **class-valid** if it satisfied the basic type rules according to each entity's type declaration; the name indicates that, as we just saw, this can be checked (and checked fast) by an incremental compiler working class-by-class. A system may be class-valid but not yet **system-valid** if its execution can still cause type violations. With the techniques seen so far, detecting this possibility seems to require a global (system-wide) analysis.

"Class and system validity", page 627.

In spite of the name, however, it is in fact possible to avoid system validity errors through completely incremental checking. This will be our final tack on the issue.

17.7 BEWARE OF POLYMORPHIC CATCALLS!

Pessimism in type checking was discussed in "Pessimism", page 614.

The System Validity rule of global analysis, it was noted, is pessimistic: to simplify type rules and their enforcement, it may reject harmless combinations. Paradoxical as this may seem, we will obtain our last solution by turning to an even **more pessimistic** rule. This will of course raise the question of how realistic the result is.

Back to Yalta

The gist of the **Catcall** solution — the name, to be explained shortly, for the new approach — is to come back to the Yalta-like character of the Anchoring solution, dividing the world into a polymorphic part and a covariant part (the latter also having, as its satellite, a descendant hiding part), but to remove the need for perfect foresight.

As before we narrow down the covariance issue to two operations: in our main example, the polymorphic assignment, *s* := *b*, and the call to a covariant routine, *s.share* (*g*). Analyzing what is truly wrong, we note that the argument *g* is not an issue in itself; any other argument, which has to be of type *SKIER* or a descendant, would be just as bad since *s* is polymorphic and *share* covariantly redefines its argument. So with *other* statically declared of type *SKIER* and dynamically attached to a *SKIER* object, the call *s.share* (*other*), which would seem to be ideally valid on its static face, will cause a type violation if *s* has been polymorphically assigned the value of *b*.

The fundamental problem, then, is that we are trying to use *s* in two incompatible ways: as a polymorphic entity; and as the target of a call to a covariant routine. (In the other working example, the problem is that we use *p* as both polymorphic entity and target of a call to a descendant-hidden routine *add_vertex*.)

The Catcall solution is drastic, in line with the Anchoring solution: it prohibits using an entity both polymorphically and covariantly. Like the Global solution, it will determine statically which entities can be polymorphic, but it will not try to be smart: instead of finding out the typeset, it just treats any polymorphic entity as suspect enough to warrant lifetime exclusion from any covariance or descendant hiding establishment.

Rule and definitions

The type rule of the Catcall approach is simple:

Catcall type rule

Polymorphic catcalls are invalid.

This is based on equally simple definitions. First, polymorphic entity:

Definition: Polymorphic entity

An entity *x* of reference (non-expanded) type is polymorphic if it satisfies any of the following properties:

P1 • It appears in an assignment *x* := *y* where *y* is of a different type or (recursively) polymorphic.

P2 • It appears in a creation instruction ! *OTHER_TYPE* ! *x* where *OTHER_TYPE* is not the type declared for *x*.

P3 • It is a formal routine argument.

P4 • It is an external function.

The aim of the definition is to capture as polymorphic ("potentially polymorphic" would be more accurate) any entity that may at run time become attached to objects of more than one type. The definition only applies to reference types, since expanded entities cannot by nature be polymorphic.

In our examples, the skier *s* and the polygon *p* are both polymorphic from rule P1, since they appear in assignments, the first with a boy *b* and the second with a rectangle *r*.

If you have read the definition of the typeset concept in the Global approach, note how much more pessimistic the notion of polymorphic entity is, and simpler to check. Instead of trying to find out all the possible dynamic types of an entity, we settle for a binary property: can it be polymorphic, or can it not? Most strikingly (rule P3), we consider that **any formal argument of a routine is polymorphic** (unless it is expanded, as with integers and the like). We do not even bother to consider the calls to a routine: if you are an argument, you are at the beck and call of any client, so we cannot trust your type. This rule is closely tied to the reusability goal of object technology, where any class has the potential, ultimately, to become part of a reusable library where any client software will be able to call it.

The distinctive feature of this rule is that it does not require any global check. To determine whether an entity is polymorphic, it suffices to examine the text of a class. There is not even any need to examine proper ancestors' texts, provided we record, for each query (attribute or function) of each class, whether it is polymorphic. (We need this information since under P1 the assignment *x* := *f* will make *x* polymorphic if *f* is polymorphic, whether or not it comes from the same class.) Unlike the computation of typesets in the Global approach, the detection of polymorphic entities can proceed class by class, as part of the checks performed by an incremental compiler.

As discussed in the presentation of inheritance, this analysis can also be precious for optimization purposes

See optimization S2, page 511.

Calls, as well as entities, may be polymorphic:

Definition: Polymorphic call

A call is polymorphic if its target is polymorphic.

The calls of both examples are polymorphic: *s*.*share* (*g*) since *s* is polymorphic, and *p*.*add_vertex* (…) since *p* is polymorphic. The definition implies that only qualified calls *a*.*f* (…) can be polymorphic. (Writing an unqualified call *f* (…) as *Current*.*f* (…) changes nothing since *Current*, to which no assignment is possible, cannot be polymorphic.)

Next we need the notion of catcall, based on the notion of CAT. A routine is a CAT (short for Changing Availability or Type) if some redefinition of the routine, in a descendant, makes a change of one of the two kinds we have seen as potentially troublesome: retyping an argument (covariantly), or hiding a previously exported feature.

Definition: CAT (Changing Availability or Type)

A routine is a CAT if some redefinition changes its export status or the type of any of its arguments.

This property is again incrementally checkable: any argument type redefinition or change of export status makes a routine a CAT. It yields the notion of catcall: any call that a CAT change could make invalid. This completes the set of definitions used by the Catcall type rule:

Definition: Catcall

A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.

Page 637.

The Catcall type rule promotes our Yalta view by separating calls into two disjoint categories: polymorphic calls and catcalls. Polymorphic calls yield some of the expressive power of the O-O method; catcalls yield the ability to redefine types and hide features. Using terminology introduced at the beginning of this chapter: polymorphism enhances the *usefulness* of the approach; type redefinition enhances its *usability*.

The calls of our examples are catcalls since *share* redefines its argument covariantly, and *add_vertex*, exported in *RECTANGLE*, is hidden in *POLYGON*. Since they are also polymorphic, they are prime examples of polymorphic catcalls and hence made invalid by the Catcall type rule.

17.8 AN ASSESSMENT

Before trying to summarize what we have learned on the covariance and descendant hiding issues, we should recall once more that system validity violations arise extremely rarely. The most important properties of static O-O typing are the ones summarized at the

beginning of this chapter: the impressive array of type-related mechanisms which, with class-level validity, open the way to a safe and flexible method of software construction.

We have seen three solutions to the covariance problem, two of them also addressing descendant hiding. Which one is right?

The answer may not be final. The consequences of subtle interactions between O-O typing and polymorphism are not as well understood as the topics of the preceding chapters. The past few years have seen the appearance of numerous publications on the question, to which the bibliographical notes give the basic pointers. I hope that the present chapter has provided the elements for a definitive solution or something close to it.

The Global solution seems impractical because of the implied need for system-wide checking. But it helps understand the issue.

The Anchoring solution is extremely tempting. It is simple, intuitive, easy to implement. We must all the more regret its failure to support some of the key software engineering requirements of the object-oriented method, as summarized by the Open-Closed principle. If you have perfect foresight, then the Anchoring solution is great; but what designer can promise to have perfect foresight, or assume perfect foresight from the authors of the library classes he reuses through inheritance?

> This assumption limits the usefulness of many of the published approaches, such as those relying on type variables. If we can be assured that the developer always knows in advance which types may change, the theoretical problem becomes much easier, but it does not accurately model the practical problem of typed object-oriented software construction.

If we must give up the Anchoring approach, the Catcall type rule seems to be the appropriate one, easy enough to explain and enforce. Its pessimism should not exclude useful combinations. If a case that appears legitimate yields a polymorphic catcall, it is always possible to let it through safely by introducing an assignment attempt; this is a way to transfer some of the checks to run time. This should only happen in a marginal number of cases.

As a caveat, I should note that at the time of writing the Catcall solution has not yet been implemented. Until a compiler has been adapted to enforce the Catcall type rule and applied successfully to many representative systems, small and large, where success means evidence that the rule is realistic (that all useful systems will pass muster, possibly at the expense of a few easily justifiable changes) and that checking it imposes no significant penalty on incremental recompilation times, we must refrain from proclaiming that on the problem of reconciling static typing and polymorphism with covariance and descendant hiding we have heard the last word.

17.9 THE PERFECT FIT

As a complement to the discussion of covariance it is useful to study a general technique addressing a common problem. This technique was devised as a result of the Catcall theory, but it can be used in the basic language framework without any new rule.

Assume that we have two lists of skiers, where the second list includes the roommate choice of each skier at the corresponding position in the first list. We want to perform the

corresponding *share* operations, but only if they are permitted by the type rules, that is to say girls with girls, ranked girls with ranked girls and so on. Problems of this kind are presumably frequent.

A simple solution is possible, based on the preceding discussion and on assignment attempt. Consider the following general-purpose function:

```
fitted (other: GENERAL): like other is
                -- Current object if its type conforms to that of object attached to
                -- other; void otherwise.
        do
                if other /= Void and then conforms_to (other) then
                        Result ?= Current
                end
        end
```

On GENERAL, conforms_to and same_type, see "Universal features", page 582.

Function *fitted* returns the current object, but known through an entity of a type anchored to the argument; if this is not possible, that is to say if the type of the current object does not conform to that of the object attached to the argument, it returns void. Note the role of assignment attempt. The function relies on *conforms_to*, a feature of class *GENERAL* that determines whether the type of an object conforms to that of another.

Replacing *conforms_to* by *same_type*, another *GENERAL* feature, yields a function *perfect_fitted* that returns void unless the types are exactly the same.

Function *fitted* gives us a simple solution to the problem of matching skiers without violating type rules. We can for example add the following procedure to class *SKIER* and use it in lieu of *share* (perhaps making *share* a secret procedure for more control):

```
safe_share (other: SKIER) is
                -- Choose other as roommate if permissible.
        local
                gender_ascertained_other: like Current
        do
                gender_ascertained_other := other.fitted (Current)
                if gender_ascertained_other /= Void then
                        share (gender_ascertained_other)
                else
                        "Report that matching is impossible for other"
                end
        end
```

For *other* of arbitrary *SKIER* type — not just **like** *Current* — we define a version *gender_ascertained_other* which has a type anchored to *Current*. To enforce identical types — so that a *RANKED_GIRL* goes only with another *RANKED_GIRL*, not with a mere *GIRL* — use *perfect_fitted* instead of *fitted*.

If you have two parallel lists of skiers, representing planned roommate assignments:

```
occupant1, occupant2: LIST [SKIER]
```

you can iterate over the lists, applying at each stage

occupant1.*item*.*safe_share* (*occupant2*.*item*)

to match elements at corresponding positions if and only if their types are compatible.

I find this technique elegant; I hope you will too. And of course parents anxious about what really happens during the ski trip should breathe a sigh of relief.

17.10 KEY CONCEPTS STUDIED IN THIS CHAPTER

- Static typing is essential for reliability, readability and efficiency.
- Static typing, to be realistic, requires a combination of mechanisms, including assertions, multiple inheritance, assignment attempt, constrained and unconstrained genericity, anchored declarations. The type system must not allow loopholes ("casts").
- Practical rules for routine redeclarations should permit covariant redeclaration: both results and arguments may be redefined to types conforming to the originals.
- Covariance, as well as the ability to hide in a descendant a feature that was exported in an ancestor, raise the rare but serious possibility of type violations when combined with polymorphism.
- Such type violations can be avoided through global analysis (impractical), limiting covariance to anchored types (conflicting with the Open-Closed principle), or the "catcall" technique which bars any covariance or descendant hiding for any routine used with a polymorphic target.

17.11 BIBLIOGRAPHICAL NOTES

Some of the material of this chapter originated with a keynote talk given at the OOPSLA 95 and TOOLS PACIFIC 95 conferences and published as [M 1996a]. Some of the overview material has been drawn from a journal article, [M 1989e].

The notion of automatic type inference was introduced by [Milner 1989], which describes an inference algorithm for the functional language ML. The connection between polymorphism and type checking is further explored in [Cardelli 1984a].

Techniques for improving the efficiency of dynamically typed language implementations are described, in the context of the Self language, in [Ungar 1992].

Luca Cardelli and Peter Wegner are the authors of an influential theoretical article on types in programming languages [Cardelli 1985]; using lambda calculus as the mathematical framework, it has served as a basis for much of the subsequent work. It followed another foundational article by Cardelli [Cardelli 1984].

For an introduction to lambda calculus Ph. Élinck: "De la Conception-Programmation par Objets", Mémoire de licence, Université Libre de Bruxelles (Belgium), 1988.

An ISE manual [M 1988a] included a brief presentation of the issues raised by the combination of polymorphism with covariance and descendant hiding. The absence of such an analysis in the first edition of this book led to some critical discussions (predated by comments in a student's bachelor thesis report by Philippe Élinck), notably [Cook 1989] and [America 1989a]. Cook's paper showed several examples of the covariance problem

and attempted a solution. At TOOLS EUROPE 1992, Franz Weber proposed a solution based on the use of generic parameters for covariant entities [Weber 1992]. [M 1992] defines precisely the notions of class-level and system-level validity, and proposes a solution based on system-wide analysis. The Catcall solution described in the present chapter was first presented in [M 1996a]; see also on-line material [M-Web].

The Anchoring solution was presented in a talk I gave at a TOOLS EUROPE 1994 workshop. I had, however, overlooked the need for **anchor** declarations and the associated restriction on conformance. Paul Dubois and Amiram Yehudai immediately pointed out that the covariance problem could still arise under these conditions. Along with others including Reinhardt Budde, Karl-Heinz Sylla, Kim Waldén and James McKim, they provided many further comments that were fundamental to the work leading to the present chapter (without being committed to its conclusions).

There is an abundant literature on the covariance issue; [Castagna 1995] and [Castagna 1996] provide both a bibliography and a mathematical overview. For a list of links to on-line articles on O-O type theory and researchers' Web pages, see Laurent Dami's page [Dami-Web]. The terms "covariance" and "contravariance" come, by the way, from category theory; it appears that their introduction into discussions of software typing is due to Luca Cardelli, who started to use them in talks in the early eighties, although they seem not to have appeared in print until the end of that decade.

Techniques based on type variables are described in [Simons 1995], [Shang 1996], [Bruce 1997].

The Sather language uses contravariance. [Szypersky 1993] presents the rationale.