4

Classes

4.1 OVERVIEW

Classes are the components used to build Eiffel software.

Classes serve two complementary purposes: they are the modular units of software decomposition; they also provide the basis for the type system of Eiffel.

This chapter explores the role of classes and the structure of class texts.

4.2 OBJECTS

Viewed as a type, a class describes (as noted in the previous chapter) the properties of a set of possible data structures, or **objects**, which may exist during the execution of a system that includes the class; these objects are called the **instances** of the class

→ The precise nature of objects is explained in chapter 19.

An object may represent a real-world thing such as a radio signal in cell phone software, a document in text processing software or an electron in physics software. It may also represent an immaterial concept from that world, such as a fabrication process in factory control software. Or it may be a pure artefact of computer programming, such as an abstract syntax tree in compilation software.

Classes corresponding to these examples might be:

- *SIGNAL*, whose instances represent signals transmitted by some device.
- *DOCUMENT*, whose instances represent documents.
- *ELECTRON*, whose instances represent electrons.
- *NODE*, whose instances represent nodes of syntax trees.

Every object that may exist during the execution of a system is an instance of some class of that system. This is an important property, since it means that the type system is simple and uniform, being entirely based on the notion of class.

→ Chapter 11 covers types.

More precisely, every object is a **direct instance** of only one class, called its **generating class**. It may, however, be an *instance* (direct or not) of many classes: all the ancestors (in the sense of inheritance) of its generating class.

→ *For exact definitions: "instance" and "direct instance", page 322; "Generating class", page 498; "ancestor", page 172.*

Some classes, said to be **deferred**, have no direct instances; they provide incomplete object descriptions. If *C* is deferred, an instance of *C* is a direct instance of some effective (that is to say non-deferred) descendant of *C*.

→ *See chapters 6 and 10 about inheritance and deferred routines.*

4.3 FEATURES

Viewed as a module, a class introduces, through its class text, a set of **features**. Some features, called **attributes**, represent fields of the class's direct instances; others, called **routines**, represent computations applicable to those instances.

→ *Features are studied in detail in chapter 5.*

Since there is no other modular facility than the class, building a software system in Eiffel means identifying the types of objects the system will manipulate, and writing a class for each of these types.

A system that includes a certain class will usually contain operations to create instances of that class (creation instructions and expressions, for a non-deferred class) and to apply features to those instances (feature calls).

→ *Creation: chapter 20; calls: chapter 23.*

4.4 USE OF CLASSES

In some cases, one of the two roles of classes is more important than the other.

- At one extreme, a class may be interesting only as a module encapsulating a number of routines. (It then resembles the "packages" of older programming languages.) Often, it will not then have any variable attributes. A system that uses such a class will not create any direct instances of it; instead, other classes of the system will make use of its features by inheriting from it, or through "non-object calls".

Even though it has no direct instances, such a class will have instances — the direct instances of proper descendants.

→ *"NON-OBJECT CALLS", 23.9, page 621.*

- At the other end, you may want to introduce a class simply because you need to describe a new type of object, without necessarily thinking of its role in the system architecture, at least at first. (It then resembles the "records" or "structures" of older programming languages, although it will usually include routines along with attributes.)

Both of these uses of classes arise in practice and both are legitimate.

In most cases, however, classes live up to their reputation, making a name for themselves in both the module and type worlds.

4.5 THE CURRENT CLASS

> **Current class**
>
> The **current class** of a construct specimen is the class in which it appears.

Every Eiffel software element — feature, expression, instruction, … — indeed appears in a class, justifying this definition. Most language properties refer directly or indirectly, through this notion, to the class in which an element belongs.

This will be complemented by the notion of "current type", which includes the formal generic parameters.

→ *"Current type", page 357*

4.6 CLASS TEXT STRUCTURE

A class text contains the class name and a number of parts, all optional except for Class_header, and all except Formal_generics introduced by a keyword:

- Notes, beginning with **note**.
- Class_header, beginning with one of: **class**; **deferred class**; **expanded class**; **separate class**.
- Formal_generics, beginning with a bracket [.
- Obsolete, beginning with **obsolete**.
- Inheritance, beginning with **inherit**.
- Creators, beginning with **create**.
- Converters, beginning with **converter**.
- Features, made of one or more Feature_clause each beginning with **feature**.
- Invariant, beginning with **invariant**.
- Notes again, for more specific index properties if desired.

Here is an extract from a class describing hash tables, which illustrates all clauses except Obsolete:

This class is a simplified form of one in the EiffelBase library. A "hash table" is a table used to record a number of elements, each identified by an individual key.

```
note
    description: "Hash tables used to store items associated %
                  with hashable keys."
    names: h_table, dictionary
    access: key, direct
    representation: array
    size: resizable
```

```
class HASH_TABLE [G, KEY –> HASHABLE] inherit
	TABLE [G, KEY]
		redefine
			load
		end

create
	make, from_tree
convert
	from_tree ({BINARY_SEARCH_TREE})

feature -- Initialization
	make (n: INTEGER)
			-- Allocate space for n items.
		… Procedure body omitted …
	load … Rest of procedure omitted

feature -- Access
	control: INTEGER
	Max_control: INTEGER is 5
feature -- Status report
	ok: BOOLEAN
			-- Was last operation successful?
		do
				Result := (control= 0)
		end
	… Other features omitted…

feature -- Removal
	remove (k: KEY)
			-- Remove entry of key k.
		require
			valid_key: is_valid (k)
		do
			… Procedure implementation omitted…
		ensure
			not has (k)
		end

invariant
	0 <= control; control <= Max_control
note
	date: "$Date: 1998/01/30 20:57:49 $"
	revision: "$Revision: 1.8 $"
	reviser: "Marcel Satchell, January 2000"
	changes: "Copy and equality semantics"
	original_author: "Eiffel Software, 1986"
end
```

This abbreviated example is a specimen of a Class_declaration, with the following general syntax:

Class declarations

Class_declaration ≜ [Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end

The next section offers an informal overview of the various parts and their roles, using *HASH_TABLE* as illustration. Subsequent sections of this chapter will only cover in detail Notes, Class_header, Formal_generics, Obsolete and the closing **end**; describing the rest is the task of the following chapters.

→ *Inheritance is discussed in chapters 6, 10, and 16, Creators in chapter 20, Features in chapter 5, and Invariant in chapter 9.*

4.7 PARTS OF A CLASS TEXT

As noted, class *HASH_TABLE* includes all of the possible parts save for Obsolete. Let's examine them informally, in their order of appearance.

The first Notes part serves to associate note information with the class, to facilitate identification, archival and retrieval of the class based on properties not found elsewhere in its text. The Notes part is studied in detail in the next section. It is organized as a sequence of clauses, each containing an optional Note term, such as *description*, a colon, and one or more associated values. Examples include a short description of the scope of the class (*description* entry), or alternate names for the notion covered by the class. The Note terms and values are free, but this example uses some of the recommended ones, part of the style guidelines.

→ *Deferred classes: 10.11, page 266 and subsequent sections.*

The Class_header introduces the class name, here *HASH_TABLE*. Instead of just **class**, the class header could begin with **deferred class**, **expanded class** or **separate class**, making the class "deferred", "expanded" or "separate".

→ *Expanded: 11.9, page 327 and subsequent sections. Separate: chapter 33. Genericity: chapter 12.*

The Formal_generics part, if present, makes the class "generic", which means it is parameterized by types. Here *HASH_TABLE* has two formal generic parameters: *G*, representing the type of the elements in a hash table; and *KEY*, representing the type of the keys which serve to retrieve these elements. To obtain a type from a generic class, you must provide types, called **actual generic parameters**. For example, you may declare an entity denoting a possible hash table as

```
ownership_record: HASH_TABLE [CAR, STRING]
```

using types *CAR* and *STRING* as actual generic parameters for *G* and *KEY*: the type *HASH_TABLE* [*CAR*, *STRING*] represents tables of cars retrievable through strings (perhaps the license plate numbers). A type obtained in this way is called a **generic derivation** of the base class, here *HASH_TABLE*. The entity *ownership_record* declared with this type may at run-time become attached to a table from which it is possible to retrieve cars from their associated strings.

The notation *KEY –> HASHABLE* in class *HASH_TABLE* indicates that the second formal generic parameter, *KEY*, is "constrained" by the library class *HASHABLE*. This means that any corresponding actual generic parameter must be a descendant of *HASHABLE*; this is indeed the case with class *STRING*. The first formal generic parameter, *G*, is "unconstrained", allowing any type to be used as the corresponding actual generic parameter.

→ On unconstrained and constrained generic derivations, see 12.3 and 12.6, starting on page 343.

The Obsolete part, if present, indicates that the class is an older version which should no longer be used except for compatibility with existing systems. For example, along with *HASH_TABLE*, a library may contain a class beginning with

```
class H_TABLE [G, KEY –> HASHABLE] obsolete
        "Use HASH_TABLE, which relies on improved algorithms"
inherit
        … Rest of class text omitted …
```

The only effect of such a clause is that some language processing tools may produce a warning when they process such a class. The warning should reproduce the String listed after the **obsolete** keyword.

The Inheritance part, beginning with **inherit**, lists the parents of the class and any **feature adaptation** applied to the inherited features. *HASH_TABLE* has only one parent, *TABLE*; its Feature_adaptation part, beginning with **redefine**, simply indicates that the new class will provide a new version of the inherited procedure *load*. There is indeed a declaration of *load* in the class text.

→ Inheritance: chapter 6.

The Creators part, beginning with **create**, lists the procedures which clients may use to create direct instances of the class. Here there are two: *make* and *from_tree*. A client may create a direct instance of *HASH_TABLE* by executing a creation instruction (also using the keyword **create**) such as

→ *Creation: chapter 20.*

```
create ownership_record.make (80_000)
```

which will allocate a new table with room for eighty thousand items.

A Converters part lists some of the creation procedures as being also conversion procedures, allowing assignment from instances of other types. Here it specifies as creation procedure *from_tree*, taking a *BINARY_TREE* as argument; this permits, for *h* a hash table and *b* a binary tree, to abbreviate the creation instruction

→ *Conversion: chapter 15.*

```
create h.from_tree (b)
```

as just

```
h := b
```

The Features part introduces the features of the class. It is made of zero or more subparts, each called a Feature_clause and introduced by the keyword **feature**. There are two reasons for allowing more than one Feature_clause:

→ *Features: chapter 5.*

- It is part of the recommended style practice to group features into categories. This yields a good class structure, facilitating understanding and maintenance. The EiffelBase libraries define a number of feature clause headers, each with a standard header comment; they include the ones used in the example: Initialization, Access, Status report, Removal.
- Each may define an export status, making the corresponding features public, secret, or available to specific clients. In the absence of such a specification the default status is public availability.

→ *"GROUPING FEATURES", 34.5, page 901.*

Here no Feature_clause departs from the default so that all the features shown — the procedures *make*, *remove* and *load*, the function *ok*, the variable attribute *control* and the constant attribute *Max_control* — are available to all clients. Calls from clients will use dot notation, as in

```
ownership_record.remove ("1745 BB 75")
        --Assuming a Variable entity status of type INTEGER:
status := ownership_record.control
ownership_record.make (10_000)
```

The last of these calls applies to *make*, which is also a creation procedure but here is just used as a normal exported procedure. (Compare this call instruction with the Creation instruction above, using the keyword **create**.)

→ A feature is "exported" if it is available to all clients. See definition on page 206.

Of course, when deciding to export *make*, the designer of *HASH_TABLE* should make sure that calls occurring after the initial Creation instruction will have the proper effect; this probably means using a new size which is greater than or equal to the original one (in other words, keeping the original if the argument to the call is smaller), and writing the routine so that resizing does not lose any of the previously inserted elements.

To ensure that *make* is not available for outside calls, it would suffice to add a Feature_clause with an empty Clients list, beginning with **feature** { }, and move the declaration of *make* there. This is explained in detail in the chapters on features and exports.

→ "Restricting exports", page 197. A full example appears in 5.5, page 134.

The Invariant part, beginning with **invariant**, introduces consistency conditions on the features of the class; here the condition simply gives the bounds for attribute *control*.

→ "CLASS INVARIANTS", 9.8, page 240.

Finally you may have a new Notes clause, complementing the one at the beginning of the class, and introducing note information of a more specialized nature, such as copyright, revision history and author name.

After this general survey of the structure of a class text, the rest of this chapter examine five clauses which apply to the class as a whole: Notes, Class_header, Formal_generics, Obsolete and ending comment.

4.8 ANNOTATING A CLASS

Through a Notes entry you may include documentary information in the text of a class.

This is particularly important in the approach to software construction promoted by Eiffel, based on libraries of reusable classes: the designer of a class should help future users find out about the availability of classes fulfilling particular needs.

We may imagine the author of a class *DOCUMENT* writing the class text as follows:

```
note
    description: "Documents of the most general form"
    domains: text, text_processing, FrameMaker
class DOCUMENT inherit … feature
    …
note
    author: "Tatiana Sergeevna Krasnojivotnaya"
    approved_by: "Giovanni Giacomo della Gambagialla"
    original: 21, March, 1999
    last: 12, July, 2006
end
```

The general form is:

Notes

Notes ≜ **note** Note_list
Note_list ≜ {Note_entry ";" ...}*
Note_entry ≜ Note_name Note_values
Note_name ≜ Identifier ":"
Note_values ≜ {Note_item ","...}$^+$
Note_item ≜ Identifier | Manifest_constant

Notes parts (there may be up to two, one at the beginning and one at the end) have no effect on the execution semantics of the class. They serve to associate information with the class, for use in particular by tools for configuration management, documentation, cataloging, archival, and for retrieving classes based on their properties.

Each Note_entry starts with a Note_name, such as *author:*, terminated by a colon. The rest of the Note_entry is a list of Note_item terms, each of which is an Identifier (such as *text_processing* or *July*) or a Manifest_constant, that is to say a value of a basic type, such as the integer *21*, or a string such as "*Tatiana Sergeevna Krasnojivotnaya*" etc.

→Manifest_constant *is introduced in 32.16, page 889, and subsequent sections.*

By the very nature of Notes parts, the choice of indices and values is free. Using consistent conventions will greatly facilitate the successful retrieval of reusable classes. Here you may wish to rely on the set of guidelines defined for the Eiffel Software Libraries.

→*"GUIDELINES FOR ANNOTATING CLASSES", 34.13, page 911.*

As illustrated by both the *HASH_TABLE* and *DOCUMENT* examples, a class may include up to two Notes clauses, one at the very beginning, before the keyword **class**, and one at the very end, before **end**. Their intended role is complementary:

- Use the initial Notes for critical information that you want every reader of the class to discover before reading anything else about the class, such as the *description* entry which succinctly explains the role of the class.

- Use the final Notes for archival and management information such as revision history, copyright and intellectual property notices, author and reviser names, and any supplementary information that will be useful to maintainers of the class.

The Notes parts of *HASH_TABLE*, shown earlier, illustrated these guidelines.

Notes semantics

A Notes part has no effect on system execution.

4.9 CLASS HEADER

The Class_header introduces the name of the class; it also serves to indicate whether the class is deferred or expanded. Here are two Class_header examples from EiffelBase and one from the Kernel Library, illustrating these possibilities:

class *LINKED_LIST*
deferred class *SEQUENCE*
expanded class *INTEGER*

The general form of the Class_header is simply:

Class headers

Class_header ≜ [Header_mark] **class** Class_name

Header_mark ≜ **deferred** | **expanded** | **frozen**

The Class_name part gives the name of the class. The recommended convention (here and in any context where a class text refers to a class name) is the upper name.

The upper name is the name written all in upper case..

The keyword **class** may optionally be preceded by one of the keywords **deferred**, **expanded**, **frozen** and **separate**, corresponding to variants of the basic notion of class:

- A **deferred** class describes an incompletely implemented abstraction, which descendants will use as a basis for further refinement.
- Declaring a class as **expanded** indicates that entities declared of the corresponding type will denote objects rather than references to objects
- A **frozen** class cannot be inherited from.
- A **separate** class, useful in concurrent programming, describes objects handled by a separate thread of control.

As the syntax specification indicates, these four options are exclusive. A class may not, for example, be both deferred and expanded; in fact, all non-expanded classes are considered to be reference classes.

This is part of a general characteristic of the syntax: unlike languages such as Ada, Java and C++, Eiffel does not use multiple successive keyword qualifiers. Where it allows you to write **property1** *x* or **property2** *x*, it does not permit **property1 property2** *x*. This keeps things simple and easy to remember.

The first two cases have an influence on the validity rule for Class_header and we now examine them in more detail.

Deferred classes

A class declared **deferred** describes an incompletely implemented abstraction, with the expectation that proper descendants of the class will provide or refine the implementation. This is useful to cover incompletely understood concepts or groups of related concepts. A typical example in EiffelBase is the deferred class *SEQUENCE*, which describes sequential data structures without prescribing any particular implementation. Proper descendants of this class, such as *LINKED_LIST*, describe concrete sequential structures. Such non-deferred classes are said to be **effective**.

→ *For details see "DEFERRED FEATURES", 10.11, page 266.*

The deferred-effective distinction applies not just to classes but to their individual *features*: a feature is deferred if its class specifies it (often with a contract: precondition and postcondition) but does not provide an implementation. In general, a deferred class includes one or more deferred features. For example procedure *extend*, which adds an element at the end of a sequence, is deferred in *SEQUENCE* and **effected** (made effective) by *LINKED_LIST* and other effective descendants, each in its own way.

Deferred classes have no direct instances (you may not create an instance of the corresponding type, as in **create** *x* for *x* of type *SEQUENCE* [*T*]); only their effective descendants do, so that **create** *x* is valid for *x* of type *LINKED_LIST* [*T*].

A less drastic way of restricting clients' instantiation rights is through the Creators part.

→ *"RESTRICTING CREATION AVAILABILITY", 20.7, page 531.*

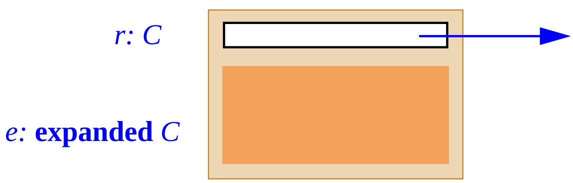
The validity rule below requires that as soon as a class has at least one deferred feature you must declare it as class as **deferred class**. If not, the class would be considered effective; then clients could create instances, and call on them a feature that you haven't implemented.

There is no converse requirement: you may declare a class as **deferred** even if it has no deferred feature. This is a way of stating that you intend to use a class as an abstract concept even though you haven't included any deferred feature yet. In particular, you are prohibiting clients from creating direct instances through **create** *x* instructions.

Expanded classes

Declaring a class *C* as **expanded** changes the assignment and comparison semantics of the entities declared of the corresponding types. With *y: C* (ignoring any generic parameters), and *C* not expanded, the assignment *x := y* is a reference assignment, and the boolean expression *x = y* compares references. But if *C* is expanded, the assignment copies the object denoted by *y*, and the test compares objects.

One application of this notion is to represent the notion of *sub-object*:



Sub-object vs. reference to another object

The figure shows an instance of a class with two attributes, one of a reference type and the other of an expanded type, representing a sub-object. The discussion of types will provide more details on the difference between expanded and reference semantics.

→ "EXPANDED TYPES", 11.9, page 327.

To declare a class as expanded you must make sure that it retains *default_create* — the default initialization procedure coming, after possible renaming or redefinition, from the universal class *ANY* — as one of its creation procedures. The reason is that initializing an object with sub-objects, such as the one illustrated above, requires initializing all its sub-objects, for which all that's available is the standard initialization.

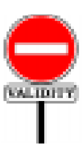
→ "OMITTING THE CREATION PROCEDURE", 20.4, page 519.

In the simplest case this requirement is automatically met: a class that doesn't have a Creators part (that is to say, doesn't explicitly list creation procedures) is considered to have *default_create* as its sole creation procedure. The details appear in the discussion of creation.

→ 20.4, page 519.

Validity of a class header

The validity rule on Class_header states the relationship between the actual class text and a declaration as **deferred**:



Class Header rule *VCCH*

A Class_header appearing in the text of a class *C* is valid if and only if has either no deferred feature or a Header_mark of the **deferred** form.

If a class has at least one deferred feature, either introduced as deferred in the class itself, or inherited as deferred and not "effected" (redeclared in non-deferred form), then its declaration must start not just with **class** but with **deferred class**.

The definition of "deferred class" is on page 303.

There is no particular rule on the other possible markers, **expanded** and **frozen**, for a Class_header. Expanded classes often make the procedure *default_create* available for creation, but this is not a requirement since the corresponding entities may be initialized in other ways; they follow the same rules as other "attached" entities.

The Class Header rule yields a simple definition:

DEFINITION

Deferred class, effective class

A class is **deferred** if its Class_header is of the **deferred** form. It is **effective** otherwise.

Any class that has at least one deferred feature is deferred; any class that only has effective features is effective *except* if the class is explicitly declared as **deferred class**.

4.10 FORMAL GENERIC PARAMETERS

A class whose Class_header is followed by a Formal_generics part, as in

```
class HASH_TABLE [G, KEY –> HASHABLE]…
```

will be called a **generic class**. (If the Formal_generics part is absent, the class is, predictably, a **non-generic class**.) A generic class has one or more **formal generic parameters**, which are identifiers, here *G* and *KEY*, not conflicting with any name of a class in the surrounding universe. The mechanism that permits generic classes and the corresponding types is called **genericity**.

As noted, a generic class does not directly yield a type, although it is easy to derive a type from it: just provide a list of types, called **actual generic parameters**, one for each formal generic parameter. This was done above in the declaration of *ownership_record* to derive the type

```
HASH_TABLE [CAR, STRING]
```

from *HASH_TABLE*, with an Actual_generics list made of the types *CAR* and *STRING*. Such a type is said to be **generically derived.**

Genericity is the main reason classes and types are not identical notions: while any non-generic class is also a type, a generic class such as *HASH_TABLE* needs actual generic parameters to yield types such as the above. The notions of class and type are, of course, closely connected. More precisely, any type has a **base class** whose features provide the operations available on the type's instances; for a generically derived type such as the above, the base class is simply the type stripped of its actual generic parameters, here *HASH_TABLE*.

→ *"BASE CLASS, BASE TYPE AND TYPE SEMANTICS", 11.7, page 324.*

A whole chapter is devoted to genericity and will give the details. Here is a is a preview of the syntax of Formal_generics parts:

→ *Chapter 12; see syntax and validity in "GENERICCLASSES", 12.2, page 341.*

Formal_generics ≜ "["Formal_generic_list"]"

Formal_generic_list ≜ [Formal_generic ";" …]

Formal_generic ≜ [**frozen**] Formal_generic_name [Constraint]

The Constraint construct, also detailed in the genericity chapter, governs *constrained genericity*, as in *C* [*G* –> *CONSTRAINING_TYPE*], which specifies that *G* represents not arbitrary types, as in the basic (unconstrained) case, but types that conform to *CONSTRAINING_TYPE*.

→ *"CONSTRAINED GENERICITY", 12.6, page 346.*

4.11 OBSOLETE MARK

By specifying an Obsolete mark for a class, you indicate that the class does not meet your current standards, and you advise developers against continuing to use it as supplier or parent; but you avoid harming existing systems that may rely on this class.

The decision to make an entire class obsolete is not a frequent one in well-planned software development: through information hiding, uniform access, dynamic binding and genericity, the language often enables developers to change a class with little or no impact on clients and descendants. Even when some aspects of a class become obsolete, the class as a whole may remain appropriate; this is why you should usually prefer the related mechanism letting you make individual **features** obsolete. The next chapter explains how to do this, with further comments about software evolution and obsolescence.

→ *"OBSOLETE FEATURES", 5.21, page 163.*

The decision to make a *class* obsolete is appropriate when you realize that even by obsoleting some of its features you won't be able to bring it up to its ideal form without disturbing existing software, and decide to replace it by a new version. The civilized way to do this is to keep the old class, at least for a while, under its original name, but mark it obsolete; this signals to client and descendant developers that they will ultimately have to adapt their classes to the new version.

An Obsolete mark has no other effect; in particular it has no bearing on the software's execution.

Here is the syntax of the mark, which comes after the Class_header and optional Formal_generics:

Obsolete marks

Obsolete $\triangleq$ **obsolete** Message

Message $\triangleq$ Manifest_string

There is no validity constraint. The semantic specification covers both obsolete classes and obsolete features:

Obsolete semantics

Specifying an an Obsolete mark for a class or feature has no run-time effect.

When encountering such a mark, language processing tools may issue a report, citing the obsolescence Message and advising software authors to replace the class or feature by a newer version.

Class obsolescence is not a way to cover up for bugs or flawed designs. If you realize that a class is incorrect or inadequate, you should face the consequences and repair the problem, even if this requires updating dependent classes. Any existing system using the flawed class cannot be functioning properly anyway. The Obsolete facility is meant for a different case: classes which were useful and sound, but cover needs for which you have now found improved solutions, based on a new design not backward-compatible with the original.