Object-Oriented Software Engineering with Eiffel

Jean-Marc Jézéquel

ISBN 0-201-63381-7

Warning: The page layout here is different from the original book

To Chantal, Gwenaëlle, Nolwenn, and Erwan.
To my parents, who helped me buy my first computer in 1980.

Contents

Preface

This is a book on software engineering the Eiffel's way.

Born in Dijon (France), Gustave Eiffel (1832–1923) graduated from the prestigious *École Centrale* in Paris. He first worked as an engineer for a railroad construction company before becoming the *Nepveu*'s company chairman and starting an office of studies dedicated to metallic construction. Using light steel modular structures instead of the usual design with cast iron, Eiffel built tall infrastructures featuring very good aerodynamic resistance. He built several viaducts, most notably at Bordeaux (1858) and Gabarit (1884). He also created the framework of the *Bon Marché* department store (1876) in Paris. Abroad he oversaw several projects in Austria, Switzerland, Hungary (Pest Railway Station, 1876), and Portugal (the Maria-Pia Bridge near Porto, 1877).

His most famous structures were the framework of Bartholdi's *Liberty Statue* in New York and the 300-meter *Eiffel Tower*, built for the 1889 universal exposition in Paris. These two world-famous landmarks were also technological marvels for that time. They opened the way for the new domain of industrial architecture. After 1890, Eiffel resigned from his business to concentrate on aerodynamic studies from the top of the Eiffel Tower. Today, more than one century after their construction, most of Eiffel's buildings are still standing and open for business.

In the software engineering domain, Eiffel is also the name of an object-oriented language that emphasizes the design and construction of large, high-quality softwares by assembling reusable software components, called *classes*, that serve as templates to make objects. Beyond classes (on which modularity is based), Eiffel offers multiple inheritance, polymorphism, static typing and dynamic binding, genericity, garbage collection, a disciplined exception mechanism, and systematic use of assertions to improve software correctness in the context of *programming by contract*.

Software engineering encompasses much more than what a computer language can offer. Computer languages are just tools that software engineers can use (or misuse) within a larger context. The Eiffel language is a

tool that has been specially designed in the context of software engineering. This book describes the tool, and provides clues on how to use it.

Chapter 1 is an introduction presenting the object-oriented approach within the context of software engineering. The main body of the book is then divided roughly into two parts.

The first part of this book presents the language itself. Chapter 2 presents the basic (procedural) elements of the language: what an Eiffel program is, what the instruction set is, and how to declare and use entities (variables) and routines. Chapter 3 introduces the concepts underlying the object-oriented approach: modularity, inheritance, and dynamic binding, and illustrates them in a small case study from the management information system domain. Eiffel programs do not exist in a void, so Chapter 4 brings in environment matters: system configuration, interfacing with external software, and garbage collection. Chapter 5 closes the Eiffel presentation with more advanced issues involving exception handling, repeated inheritance, typing problems, and parallelism.

The second part of this book addresses some Eiffel software development issues. In Chapter 6, we outline how an object-oriented software engineering process may make the best use of Eiffel, concentrating on specific guidelines to facilitate the translation OOAD to a maintainable Eiffel implementation. This process is illustrated by a rather large case study from the telecommunications domain. As a logical continuation of this study, Chapter 7 addresses verification and validation (V&V) issues of Eiffel software systems built in a software engineering context. Building reusable libraries discussed in Chapter 8, which presents three competing Eiffel data structure libraries. Finally, Chapter 9 shows how Eiffel can be used as an enabling technology to master a very complex problem: the building of a parallel linear algebra library that allows an application programmer to use distributed computing systems in a transparent way.

If you are lost at some point with the Eiffel-related vocabulary, there is a small glossary in Appendix ?? on page ??. An Eiffel syntax summary is presented in Appendix ?? on page ??, and a contact list closes this book (Appendix ?? on page ??).

Acknowledgments

This book would not exist in its present form without Bertrand Meyer (Interactive Software Engineering Inc. [ISE]), the designer of the Eiffel language. Rock Howard (Tower Technology Corp.) and Michael Schweitzer (SiG Computer Gmbh) also helped me by providing some input on their Eiffel products (compilers and libraries).

I would also like to thank the countless people involved in enlightening discussions on the **comp.lang.eiffel** Internet newsgroup, and particularly Richard Bielak, Roger Browne, Hank Etlinger, Jacob Gore, James McKim, Jean-Jacques Moreau, Erwan Moysan, and Michel Train, who read early versions of this book and gave me a lot of feedback as well as many pertinent suggestions.

My colleagues at Irisa deserve credit for relieving me of a share of my everyday workload, thus allowing me to complete this book in reasonable time. I have a special debt toward F. Guidec, who did most of the Paladin library design, and F. Guerber, who was the main contributor on the switched multi-megabits data service (SMDS) project.

Finally I would like to thank my editorial contact, Katie Duffy, for her constant support in making this book take shape.

Dr. Jean-Marc Jézéquel

I.R.I.S.A./C.N.R.S.
University of Rennes

Chapter 1

The Software Engineering Context

In this chapter we introduce the context of software engineering. Software construction and maintenance can benefit from an object-oriented approach. The Eiffel language, which has been designed specifically along this line, is introduced.

1.1 Introduction

1.1.1 What's the big deal about programming a computer?

Programming is easy. Nearly everybody can give the proper instructions to cook a dish or to record a movie on a VCR (though a good programming interface might be helpful). Only a handful of training hours are required for most people to learn how to write spreadsheet or BASIC programs. Even young children have little problem driving the Logo turtle back and forth on the screen.

There are few concepts, however, as widely admitted as the "software crisis." This expression was coined in the late 1960s when it appeared that most first releases of software products were notoriously buggy or late on delivery and hard to maintain. Today, the maintenance of a large software system is usually more costly than the total development phase, with a fair share due to bug corrections. The overall maintenance cost can even reach three or four times the initial cost for long life products.

What does this crisis mean? Are software engineers grossly overpaid

and incapable and should they be fired in favor of teenager programmers? This approach has been tried, but works in Hollywood movies only. As most software engineers know, the problem is actually twofold. At the micro level (also called programming in the small), we face the problem of designing and implementing correct algorithms. This activity is much like theorem proving because its complexity is of a mathematical nature. Still, it may be mastered by a single person who understands everything from top to bottom. At the macro level (also called programming in the large), we face the structural complexity of systems made of hundred of thousands or millions of lines of code and developed by large teams of programmers. This complexity management problem is not at all specific to software systems but is amplified by the well-known software "softness."

1.1.2 Programming in the small

The strange thing about computer science is that it has been invented *ex nihilo* to demonstrate an impossibility result in mathematics. Alan Turing built his famous mathematical model known as the *Turing machine* to prove that some properties can be undecidable (e.g., stopping the machine). An immediate and painful consequence is that computer programs cannot be proved correct in general, because *every (general purpose) programming language is formally equivalent to a Turing machine.* This may seem a remote problem arising only in very complicated cases. Consider, however, the following program, presented in [35]:

```
input a positive number n
while n is not equal to 1 do
  if n is even then n := n/2
  else n := 3*n+1
end
print "Terminated"
```

If you are not familiar with computer science foundations, you should try to craft a proof to get some insight into its mathematical aspects.

Can you formally prove that this program terminates for all positive input values?

Since the emergence of structured programming [7] in the late 1960s, the recommended way of dealing with this kind of problem has been to build software in such a way that it can be formally proved [9]. In spite of all the effort expended since then, proving techniques cannot be applied practically to real programs, because the complexity of the proof may be much greater than the program itself (and who is going to check the proof anyway?).

What is left is a general method of software production [35] that associates partly formal correctness arguments (called *assertions*) with the

1. **Initial** The software process is characterized as *ad hoc*, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
2. **Repeatable** Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3. **Defined** The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
4. **Managed** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5. **Optimized** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

Table 1.1: The SEI/CMM Levels

programs as they are being built. This method enables the construction of robust software modules, which can then be reused safely.

The problem of software correctness takes on a new dimension when parallelism is brought in. The complexity brought along by parallelism and its associated asynchronism is orthogonal to sequential complexity. Even when you restrict your programming language to have the power of a finite state automaton (FSA), once you put two FSA's to work in parallel and connect them through unbounded first in, first out (FIFO) channels, you may obtain the power of a Turing machine. Thus, even for very simple examples (two FSA's with no more than three states each in [20]), you may get infinitely complex behaviors. Here again, reusing carefully designed parallel software components appears to be a promising avenue toward mastering the inherent complexity of parallel systems.

Such a programming language has a fairly low power indeed: you can't even count *items with an FSA.*

1.1.3 Programming in the large

The size of software projects has increased by several orders of magnitudes since the 1960s to become widely out of the grasp of a single programmer. Software development is now a cooperative process. This problem is usually tackled along two different lines. On the one hand, the tools used

to develop software (programming languages and environments) are continuously improving to support this scale of complexity. On the other hand, a great deal of effort is devoted to improve the *process* by which software is developed. The best results are obtained when the tools and the process fit together well.

Some effort has been devoted to adapting the notion of *Total Quality Management* associated with manufacturing processes to the software industry. For instance, the ISO 9000-3 standard is an adaptation to the software world of the ISO 9001 general standard on quality assurance in the industry. Other examples of this trend are the variants of the ISO 9001 standard in the militaries of several countries, or the levels (Table 1.1) defined by the Software Engineering Institute (SEI) in its capability maturity model (CMM), a process-based quality management model for assessing the level of an organization's software development [17].

In this context the software development process is considered from an engineering point of view. It generally is divided into several subtasks, called *phases*. Each phase addresses different problems on the road leading from a set of requirements to a working software system. The output from each phase is the basis for the next. Although there is some dispute about their names and their boundaries, there is a broad agreement on the nature of these phases. In Europe, one popular approach is the V model, as illustrated in Figure 1.1 (there are several variants of the V model, such as the waterfall model [37] most popular in the U.S. or the spiral model [4]). The boxes represent the successive stages of a software development project, from the initial requirements down to the executable code (through analysis, design, and implementation), and then up to an operational system (through testing, integration and delivery). The dashed lines connecting the right-hand side boxes to the left-hand side boxes suggest a match between the requirements of the descending stages and the results of ascending stages. This V model is itself the first phase in the life-cycle of most large software systems, which usually endow several years of maintenance (both corrective and evolutive).

The study of this kind of process led to a new branch of computer science: software development methodology. This branch is the study of methods for designing and implementing software in a rational way. As with any other scientific domain, the increased complexity of a large system has been dealt with modularity, or a mechanism to break down problems to a manageable size.

A major breakthrough was made with the introduction of structured design [34, 36], which follows the spirit of structured programming and top-down functional design [44]. The structured-design family of methods provides a rational, systematic, and teachable process to go efficiently from

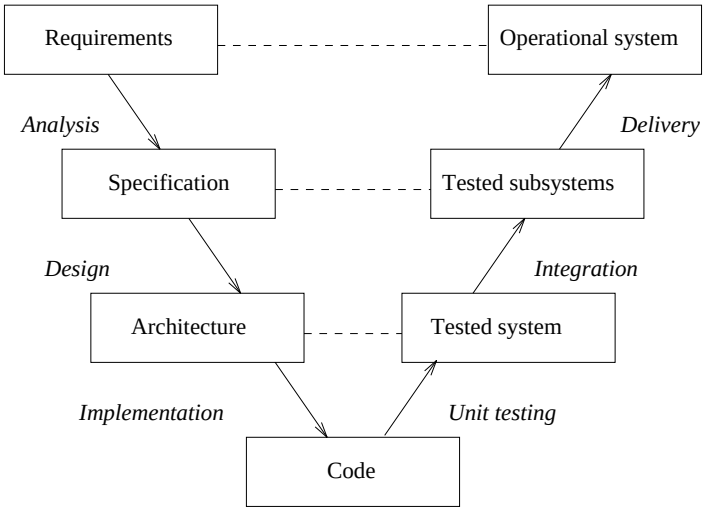


Figure 1.1: Phases of Software Development in the V Model

a well-defined functional specification to a working implementation. However, for large real systems, software specifications are usually imprecise, ambiguous, unclear, and much more subject to change than for other artifacts, because of the widely held belief in software "softness." When you build a house, you have to make up your mind about the disposition of the walls *before* putting on the roof. Is it possible for software systems to be relieved of that kind of constraint?

Jackson [19] showed that the main flaw of structured-design techniques is that they neglect this softness aspect of software construction. Because each module is produced to meet a precise sub-requirement, no provision is made for future evolutions, nor for dealing with potential analysis or design flaws. On the premise that *entities are more stable than functions,* Jackson's system development (JSD) method recommends that the programmer start the specification of a system with the elaboration of a "real-world" model representing the stable part of the system. This model is made of *entities* performing or suffering *actions*, the temporal pattern of which is precisely defined. Functionality specifications are added to this model at a later stage. This approach makes JSD a clear winner in terms of maintenance savings, but JSD suffers from a lack of structure and too much fuzziness.

This is not to speak of reusing software components, which lies completely out of the scope of these approaches.

Another popular approach, also based on modeling, consists of building entity relationship models of the problem domain [6]. The family of

methods that relies on this approach strongly emphasizes data and their organization. It is then very well suited to a relational database type of application, but may not fit so well with other problem domains.

Once the idea of analyzing a system through modeling has been accepted, there is little surprise that the object-oriented approach is brought in, because its roots lie in Simula-67, a language for simulation designed in the late 1960s, and simulation basically relies on modeling.

1.2 The Object-Oriented Approach

1.2.1 Origin

The object-oriented paradigm evolved from a set of concepts already present in computer science in the early 70s:

- *Classes* of objects used to simulate real-world application. In Simula-67 [8], an execution of a computer program is organized as a combined execution of a collection of objects. Objects sharing common behaviors are said to constitute a class.

- Protected resources in operating systems. Hoare [16] proposed the idea of using an enclosed area as a software unit and introduced the concept of a *monitor*, which is concerned with process synchronization and contention for resources among processes.

- Units of knowledge called *frames*, are used for knowledge representation. Minsky [31] proposed the notion of frames to capture the idea that a behavior goes with the entity whose behavior is being described. Thus, a frame can also be represented as an object.

- Data abstraction in programming languages such as CLU [12], which refers to a programming style in which instances of *abstract data types* (ADTs) are manipulated by operations that are exclusively encapsulated within a protected region.

1.2.2 Definitions in the Context of Software Engineering

Whereas the approaches described in Section 1.2.1 led to several paradigms in various computer science fields, the notions of *object, class, function*, and *inheritance* take a particular meaning in the context of software engineering.

Objects A computer science theoretician would define an *object* as the transitive closure of a function. More concretely, an object is an encapsulation of some state together with a defined set of operations on that state.

Even more concretely, an object takes up space in a computer memory, and has an associated address like a record in Ada, Pascal, or C. The arrangement of bits in an object's memory space determines that object's state.

An object embodies an abstraction characterized by an entity in the real world. Hence, it exists in time, it may have a changeable state, and it can be created and destroyed. An object has an identity (which is a distinguishing characteristic of an object) that denotes a separate existence from other objects. The object's behavior characterizes how an object acts and reacts in terms of changes in its state. In fact, each object could be viewed as a computer endowed with a memory and a central processing unit (CPU), and could provide a set of services.

Classes A *class* is a template description that specifies properties and behaviors for a set of similar objects. From the point of view of a strongly typed language, a class is a construct for implementing a user-defined type.

The term object *is sometimes used to refer to both class and instance, especially with languages like Smalltalk where a class is itself an object.*

Every object is an instance of only one class. A class may have no instances (usually termed an *abstract* or *deferred* class). Every class has a name and a body that defines the set of attributes and operations possessed by its instances. It is important to distinguish between an object and its class. In this book the term *class* is used to identify a category of objects and is a compile-time notion, whereas the term *object* is used to mean an instance of a class and exists at run time only.

Features Features of an object are either *attributes* or *routines*. They are part of the definition of classes. Attributes are named properties of an object and hold abstract states of each object. Routines characterize the behavior of an object, which is expressible in terms of the operations meaningful to that object. The routines are the only means for modifying the attributes of an object, hence the encapsulation properties of an object.

Attributes also may be viewed as parameterless functions.

An object may invoke routines or read the attributes that are part of another object's *interface*. Consider for example a radio set. Its interface is made of:

- Buttons allowing you to select between AM and FM,
- A frequency selector,
- A display showing the current radio station.

Usually, you don't need to know how the radio set is built to use it. The same should hold for software objects.

Inheritance The *inheritance* mechanism can be used to represent a relationship between classes. Every inheritance relationship has parents called the *super-classes* and children called the *sub-classes*, and attributes and routines inherited. Inheritance allows the definition and implementation of a new class by combination and specialization of existing ones. It is a mechanism for sharing commonalities (in terms of attributes and routines) between these classes, thus allowing classification, subtyping and reuse.

1.2.3 Object-Oriented Analysis and Design

Object-Oriented analysis and design (OOAD) methods subsume the best ideas found in previous methods. They still fit quite well in the V model, even if the software development process in most of these methods does not proceed linearly but swings back and forth between phases (seamless development). Numerous OOAD methods have been documented in the literature. A 1992 survey article by D. E. Monarchi and G. I. Puhr [32] mentions more than 20 OOAD techniques. Let's try to highlight their common rationale.

More details on OOAD will be given in the second part of this book, in Section ??.

The first step toward an object-oriented analysis is concerned with devising a precise, relevant, concise, understandable, and correct model of the real world. The purpose of object-oriented analysis is to model the problem domain so that it can be understood and serve as a stable basis in preparing the design step.

> *Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the domain. (G. Booch)*

The *design* phase starts with the output of the analysis phase and gradually shifts its emphasis from the application domain to the computation domain: the implementation strategy is defined, and trade-offs are made accordingly. Auxiliary classes may be introduced at this stage to deal with complex relationships or implementation-related matters. The output of the object-oriented design phase is a blueprint for the implementation in an object-oriented language, which is basically an extension of the design process.

Again the boundary between design and implementation is not rigid. This seamlessness of the object-oriented approach may upset the old-time

programmers who favor the well-established structured methods that feature strong frontiers between phases. A reality check might be necessary here: how often does a final product match its initial requirements? What is the situation 5 or 10 years later? Using the same conceptual framework (based on objects) during the whole software life cycle (from analysis to implementation, testing, delivery, and maintenance) yields considerable benefits in terms of flexibility and traceability. These properties translate to better quality software systems (fewer defects and delays) that are much easier to maintain because a requirement shift usually may be traced easily down to the (object-oriented) code (see for example [40]).

See B. W. Boehm and W. Humphrey's works [3, 17] for more thoughts on this topic.

1.3 Eiffel: An Object-Oriented Language for Software Engineering

1.3.1 A Software Engineering Tool

> *A good programming language is one that helps programmers write good programs. No programming language will prevent its users from writing bad programs (Kees Koster).*

As a software engineering tool, a computer language must address both issues of

- Fostering a rigorous approach based on formal assertions when programming in the small,
- Providing support for managing the structural complexity of programming in the large.

The Eiffel language has been designed specifically to meet these requirements. It is based on the principles of object-oriented design, and achieves a careful balance between the use of sophisticated concepts and overall simplicity, consistent minimalism, and pragmatism. It brings object-oriented design and programming closer together. It emphasizes the design and construction of large, high-quality software by assembling reusable software components made of classes.

Beyond classes (on which modularity is based), Eiffel offers multiple inheritance, polymorphism, static typing and dynamic binding, genericity, and a disciplined exception mechanism, and fosters a systematic use of assertions to improve software correctness in the context of *design by contract*.

Eiffel provides a consistent framework that fosters the design and implementation of software components that feature the following quality factors:

- **Correctness** is the ability of components to perform their tasks exactly, as defined by the requirements and specifications. Assertions available in Eiffel are elements of formal specifications that characterize the semantics of classes and their features independently of their implementation. They provide a reference against which correctness can be checked.

- **Reusability** allows software components to be used as building blocks for future software developments. Taking components created by others rather than creating new ones from scratch. The savings are not so much expected in the initial development phase (a good cut-and-paste editor could be as efficient) as in the testing, integration, and above all, maintenance phases of the software life-cycle. Inheritance plays a major role in increasing software reusability by allowing components to be customized.

- **Extensibility** permits new functionalities to be added easily with little modification to existing software systems. With this property, software systems can be extended easily to meet new requirements. This incremental development also relies on the inheritance mechanism. It is a fundamental part of object-oriented thinking.

- **Compatibility** is the ease with which software components may be combined and assembled to build useful programs. The Eiffel approach to building software can rely on the mere notion of assembling software components. All components are orthogonal: the concept of "main program" (or entry point) does not require special syntax, therefore a class that contains the "main program" for one system may be an ordinary class in some other system.

- **Robustness** is the ability of software components to function even in abnormal conditions. This ability boils down to avoiding catastrophic behavior when things go awry. Eiffel disciplined exception mechanisms play an important role in improving robustness.

- **Testability** is the ease of preparing validation suites for software components. In Eiffel, testability relies on the underlying paradigm of programming by contract.

- **Efficiency** is the good use of resources (e.g., processor or memory), to make good trade-offs among various strategies. The clean semantics of Eiffel enable sophisticated compiler optimizations that help produce efficient components.

- **Portability** is the ability to port software components across various software and hardware environments. It is often at odds with the notion of efficiency, but sometimes (as described in the case study of Section **??**) they can be reconciled.
- **Friendliness** is the ease of learning how to use software components. Good, up-to-date documentation as provided by Eiffel assertions and bound comments is of great help here.

1.3.2 Importance of a Language

> *Language shapes the way we think, and determines what we can think about. (B. Stroustrup)*

Some people still think that the technical differences between programming languages are irrelevant: a good design can accommodate any language. Clearly they are both right and wrong. Granted, a loop looks more or less the same whatever the language; and any design could be implemented with any language (remember, all languages are equivalent to a Turing machine). Few people still take seriously the old argument that method is everything, tools are nothing. If this was the case, we might as well still be writing everything in assembler. Languages are important as tools to best support the modeling paradigm. Where there are mismatches between the modeling paradigm and the tools, there must be manual translation by programmers. This is a costly and error-prone business [21].

Object-oriented languages are close to the design concepts used to deal with ever-evolving software systems. In the context of software engineering, Eiffel is one of the most consistent and well-designed object-oriented languages on the market. It is definitively not a "universal" language (e.g., for small, one-shot programs, simpler languages may be more appropriate) nor the "ultimate" language (there will be life after Eiffel). It provides the right paradigms to address the construction of large, long-lived object-oriented software systems while staying quite easy to master, though, so Eiffel is probably an important stage in the history of language evolution.

1.3.3 An Eiffel Overview

Eiffel is a pure object-oriented language: objects are the only things that can be manipulated at run time. It is not a superset or extension of any other language, although it retains the main lexical and syntactical conventions found in the ALGOL family of languages. Eiffel strongly fosters object-oriented programming and allows the programmer to avoid dangerous pitfalls from previous-generation languages. Still, Eiffel is an open

language that does interface easily with other languages such as C or FORTRAN.

Software texts in Eiffel are made of autonomous software units called *classes*. An executable software product (a *system*) is obtained by assembling a combination of one or more classes, one of which is called the *root* of the system.

A *class* defines a type, and is also the modularization unit. It describes a number of potential run-time objects, called its *instances*. A class is characterized by its features. A *feature* is either an attribute (present in each instance of the class) or a *routine* (describing a computation applicable to each instance of the class). A routine is either a function if it returns a result, or a procedure otherwise. A routine may have formal arguments; if so, calls to the routines must include the corresponding actual arguments (which are expressions having a type conforming to the formal argument).

Value semantics are also available to deal with simple objects like integers and characters.

An *entity* is a name in a class text. It is either an attribute of a class, a local variable or a formal argument of a routine, or the predefined entity *result* holding a function result. An entity stands for a value at run time. This value is normally a reference to an actual object, or may be *Void*.

Eiffel assertions are used for writing correct and robust software, debugging it, and documenting it automatically. Assertions include routine preconditions (which must be satisfied when the routine is called), routine postconditions (guaranteed to be true at the end of the routine), and class invariants (global consistency conditions applying to every instance of a class). Disciplined exception handling is used to recover gracefully from abnormal cases.

Eiffel classes may be generic, i.e., they may have a formal generic parameter such as T in LIST[T]. The class may use any type (class) as a generic parameter, thus making a flexible container structure such as LIST[INTEGER] or LIST[ANY_USEFUL_CLASS] easily available.

Eiffel classes may be structured in an inheritance hierarchy. Multiple inheritance is available with a set of mechanisms to manage it (renaming, selection, redefinition, undefinition, and repeated inheritance). Strict static typing is used for improving safety in a software system and dynamic binding is used for flexibility.

Entities may reference any object with a type that conforms to their declared type (hence their polymorphic nature). Dynamic binding of features to entities then ensures that the feature most directly adapted to the actual target object is selected.

1.3.4 Status of the Eiffel Language

Eiffel was created by Bertrand Meyer and developed by his company, Interactive Software Engineering Inc. (ISE) of Goleta, CA.

The definition of the Eiffel language [30] is in the public domain. This definition is controlled by the Nonprofit International Consortium for Eiffel (NICE). Thus, anyone or any company may create a compiler or interpreter having to do with Eiffel. NICE reserves the right to validate that any such tool conforms to the current definition of the Eiffel language before it can be distributed with the Eiffel trademark. (e.g., advertised as an "Eiffel" compiler).

NICE directions are given in Appendix ?? on page ??.

There are at least four Eiffel compilers (see Appendix ?? on page ??). These compilers should be compatible to a large extend now that NICE has published *The Eiffel Standard Library Vintage 95* (described in Section 4.3). Note that various versions of these compilers are available for free (see Appendix ?? on page ??).

Part I

Language Elements

Chapter 2

Basic Language Elements of Eiffel

The basic constructions of the Eiffel language are the system, the class and its components, the imperative instruction set, and the assertions.

2.1 The Eiffel Notion of Systems

2.1.1 System and Program

Eiffel is a language that focuses on *software components*, not on *programs*. The *class* is the top construct of the Eiffel grammar. The notion of program found in most computer languages is downplayed in Eiffel.

Building programs with Eiffel consists of *assembling* on-the-shelf and ad hoc software components, or classes. For that you have to tell the compiler (or another Eiffel environment tool, such as an interpreter) where the relevant classes are, that a particular class among them is the "root" of the Eiffel program, and that this program entry point —like the *main()* function in C— is a particular creation routine of the root class.

ACE files are described in depth in Section 4.1.1 on page 123.

The exact way of doing this assembly depends on your Eiffel environment. It could be through command-line arguments. Most probably it is through a configuration file, called an Assembly of Classes in Eiffel (ACE). An ACE file is remotely related to the well-known *makefile* found in UNIX or C environments, the main difference is that you don't have to deal with dependency rules nor imperative instructions. An ACE file only contains information on the root class of a system, the executable(s) name, the com-

pilation options, and where to find the other classes needed by the system. In Eiffel, it is the job of the compiler to determine dependencies and to decide everything about the compilation and linking process.

2.1.2 "Hello, world!"

A creation procedure corresponds to the class constructor in the C++ terminology.

``Hello, world'' is the unavoidable example of one of the smallest Eiffel programs. The class listed in Example 2.1 describes objects that are only able to print "Hello, world" when they are created. The feature *make* is declared to be a creation procedure; i.e., *make* is called to create a HELLO object.

Example 2.1

```
-- A simple "Hello world" example

class HELLO
creation
    make                                                    5
feature
    make is
            -- class entry point
        do
            print("Hello world!%N")                         10
        end
end -- HELLO
```

%N is the new line character, equivalent to the \n found in C.

Once the Eiffel system with HELLO as its root class and *make* as its entry point has been compiled to an executable program called **hello,** it can be run as any other program.

The execution model of Eiffel is very simple: when a program is run, a single instance of the root class (here HELLO) is created (the root object), and the specified creation procedure is called. A creation procedure can be designed to create other objects (which may themselves create other objects). and do real work with them. In our simple example, the creation procedure prints the expected "Hello world" message and exits. The program then finishes.

2.2 Class = Module = Type

2.2.1 Foundation Principles

> *There is a magic number: seven plus or minus two. This refers to the number of concepts that we humans can keep in mind at any one time. (H. A. Miller)*

The fewer the number of concepts in a programming language, the easier it is to learn it and to understand programs written with it. In Smalltalk, for example, everything is an object; this brings conceptual simplicity and frees space in the programmer's mind to let him or her concentrate on useful things.

The Eiffel way of liberating our minds is through the unification of the notions of module and type in the language construct called *class*. Then an object is just an instance of a class, just as you are an instance of the *homo sapiens sapiens* species. Simplifications often occurs at the cost of some limitations or more complexity somewhere else. In Smalltalk, "everything is an object" leads to a complex structure of meta-classes [13, 5]. Some consequences of the Eiffel unification are discussed in Section **??**.

Both notions of type and module existed for years, but Eiffel was the first computer language in which they were fully unified. Still, like a Janus statue, a class has two faces. Let's explore them separately.

2.2.2 The Class as a Module

Modularity helps engineers (and others) manage the complexity of systems. The principle of modularity is the key to support modifiability, reusability, extensibility, and understandability. A module is characterized by a well-defined interface and by information hiding. An interface should be small and simple in order for modules to be as loosely coupled as possible.

Modularity rapidly made its way into computer science through the notion of subroutine (procedure). The next step was to consider coarser grain modularity. At the program unit level, a set of data and procedures are encapsulated in a programming module, which can be compiled separately. This idea is already present in C, where a module is just a file.

Information hiding is implemented in C with the "static" declaration occurring for a variable or a function: it makes this variable or function private to the file.

The next step was to make the notion of module part of the language. This is achieved in Modula-2 [45] (hence the name) and Ada83 [18]. In Modula-2, modules are split into specification (interface) and implementation parts. Each of these module components is compiled separately. Data types and procedures may be specified in the interface of the module without revealing their representation details. These are provided in the implementation part of the module. Any other module can only have access

to the interface of the module. In Ada a module is called a "package." It is essentially the same notion as that of Modula-2's, including a separate interface specification.

The Eiffel notion of class evolved from these notions of modules. You cannot define various types, however, but just one, which is identified to the class itself. As a consequence, the interface of an Eiffel class, although not described separately from the implementation part, still exists conceptually. An Eiffel class may even have several interfaces. This property is called *subjectivity*: the way you see a class depends on who you are (this is explained in depth in Section 3.1.4).

Producing interface specifications of Eiffel classes is usually done by a tool present in most Eiffel environments. Usually called *short*, this tool eliminates the need to maintain the consistency between the interface specification and the module implementation, and still retains the benefits of having an interface specification without implementation details.

2.2.3 The Class as a Type

The notion of user-definable *data type* was already present in the ALGOL family of languages [33], and has not evolved much since then. Formally, a type characterizes:

- A domain of values,
- A set of operations applicable to objects of the named type.

In a typed language, objects of a given type may take only those values that are appropriate to the type, and the only operations that may be applied to an object are those that are defined for its type. A typing error results when one of these conditions is violated.

Depending on the moment when this typing error is detected, we can classify languages into three categories:

- Untyped (or loosely) typed language. A typing error may remain undetected until after run time. The usual error message is "*Bus Error. Core Dump*" (your mileage may vary).
- Dynamically typed language. The typing error is detected at run time, and in Smalltalk it can result in the well-known "*Message not understood*" error message (meaning an operation has been invoked on an object that didn't define it).
- Statically typed language. A program is rejected by the compiler as soon as it may contain a typing error. Eiffel is such a language. When

a typing error is detected, the usual (compile time) error message is "*Operation xxx not defined on object yyy.*"

Each approach has pros and cons. To be brief, untyped languages are useful for low-level system programming or as target languages for a compiler, but 20 years of large software developments with C have demonstrated the limitations of this kind of languages.

Dynamic typing has made its way into software prototyping and interactive systems, and more generally into environments where the product is short-lived or requires very rapid turnaround. In such a context, where the software must be as soft as possible, static type checking tools are thought to be too cumbersome, whereas dynamically typed languages give the programmer the freedom needed to get things working very quickly, and to make changes at a rapid pace.

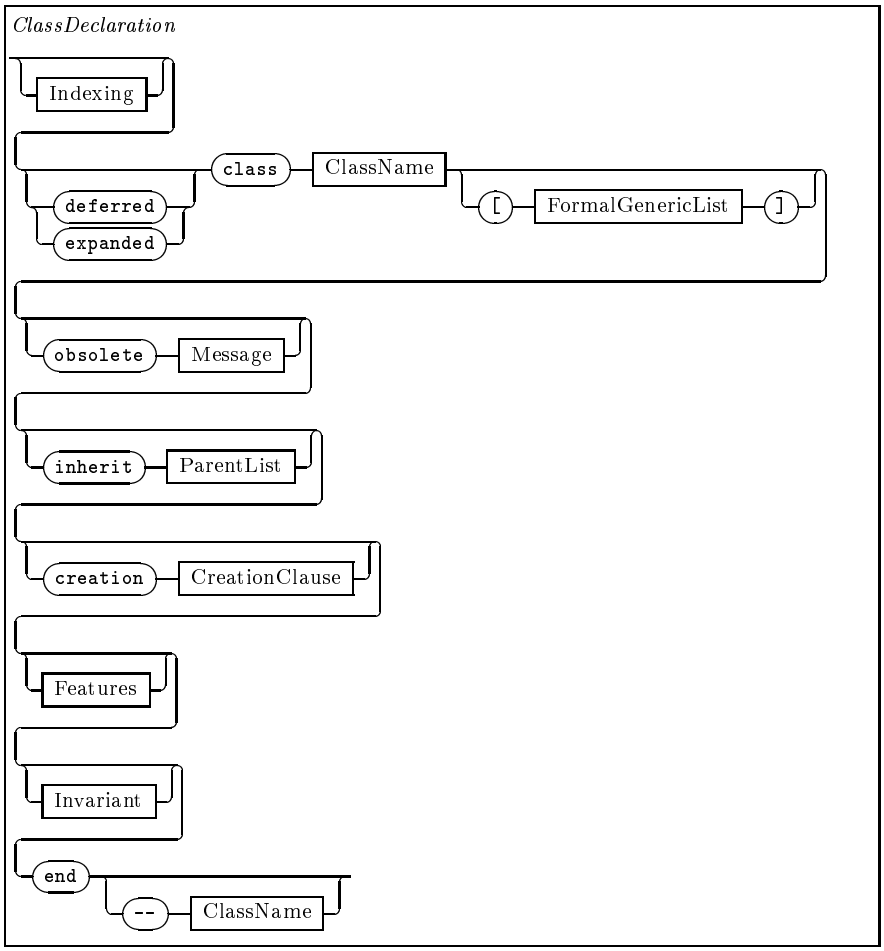
Finally, static type checking is a must if the product is supposed to have a long and active life (e.g., requires continuous upgrading, or serves as the source of many spin-off products), or if the software correctness is a prime concern—if only to save money in detecting defaults earlier in the software life cycle.

The Eiffel notion of class is actually based on the notion of abstract data type (ADT), a type in which the allowed operations have associated formal properties defining their semantics [25, 14]. The Eiffel language constructs corresponding to these formal property specifications are the *assertions*, taken from the ADT theory. Assertions will be described in more detail in Section 2.5.3 on page 57, but it was necessary to introduce them here, because they belong to the Eiffel notion of type. They play a major role in constraining subtyping through inheritance, so assertions should belong to every object-oriented language notion of type (as has been proved in [27]).

2.2.4 Components of a Class Declaration

Notation for Describing Eiffel Syntax

There are several ways to formally present the syntax of a programming language. The Backus-Naur form (BNF) is the most concise, but not the most readable. Thus, we present Eiffel's form with syntax diagrams that are essentially graphical representations of the BNF. Syntax diagrams are read from left to right. The lines may loop back on themselves, indicating that a construct may be repeated. A circle or ellipse denotes a literal string that appears exactly as stated. A rectangle surrounds a construct that is defined in another syntax diagram (this construct is called *nonterminal).* A full set of Eiffel syntax diagrams is provided for reference in appendix **??**.



Syntax Diagram 1: The class declaration

The syntax diagram 1 describes the syntactical components of a class declaration. It is made of the following parts:

Indexing clause, for documentation and indexing purposes. The usage of the indexing clause is described in Section ?? on page ??. Until then, you may consider this clause as a structured comment.

Class header, which allows regular, deferred, or expanded classes. De-

ferred classes are presented in Section 3.6 on page 105. Expanded classes are discussed along with the notion of entity in Section 2.3 on page 37.

Class name, which is also the module name and the type name.

Generic clause, made of a list of formal generic parameters between brackets. This clause makes it possible to build classes with parameters (see Section 3.2 on page 86).

Obsolete clause, which if present denotes that this class is to be eliminated from the library in future releases. More details are given in Section ?? on page ??.

Inherit clause, which allows you to specify how this class inherits from other ones (see Section 3.3 on page 90).

Creation clause, which specifies which routines may be called on creation of an instance of this class (see Section 3.1.1 on page 74).

Feature clauses, which describes the class features (attributes and routines) grouped by exportation sets (see Section 3.1 on page 73).

Invariant, which specifies the class invariants (see section 2.5.3 on page 57).

The minimal class declaration in Eiffel is made of the keywords **class** and **end** separated with the class name; all other clauses are optional. The class BOOK in example 2.2 reflects the Eiffel syntax more concretely. This class, which encapsulates a book description (basically a title, a list of authors, and an inventory number) is used in the case study of Section 3.8.

The Eiffel notion of class encompasses both notions of ADT implementation and module; that is, a program unit. In this chapter, we concentrate on the latter aspect of a class to present what is often called the imperative part of Eiffel, just considering one program unit (i.e., an isolated class), without *generic*, *obsolete*, or *inherit* clauses.

Lexical Components

On the lexical level, despite some unusual features (multi-line strings, expected comments, use of the % symbol as an escape character,see Appendix ??) Eiffel mostly conforms to usual ways of doing things in other software engineering-oriented languages. Eiffel is case-independent, and follows the usual conventions for the syntax of identifiers. As in Ada, comments are introduced with a double dash (– –) and end at the end of the line.

However, notation conventions exist. See Section ??.

Example 2.2

```
class BOOK
creation
    make
feature
    title : STRING                                                    5
    authors : STRING
    inventory : INTEGER
    make (new_title, new_authors:STRING; new_inventory:INTEGER) is
            -- make a new book record
        require                                                      10
            title_non_void: new_title /= Void
            authors_non_void: new_authors /= Void
        do
            title := new_title
            authors := new_authors                                   15
            inventory := new_inventory
        end -- make
    print_description is
        do
            print(title); print(", by ")                             20
            print(authors); print(" (")
            print(inventory); print(")%N")
        end -- print_description
end -- BOOK
```

Manifest Constants and Basic Types

A manifest constant is a literal value present in the text of the class. It has a type, deduced from the lexical structure, e.g., BOOLEAN, CHARACTER, INTEGER, REAL, BIT sequence, STRING, and ARRAY.

Usual conventions also apply here: **True** and **False** are the only BOOLEAN constants, an ASCII character enclosed in single quotes (e.g., 'a') denotes a CHARACTER constant, 42 is an INTEGER constant, and a sequence of character values enclosed in double quotes (e.g., "Hello, world") is a STRING constant.

Full details on the syntax for all manifest constants are given in appendix ?? on page ??.

A type is nothing but a class, so manifest constants are just constant instances of their classes (sometimes called *basic types*). Conversely, these classes are nothing special further than being able to have literal instances (which implies that they must be known by the compiler, which may then

make several optimizations with them, but this is another story). Along with a set of other Eiffel classes, these basic types must also be the same (or at least compatible) regardless of the Eiffel compiler; this is why they are sometimes called kernel library classes.

2.3 Definition of Entity Declaration

2.3.1 Entity Declaration

At run time, only objects exist. An entity is a language-level notion that allows the programmer to designate objects. The notion of entity encompasses the usual notions of variable, formal parameter, and result value found in most languages. Eiffel is a strongly typed language, so every entity used in an Eiffel program must be declared, or associated to a type. As in Pascal or Ada, an ALGOL-like syntax is used to declare entities (see Syntax Diagram 2).

Syntax Diagram 2: Entity declarations

In Example 2.3 an entity *title* is declared to be of type STRING, and the entity n is declared to be of type INTEGER.

Example 2.3

```
title : STRING
n : INTEGER
```

2.3.2 Entity Expansion Status

Eiffel expanded *entities correspond to C++* value *objects or to Smalltalk* immediate *objects.*

An entity may either *refer* to an object, or directly *hold* it, depending on whether its type is a reference class (e.g., INTEGER_REF) or an expanded class (e.g., INTEGER), as illustrated in Figure 2.1. For both cases in this example, `print(e)` would display 547.

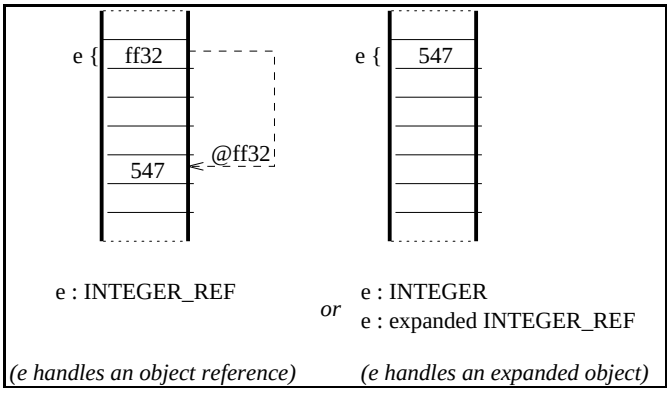


Figure 2.1: Reference vs. expanded objects

Whereas the natural semantics for entities are based on the notion of reference to an object, sometimes the expanded semantics may make more sense. This is the case when dealing with such basic notions of integer or real numbers, or when real-world modeling suggests it (a person has one head, not a reference to a potentially shared head). The choice of value *vs.* reference objects is simply a design decision based on the model one is trying to build.

On the contrary, C basic types such as int *still exist in C++ as a separate notion, unrelated to the class notion. This is why C++ is sometimes referred to as a hybrid language, whereas Eiffel or Smalltalk are called pure object-oriented languages.*

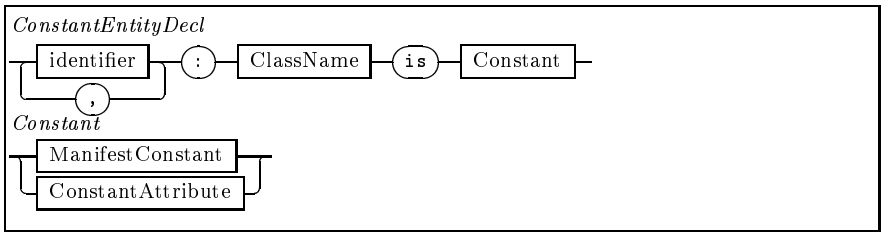
An object may indeed have several entities referring to it, whereas an expanded object is just the run-time value of an entity. Basic data types such as BOOLEAN, INTEGER, REAL, DOUBLE, and CHARACTER are just the expanded forms of BOOLEAN_REF, ... CHARACTER_REF and are nothing special with respect to the type system. The only thing that makes this set of classes (along with the classes STRING and ARRAY) somehow special is the possibility of declaring *manifest* constants of these types in a program text.

2.3.3 Constant Entities

Thus it does not need to be physically stored with the instance.

A *constant entity* is tied to a given object. Its value does not change at run time, and is the same for all instances of a class. The syntax of a constant entity declaration is presented in Syntax Diagram 3.

The constant entity may be tied to a manifest constant as described in Section **??**. Example 2.4 presents a set of constant entity declarations. BIT16 and BIT8 are conceptually different classes (as for any n in BITn).



Syntax Diagram 3: Constant entity declaration

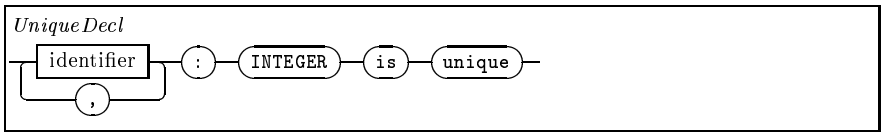
Example 2.4

```
i : INTEGER is 3
a_negative_number : INTEGER is −864322
a_huge_number : INTEGER is 3789641370
PI : DOUBLE is 3.14159265453
message : STRING is "This is a message string"
mask : BIT16 is 0101000011110101B
BCD13 : BIT8 is 00010011B
```

Unique Constants

This is related to in Pascal or Ada or enum *in C.*

Sometimes the actual value of an integer constant is not really important to the programmer: the only important thing is that each constant in a set has a unique value. To let the compiler choose a code for an integer constant, one may declare it as **unique** (Syntax Diagram 4).



Syntax Diagram 4: Unique constant declaration

In Example 2.5, the value of a **unique** constant such as **Red** is a positive integer. If two unique constants are introduced in the same class their values are guaranteed to be different. Furthermore, if they are declared to be in the same clause (as **Red**, **Green,** and **Blue** in our example), these

Example 2.5

```
Red, Green, Blue : INTEGER is unique
Yellow : INTEGER is unique
```

constants will have consecutive values.

Unique constants may seem more primitive than the usual enumerated types found in procedural languages. However, in these languages the main use for the enumerated types is to help implement variant records or to allow clever set operations. Both concepts are superseded by object-oriented techniques: inheritance and the use of sets of anything (instead of sets of enumerated data). Still, unique constants are useful when dealing with error codes, or finite state machine state encoding.

Other Constant Declaration

The last way to declare constants is through the use of *once* functions (described in Section 2.5.6 on page 62). Once functions allow for computed constants, and also for constants with types that cannot be expressed with the manifest constants.

2.3.4 Default Initialization Rule for Entities

The value of an entity is always defined in Eiffel. The initial value of an entity depends on its type, according to the rule described in Table 2.1 (the value *Void* denotes an empty reference).

Entity type	Initial value
BOOLEAN	**false**
CHARACTER	'%U' (NUL)
INTEGER	0
REAL	0.0
DOUBLE	0.0
reference to class A	**Void**
expanded class A	All attributes of A initialized to their default values

Table 2.1: Default initialization rule for entities

2.4 Statements

There are relatively few instructions in Eiffel. In this respect, it is almost minimal. All classic constructs usually found in an imperative language still exist in Eiffel, but in one instance only (e.g., there is no chance to choose among various loop constructs).

Eiffel instructions include the object creation, the assignment, the feature call, the sequence, the conditional, the multi-branch choice, and the loop. Also available are the debug instruction (to include optional debugging code) and the check instruction to check an assertion at any point in the code.

2.4.1 Assignment

The assignment instruction allows a new value to be given to a variable provided the value type conforms to the variable one (Syntax Diagram 5).

For now, consider that type conformance is type equality. Its exact definition is given in Section 3.3.3 on page 92.

Syntax Diagram 5: Assignment syntax

This assignment syntax is typical of the ALGOL family of languages. Consider the following assignment:

This assignment syntax is the same as in Pascal, Ada, or Modula-2.

```
target := source
```

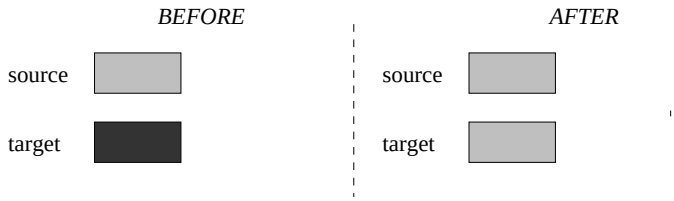
The exact effect of this instruction depends on whether the target and the source are expanded objects or references.

- If the target is an expanded type, then the source object is copied into the target (Figures 2.2 and 2.3. On this set of figures, the rectangles represent the source and target contents). However, if the source is Void the assignment will fail. When something *fails* in an Eiffel program, an exception is triggered (see Section ?? on page ??).

- If the target is a reference, and

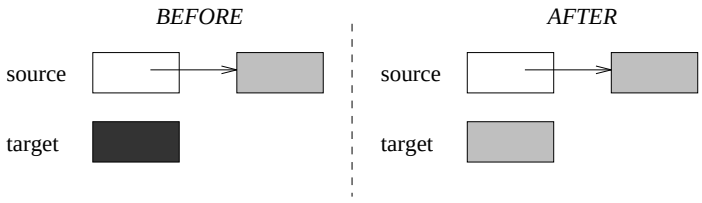
  – If the source is a reference, this reference is copied to the target, and thus both target and source refer to the source object after

Two entities are said to be aliased *if their values are references to the same object.*



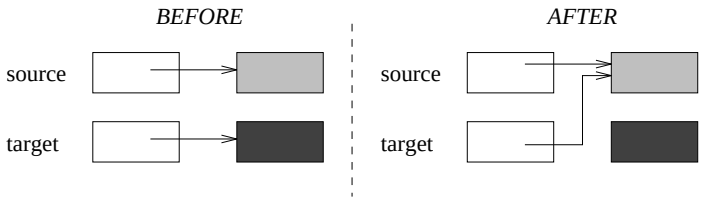

```
target := source
```

Figure 2.2: Assigning an expanded object to an expanded entity



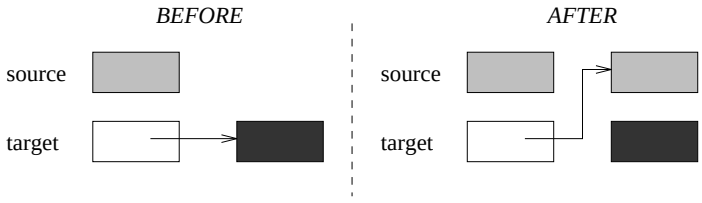

```
target := source
```

Figure 2.3: Assigning a reference object to an expanded entity




```
target := source
```

Figure 2.4: Assigning a reference object to a reference entity




```
target := source
```

Figure 2.5: Assigning an expanded object to a reference entity

the assignment (this is sometimes called *aliasing*). See Figure 2.4 for an example.

– If the source is an expanded object, it is cloned to a twin object, and the target then refers to this twin as illustrated in Figure 2.5.

You never need to worry about memory management issues with Eiffel unless you really insist on doing so.

In both cases where the source is a reference object, it may become unreachable (lost) after the assignment instruction. In Eiffel, it is the task of the *garbage collector* to recycle this kind of unreachable memory. Section 4.5 gives more detail on how it works.

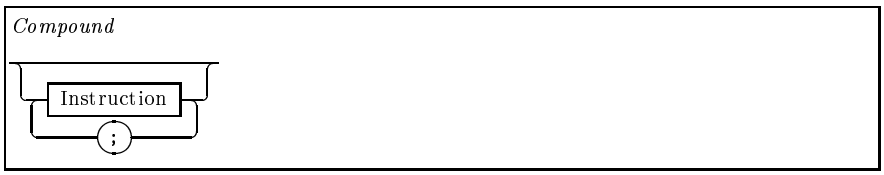
A variant of the *Assignment* instruction, is called the *Assignment Attempt* (denoted ?=). It is described in Section ?? on page ?? in the discussion on type conformance.

2.4.2 Testing for Equality

Related to assignment is the test for *equality*, which exists in three flavors in Eiffel:

- $a = b$ tests whether a and b refer to the same object (reference equality). If both are expanded entities, it tests for the equality of their values.
- *equal(a,b)* tests whether a and b are identical objects, that is all their fields a_i and b_i are such that $a_i = b_i$. For expanded entities, *equal(a,b)* has the same meaning as $a = b$.
- *deep_equal(a,b)* tests whether a and b have equal values if they are expanded entities, or refer to isomorphic object structures; that is, all their fields a_i and b_i are $deep\_equal(a_i, b_i)$.

2.4.3 Sequence



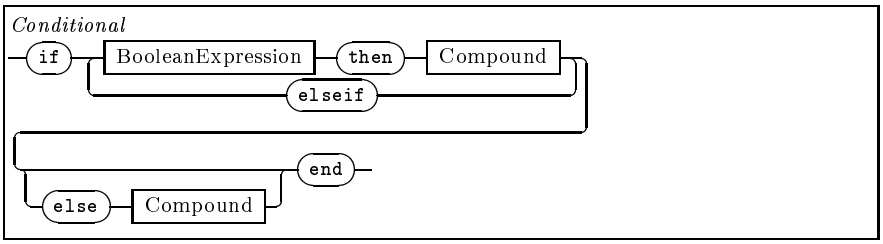
Syntax Diagram 6: The compound instruction

The sequence is the control structure denoting that a set of instructions (called a compound instruction) must be executed sequentially, in their

textual order. As illustrated in Syntax Diagram 6, the semicolon (;) is optional, and an empty sequence is a valid instruction.

2.4.4 Conditional

Conditional control structures allow the selection of one of a number of alternative sequences of statements, depending on the value of some condition. The syntax of an **if** statement is presented in Syntax Diagram 7.



Syntax Diagram 7: The conditional

This Eiffel conditional statement is very similar to Ada's, except that **elseif** *is used instead of* **elsif**.

Example 2.6

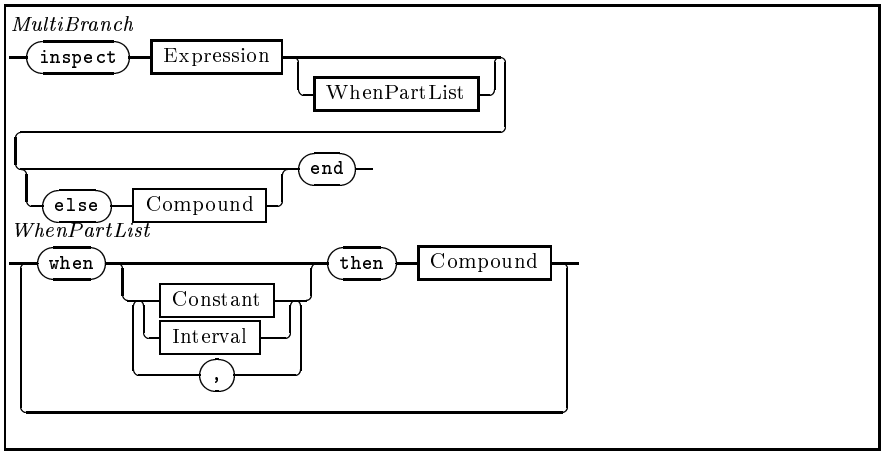
```
if last_read_value = 0 then
  print ("zero%N")
elseif last_read_value > 0 then
  print ("positive%N")
else                                                    5
  print ("negative%N")
end
```

This statement works as follows. The first condition is evaluated:

- If *True* then the first compound is executed and the flow of control passes to the instruction following the *end* clause.
- If *False* then the next condition is evaluated, and so on until the last condition;
- If all the conditions have been evaluated to *False,* then the compound following the *else* clause is executed (if it exists).

2.4.5 Multi-branch Choice

Like the *if* statement, the multi-branch choice statement selects one from many alternative sequences of statements (its syntax is described in Syntax Diagram 8).



Syntax Diagram 8: The Multi-branch choice

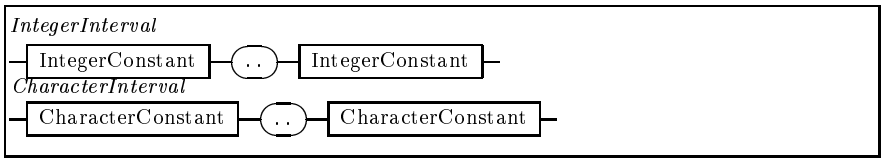
This selection is in the spirit of Pascal case or C switch constructs, and has exactly the same semantics as the Ada case statement: Ada's ⇒ *becomes Eiffel's* **then**, *and* **when others** *becomes* **else**.

The selection, however, is based on the value of the expression following the *inspect* clause (*inspect* expression). This value must be of the same type as the possible constant values listed in the clauses, where only INTEGER and CHARACTER constants (or intervals) are allowed.

These alternatives must be exhaustive and mutually exclusive. An *else* clause (as in the *if* statement) is available as a last alternative to cover all values not given in previous *when* clauses. Several alternatives may be declared for the same *when* clause, either by enumeration (alternatives are separated with a comma) or by range of values (the bounds of which are given and separated with two points ".."; see Syntax Diagram 9) or any combination of both.

The effect of this statement is that the compound associated with the clause matching the *inspect* expression is executed and the control is then passed to the instruction following the *end* clause.

The multi-branch choice may be the only redundant Eiffel statement. The *if* statement can always be used instead. The multi-branch choice was not present in earlier versions of the language, because in procedural languages such as C or Ada, the multi-branch choice statement is mainly used



Syntax Diagram 9: INTEGER and CHARACTER interval syntax

Example 2.7

```
inspect lastchar
  when 'a'..'z' then
    print ("lowercase letter")
  when 'A'..'Z' then                                    5
    print ("uppercase letter")
  when '0'..'9' then
    print ("digit")
  when '+','-','*','/', then
    print ("operator")                                  10
  else
    print ("other character")
end
```

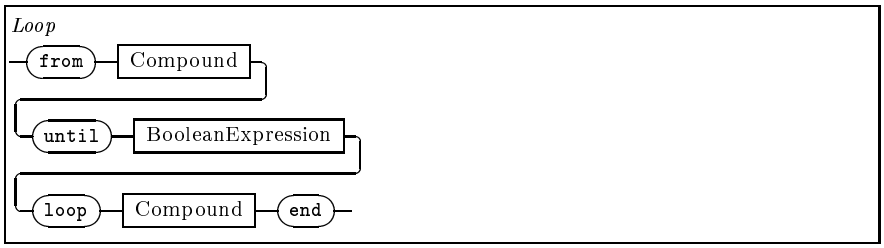
to select pieces of code related to variants of a data type (union in C, variant records in Pascal or Ada). This usage has proved to hamper modifiability (see for example [29]) because each time a component is added or removed to the variant record, every multi-branch choice dealing with the associated data structure must be changed. Object-oriented languages provide a much better solution to this problem through the use of inheritance (to replace variant records) and dynamic binding (to automatically select the relevant piece of code, see Section 3.5.2 on page 103).

The multi-branch choice was included in Eiffel to alleviate the syntax of dealing with input data (as in Example 2.7) while allowing the compiler to produce more efficient code.

As a general guideline, the *inspect* statement should be used mainly to discriminate among input data. Using it in another context could be an indication that the style is not really object oriented.

2.4.6 Iterative Control: The Loop

Iterative computation is a programming technique based on the repetitive application of the same processing. It is a fundamental concept of computer science that has been in computer languages from the beginning (in the form of a *goto* to a *label*).



Syntax Diagram 10: The basic syntax of the Eiffel loop construct

An iteration is made of three main parts:

1. The initialization part, to establish the initial state of the loop. In Eiffel, it is the compound following the keyword **from**;
2. The termination condition, or the Boolean expression specifying when the loop is considered finished. In Eiffel, the termination condition follows the keyword **until**;
3. The body of the loop; e.g., both the processing to be performed at each iteration and the progression code to advance toward the verification of the termination condition. In Eiffel, the body of the loop is the compound following the keyword **loop** and terminated with the keyword **end**.

In Example 2.8, we compute the quotient of a number n by a *divisor* (this operation is also known as an integer division). Provided n and *divisor* are positive integers, the result of this function must satisfy:

$$Result \times divisor \leq n < (Result + 1) \times divisor \tag{2.1}$$

The idea of this loop is to count how many times the *divisor* can be subtracted from n. The variable *remainder* is initialized to n, whereas *Result* is set to 0 through the default initialization rules presented in Table 2.1 on page 40. If the termination condition ($remainder < divisor$) does not hold,

then the loop body is executed (*remainder* becomes *remainder* minus *divisor* and *Result* becomes *Result + 1*). This loop body execution is repeated until the *remainder* eventually becomes smaller than the *divisor*.

Example 2.8

```
from remainder := n
until remainder < divisor
loop
    remainder := remainder − divisor
    Result := Result + 1                                    5
end −−loop
```

The same thing holds for recursion, which has the same expressive power as the loop.

The loop is a powerful construct. Imagine building programs without loops: you just lack expressive power. The drawback of the loop construct is that it is too powerful: once you have the (unbounded) loop in a language, you get the power of the Turing machine. That is, there are programs that you cannot prove correct (see the example in the introduction).

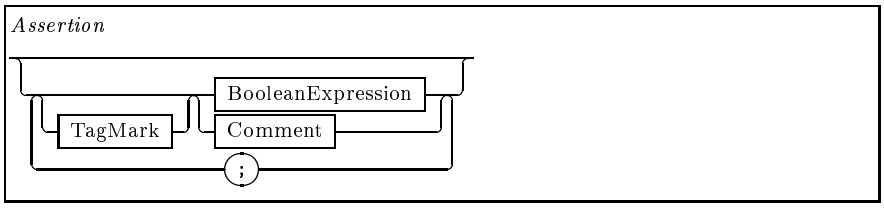
The computer science answer to this problem is to design each loop *in such a way that it can be proved correct*. Eiffel helps you to follow this line, by means of the notion of loop variants and invariants.

2.4.7 Designing Correct Loops with Loop Assertions

The Notion of a Loop Invariant

A loop invariant characterizes what the loop is trying to achieve, without describing how [10]. It is a Boolean expression that must be true on initialization of the loop variables, maintained with each iteration of the loop, and held to be true at the termination of the loop. For example, a loop invariant in Example 2.8 might be: $n = Result \times divisor + remainder$. In Eiffel, the loop invariant is an *assertion* that may be specified after the keyword **invariant** (see Syntax Diagram 12). The assertion itself is either a comment or a run-time checkable Boolean expression that may be tagged with an identifier. The assertion syntax is presented in Syntax Diagram 11.

Assertions checked at run time have their uses for debugging and testing, but the real value of writing down such assertions is as an aid to human thinking and reasoning about programs. It should bring you to the stage where you can see how in principle the software might be proved to calculate what it claims. It describes properties that remain true on loop boundaries (initial, final and intermediate states), so the loop invariant may be given a role very much like the inductive hypothesis employed in a mathematical



Syntax Diagram 11: Assertion syntax

induction. It thus can play an important role in deriving the loop body. A loop invariant also can be used to reason about the correctness of a loop. At the end of the loop of Example 2.8 we have both:

$$\text{invariant: } n = Result \times divisor + remainder$$

and

$$\text{termination condition: } remainder < divisor,$$

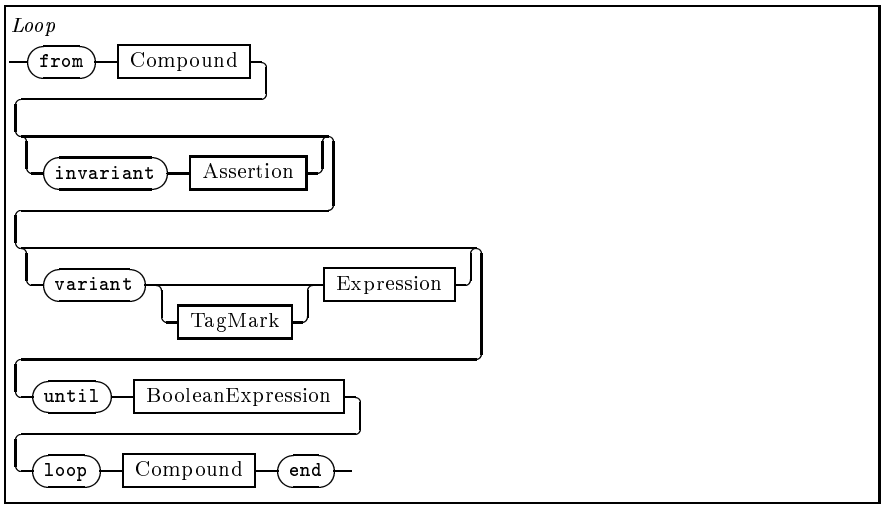
which trivially implies the inequality 2.1. Thus, if we reach the end of the loop, then the result will be the one we wanted. This property is called *partial correctness*. Let us see how to prove total correctness.

The Notion of a Loop Variant

Total correctness is just partial correctness *and* the proof that the loop terminates (i.e., the termination condition eventually will hold). The idea of a loop variant is to characterize how each iteration can bring the loop closer to its termination.

A loop variant is a non-negative integer expression that is decreased by at least one at each iteration. By definition, it cannot go below zero, thus the number of iterations of a loop with a variant is bounded and then the loop eventually terminates. A suitable variant for the Example 2.8 is *remainder*. It is a strictly decreasing function that takes non-negative values only.

A loop variant has been found, so we have proved that the loop eventually terminates. This proof completes the partial correctness to give the total correctness of the loop. In other words, the loop terminates and computes the right result. In Eiffel, the loop variant may be specified after the keyword **variant**, as a tagged integer expression. See the full syntax of the Eiffel loop construct in the Syntax Diagram 12.



Syntax Diagram 12: The full syntax of the Eiffel loop construct

So, with loop variant and invariant, Example 2.8 becomes the loop of Example 2.9.

Example 2.9

```
from remainder := n
invariant reversible: n = Result*divisor+remainder
variant decreasing_remainder: remainder
until remainder < divisor
loop                                                    5
    remainder := remainder - divisor
    Result := Result + 1
end --loop
```

Sometimes, you cannot express the loop invariant with the Eiffel assertion expressive power. Still, write it in a comment. It gives your readers the intent of what you are trying to achieve with the loop. Consider for example the problem of computing the factorial of a number n (Example 2.10). The loop invariant ($Result = i!$) describes what the iteration has computed so far, but can only be expressed as a comment.

Example 2.10

```
from i := 1; Result := 1
invariant Result_is_factorial_i: -- Result = i!
variant increasing_i: n−i
until i = n
loop                                                    5
   i := i + 1
   Result := Result * i
end
```

Designing Bug-free Loops

Eiffel fosters the systematic design of bug-free loops (but cannot force you to do so). Here is the method:

1. Design the loop invariant. It should provide a concise and preferably formal description of the properties of the loop.
2. Find the termination condition.
3. Find the variant (how the loop can advance toward its termination).
4. Write the body of the loop:
   - First deduce from the variant the progression instruction,
   - Then write the code corresponding to the restoration of the invariant.
5. Write the initialization part to set up the loop invariant.

For simple cases, steps 1 and 3 can be omitted. Still, it is a good practice to stick to the order described (termination, loop body, initialization). Example 2.9 is such a simple case, because it involves a bounded loop (i.e., the number of iterations is always n, and thus is bounded). There is no difficulty in proving that the loop terminates.

Let us exercise our systematic design method for bug-free loops with a more complex example: a binary search. Let it be an array of integers sorted in increasing order. The feature $item(i)$ gives the value of the integer stored at position i. The first position in the array is *lower*, and the last is *upper*. The array is sorted means: $\forall i \in [lower, upper - 1] \;\; item(i) \leq item(i + 1)$.

The problem is to find whether an integer x belongs to the array. The algorithm chosen is the binary search. Its principle is to compare x with *// is the integer division*

the median element m of the array, that is, **item ((lower + upper) // 2)**.

- If they are equal, we have found x (hence we set Result to True).
- If x is smaller than m, then we have to look for it in the lower part of the array (with the same method).
- If x is greater than m, then we have to look for it in the upper part of the array (with the same method).

To apply the five-step method to solve this problem in a systematic and repeatable way:

1. First, find the loop invariant. Let l and u be the lower and upper indexes delimiting the section of the array where we look for x at a given iteration. If x belongs to the array, then x lies in between *item(l)* and *item(u)*, which by contraposition gives

   $$(x < item(l) \ \ or \ \ x > item(u)) => Result = False)$$

   as loop invariant.

2. The termination condition is either that x does not belong to the array ($l > u$) or that x has been found ($Result = True$).

3. The variant is $u - l$: the range where x is supposed to be shrinks more and more.

4. The body of the loop is:

   - Make the loop progress. Let $m = (l + u)//2$. If $x < item(m)$ then $u := m - 1$ else if $x > item(m)$ then $l := m + 1$
   - Reestablish the invariant. If $x = item(m)$, $Result := True$

5. Set up the invariant in the initialization. $l := lower$ and $u := upper$.

The code presented in Example 2.11 directly follows from this loop design.

Finally, don't worry about the performance penalties of evaluating such assertions. As with other Eiffel assertions, their evaluation can be disabled with a kind of switch at compile or at run time.

Example 2.11

```
from
   l := lower; u := upper
invariant
   in_bounds: (x<item(l) or x>item(u)) implies Result = False
variant                                                          5
   range_must_shrink: u − l
until l>u or Result
loop
   m := (l + u) // 2
   if x < item(m) then                                           10
      -- x cannot be in the upper part
      u := m−1
   elseif x > item(m) then
      -- x cannot be in the lower part
      l := m+1                                                   15
   else -- x = item(m)
      Result := True
   end -- if
end -- loop
```

Syntax Diagram 13: The syntax of the *check* statement

2.4.8 The Check Statement

The check statement allows you to check that a set of assertions are verified at a given point in a program. Its syntax is described in Syntax Diagram 13.

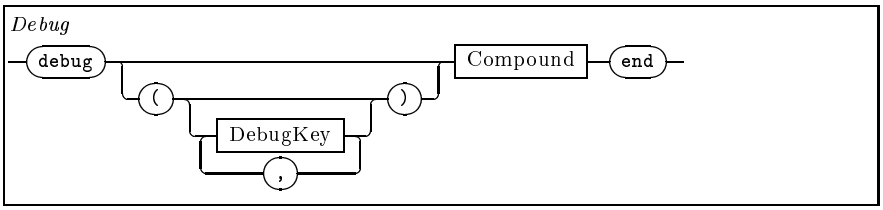
In addition to reassuring yourself that certain properties are satisfied, the check statement is another convenient way to make the assumptions on which you are relying explicit for your readers.

Actual code is generated for the *check* statement only if you activate the relevant switch at compile time. As with other constructs involving assertions, an *exception* is generated on violation of an assertion. What happens then is described in Section ?? on page ??.

2.4.9 The Debug Statement

In C (or C++) the debug statement is often realized though the preprocessor:
```
#ifdef DEBUG
...
#endif.
```

The debug statement enables the conditional execution of a compound statement, depending on a compilation option. The **debug** keyword may be followed by a list of manifest strings called *debug keys* (Syntax Diagram 14).



Syntax Diagram 14: The syntax of the **debug** statement

The debug code is executed if one of the debug keys has been selected (all keys also can be enabled at once). Depending on your Eiffel environment, this selection may be done either at compile time or at run time. See Section 4.2 on page 127 for more details. In this example, the message

Example 2.12

```
debug ("TRACE","LEVEL3")
  print ("Entering the interesting part %N")
end
```

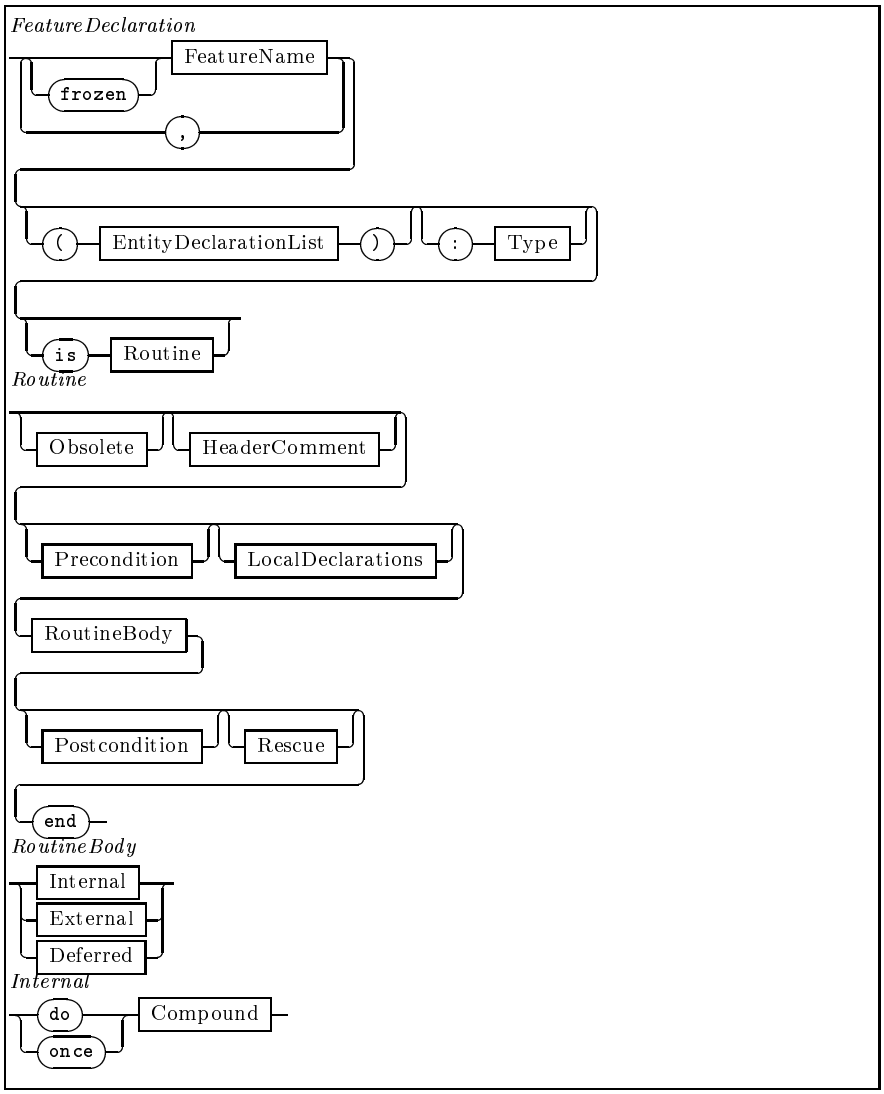
will be printed only if either TRACE or LEVEL3 debug keys are activated.

2.5 Routines: Procedures and Functions

Eiffel routines *are called* methods *in Smalltalk or* member functions *in C++.*

As seen in Section 2.2.4, the features of a class are either attributes describing data fields or routines describing computations that are applicable to instances of that class. Routines may access or update attributes of their class. A routine returning a result is called a *function;* otherwise, it is called a *procedure.* However, Eiffel fosters a style of design that clearly separates commands, implemented as procedures, from queries implemented as *pure* functions—that is, functions without side effects (see Section ??).

2.5.1 Routine Declaration



Syntax Diagram 15: Feature declaration

A routine declaration consists of an interface specification and its body

(see Syntax Diagram 15). The interface specification reflects the abstract data type view of a routine, or signature and preconditions and postconditions. The body may be either:

external: the implementation of this routine falls out of the scope of the Eiffel compiler. More details on interfacing Eiffel with foreign software are given in Section 4.4.1 on page 135.

deferred: no implementation is given for this routine. Deferred routines are presented in Section 3.6 on page 105.

regular: the description of the computations the routine performs.

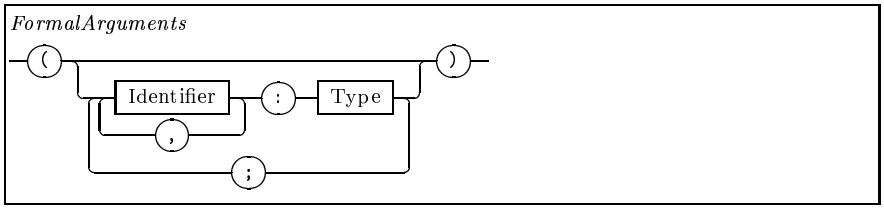
The routine may have synonyms, i.e., various feature names may be attached to the same feature body. This is actually equivalent to multiple independent definitions with the same interface and body.

*Eiffel uses the convention opposite to C++. By default Eiffel routines may be redefined in subclasses (*frozen *prevents this), whereas a C++ routine may be redefined only if declared virtual in the base class.*

The keyword **frozen** means that the associated feature name cannot be redefined in subclasses (see Section 3.4.2 on page 97).

2.5.2 Arguments to a Routine

An argument allows the routine caller to pass it information for a given execution. Within the routine, an argument is represented with a purely local name associated with a type and bears the name *formal argument*.



Syntax Diagram 16: Formal arguments syntax

Consider the binary search presented in Section 2.4.6 on page 47. It makes sense to encapsulate it within a function returning a Boolean result. Let's call this function *contains*. The corresponding declaration is then:

```
contains (x : INTEGER) : BOOLEAN is ...
```

The formal argument x is then used in the body of the routine as an entity denoting the element to look for (see Example 2.11). The *signature* of this routine is made of its name (*contains*), the type of input parameters (here INTEGER), and the type of output result (here BOOLEAN).

2.5.3 Preconditions, Postconditions, and Invariants

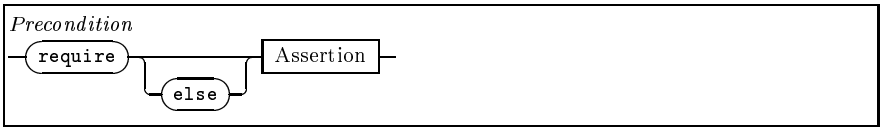
The syntax of assertions is described in Syntax Diagram 11 on page 49.

Like other assertions, preconditions, postconditions, and class invariants express the specification of software components, and are essential for documenting and testing them. A detailed introduction to the formal aspects of these assertions can be found in [10]. As a specification tool, Eiffel preconditions and postconditions try to fulfill the goal of specifying the *what* rather than the *how*. Thus even when assertions not easily specifiable with Boolean expressions (for example when quantifiers would be needed), you should still try to describe them through comments.

Preconditions

A precondition is introduced with the keyword **require** (see Syntax Diagram 17), and states the conditions under which the routine may be called. The routine caller must guarantee this condition when calling the routine, or else the routine work cannot be done. More formally, a precondition is a predicate that characterizes the set of initial states for which a problem can be solved. It specifies the subset of all possible states that the routine should be able to handle correctly.

The keyword **else** *is used in conjunction with the mechanism of redefinition described in Section 3.4.2.*



Syntax Diagram 17: Precondition syntax

For example, a precondition should be used to specify that a quotient may be computed for non negative integers and positive divisors only (see Example 2.13).

Example 2.13

```
quotient(n,divisor:INTEGER):INTEGER is
        -- Integer division of n by divisor
    require
        non_negative_n: n >= 0
        positive_divisor: divisor > 0
```

Postconditions

A postcondition is introduced with the keyword **ensure** and appears just before the end of the routine (see Syntax Diagram 18). The postcondition states the property that the routine must guarantee at completion of any correct call. It provides a formal specification of what the routine should accomplish. It thus can assist in the construction of the routine, and be used as a constructive proof of its correctness.

The keyword **then** *is used in conjunction with the mechanism of redefinition described in section 3.4.2.*

Syntax Diagram 18: Postcondition syntax

In the case of the quotient function, the function specification (as described in Inequality 2.1) readily translates itself to the function postcondition. Putting all parts together, the specification of the quotient function becomes what is presented in Example 2.14.

Example 2.14

```
quotient(n,divisor:INTEGER):INTEGER is
        -- Integer division of n by divisor
    require
        non_negative_n: n >= 0
        positive_divisor: divisor > 0                                    5
    ensure
        Result*divisor <= n and n < (Result+1)*divisor
    end -- quotient
```

An *"old" expression* is a special notation (see Syntax Diagram 19) available in postconditions only. The value of an old expression is the value the expression had before entering the routine. It appears in the routine postcondition, so it allows the specification of the effect of the computation with respect to the previous state. Consider for example the specification of a routine to swap two elements of an array (Example 2.15). Its postcondition states that the value at index i is now the value that was at index j on entering the routine, and conversely.

Syntax Diagram 19: Old expression

Example 2.15

```
swap (i, j: INTEGER) is
        -- swap item(i) and item(j)
    require
        valid_i: i>=lower and i<=upper
        valid_j: j>=lower and j<=upper                    5
    ensure
        item(i) = old item(j) and item(j) = old item(i)
    end -- swap
```

Invariants

Class invariants do not syntactically belong to a given routine. They actually characterize properties that the enclosing module must respect at any time. This relationship has a consequence for routines, because the class invariant must be true both on entering a routine (thus strengthening its precondition) and on exiting it (thus strengthening its postcondition).

Assertions and Programming by Contract

Assertions fulfill a crucial role in supporting a clear separation of responsibilities in a modular system. They foster the formalization of the contract binding a routine caller (the client) and the routine implementation (the contractor or supplier).

> *Provided the client calls the contractor routine in a state in which the class invariant and the precondition of the routine are respected, the contractor promises that when the routine returns, the work specified in the postcondition will be done, and the class invariant will be respected.*

If either party fails to meet the contract terms, the whole program should be considered invalid. In other words, there is a bug somewhere. The section on exception handling (Section **??** on page **??**) explains what

happens next, and how Eiffel programming environments help to identify both the faulty party and the fault itself. The contract can actually be broken in two different ways:

- If the precondition was violated, then the client broke the contract. Its code should have been written to avoid this. The contractor should not try to comply with its part of the contract, but should signal the fault by raising an exception.

- If the precondition was satisfied but the postcondition was violated, the implementation of the routine did not fulfill its promises, which is commonly referred as an *implementation bug.*

The design by contract approach provides a methodological guideline to build robust, yet modular and simple systems without resorting to defensive programming. It has a sound theoretical basis in relation to partial functions (see reference [29]). It is not surprising then that the notion of contractual programming is a cornerstone in the design of reusable software components in Eiffel. This approach could be emulated partially in C or C++ with the "assert.h" package. The Eiffel assertion mechanism is, however, much more powerful because it is fully integrated to the type system and the inheritance mechanism, and thus provides the necessary semantics for subtyping and subclassing. This mechanism will be described in full detail in Section 3.4.2 on page 97.

2.5.4 Calling a Routine

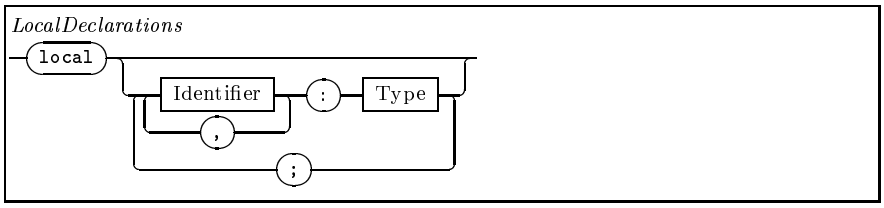
The only syntactical difference between a function and a procedure is that the function yields a result. Calling a function without argument presents no syntactic difference with the reading of an attribute: the caller does not need to know whether a given function is implemented with an attribute. The transmission of values between the caller and the callee consists of assigning the actual arguments provided by the caller to the formal arguments of the routine. For example, calling $q := quotient(10, 3)$ is equivalent to setting $n := 10$ and $divisor := 3$ within the function *quotient*, executing the rest of the routine, and assigning its result to q.

The semantics of this assignment are described in Section 2.4.1.

2.5.5 Internal Routine Body

A routine body may have a local declaration clause that allows the declaration of entities available within the routine body only. The name of these local entities may not override the feature names of the enclosing module.

A local declaration clause is equivalent to the variable and constant declarations that are local to a procedure or function in C, Pascal, or Ada.



Syntax Diagram 20: Local declaration clause in an internal routine body

Example 2.16

```
local
  i, j : INTEGER
  is_empty : BOOLEAN
  total : REAL
```

The syntax of a local declaration clause is presented in Syntax Diagram 20.

Functions have an additional local entity (denoted with the reserved word **Result**) that holds the result returned by the function. All the local entities (including **Result** for a function) are always initialized according to the default initialization rules for their type (see Section 2.3.4). Local entities are destroyed when the routine finishes:

- Expanded objects just vanish (are popped from the stack),
- Reference objects become unreachable (if they are not attached to non-local entities) and thus become fair game for the garbage collector.

Note that you never need to worry about memory management issues with Eiffel unless you really insist on doing so (See Section 4.5).

The executable part of a routine normally is introduced with the keyword **do**. It consists of a compound statement followed by an optional rescue clause, which allows it to deal with the exceptions that might have been raised in the compound (see Section **??** on page **??**). Example 2.17 presents the full definition of the *quotient* function.

Example 2.17

```
quotient(n,divisor:INTEGER):INTEGER is
        -- Integer division of n by divisor
    require
        non_negative_n: n >= 0
        positive_divisor: divisor > 0
    local
        remainder : INTEGER
    do
        from remainder := n
        invariant reversible: n = Result*divisor+remainder
        variant decreasing_remainder: remainder
        until remainder < divisor
        loop
            remainder := remainder - divisor
            Result := Result + 1
        end --loop
    ensure
        Result*divisor <= n and n < (Result+1)*divisor
    end -- quotient
```

2.5.6 Once Routines

Once routines are special routines. They have the same syntax, except for the keyword **once** used instead of **do** to introduce the compound statement. The first time the routine is called, it works exactly like a regular routine. Subsequent calls, however, have no effect. If the **once** routine is a function, the value it returns is the same as the value returned by the first call. This mechanism enables:

- The initialization of a data structure without an explicit initialization requirement,

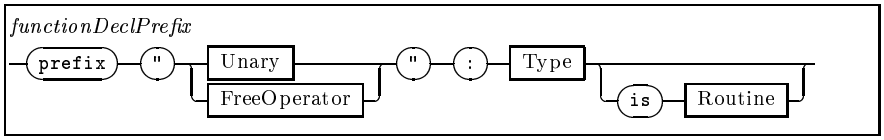
This is equivalent to the notion of class variable found in Smalltalk or C++

- The sharing of values that are computed at run time,
- The sharing of variables among all instances of a class.

2.5.7 Prefix and Infix Function Declaration

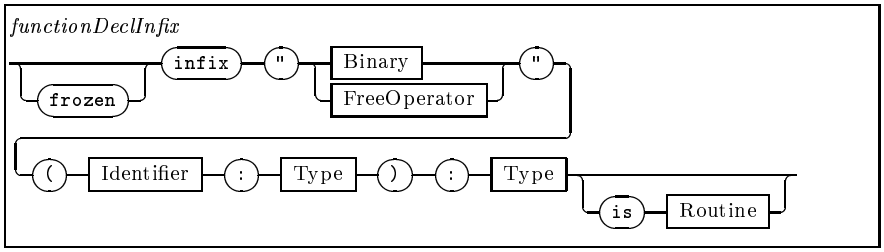
To enable the usual practice of writing boolean and numerical expressions (such as -2*x), two alternative forms of function declarations are provided:

- The prefix form for unary operators (functions without argument), the syntax of which is described in Syntax Diagram 21. Predefined unary operators include: `not, +, -`.



Syntax Diagram 21: Prefix operator declaration

- The infix form for binary operators (functions with exactly one argument), the syntax of which is described in Syntax Diagram 22. Predefined binary operators are listed in Syntax Diagram 23.



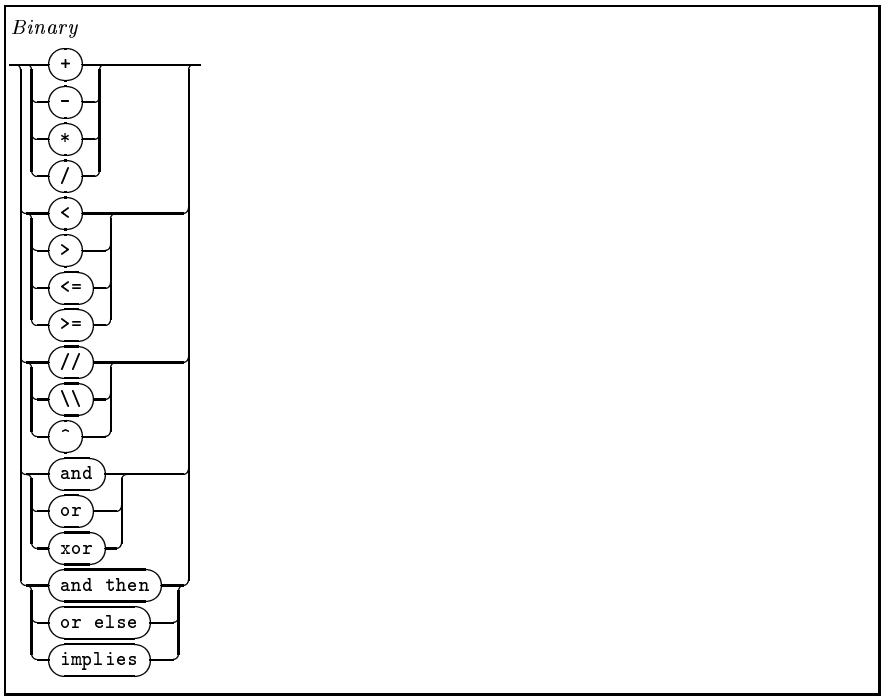
Syntax Diagram 22: Infix operator declaration

Free operators are sequences of non blank characters beginning with either @, #, |, or & .

In the Eiffel standard library class BOOLEAN, we can find the declaration of the prefix function *not* giving the negation of a Boolean value (Example 2.18). We also can find in this class the usual infix operators to manipulate Boolean values (Example 2.19). The operator **and then** is called *semistrict* because it evaluates its second argument only if the first one is *True*. In the same way, the operator **or else** evaluates its second argument only if the first one is *False*.

These operators have the same semantics as the && and || found in C and C++.

These notations allow for the usual way of describing a Boolean expression; e.g., `not (a or b) implies c or else d`.



Syntax Diagram 23: Binary operator list

Example 2.18

```
prefix "not" : BOOLEAN
        -- Negation.
```

Prefix and infix declarations (for arithmetic and comparison operators) are also available in Eiffel standard library classes such as INTEGER, REAL, and DOUBLE. Calling such prefix and infix functions involves writing expressions with the usual ALGOL like syntax found in Pascal or Ada:

```
a := - b^2 * (c + d)
if a <= e // 2 then ...
```

Example 2.19

```
infix "and" (other: BOOLEAN): BOOLEAN
      -- Boolean conjunction with 'other'
infix "and then" (other: BOOLEAN): BOOLEAN
      -- Boolean semi strict conjunction with 'other'
infix "implies" (other: BOOLEAN): BOOLEAN
      -- Boolean implication of 'other'
infix "or" (other: BOOLEAN): BOOLEAN
      -- Boolean disjunction with 'other'
infix "or else" (other: BOOLEAN): BOOLEAN
      -- Boolean semi strict disjunction with 'other'
infix "xor" (other: BOOLEAN): BOOLEAN
      -- Boolean exclusive or with 'other'
```

2.5.8 Recursion

Recursion is a powerful programming technique that fosters elegant solutions for several problems. The concept of recursivity is the same in Eiffel as it is in Ada, Pascal, or C. The explicit preconditions and postconditions associated with a recursive routine, however, make it easier to produce correct implementations. A routine is said to be recursive if its body makes reference to itself, either directly or indirectly through other routines that call it.

Recursion is analogous to mathematical induction. The solution of a problem p_m is formulated supposing that each problem p_{n-1}, $p_{n-2}, \ldots, p_1$ is solved.

Consider the factorial function again. Another mathematical definition for it is:

$$\forall n \in [1..\infty[\ \ n! = n * (n-1)!\ ;\ 0! = 1$$

The corresponding algorithm directly follows (Example 2.20).

A more interesting example is the recursive version of the binary search presented in Example 2.11. The loop invariant of this example readily transforms itself in to the recursive routine postcondition: $(low > up) \Rightarrow Result = False$. The recursivity progresses by reducing the "search space": the function *belongs_range* calls itself recursively on a range that is smaller than the current one. The recursivity stops when an empty range is found (see Example 2.21).

Example 2.20

```
rfacto(n : INTEGER): INTEGER is
        -- the factorial of n, recursive algorithm
    require non_negative: n >= 0
    do
        if n = 0 then                                          5
            Result := 1
        else
            Result := n * rfacto(n-1)
        end; -- if
    ensure                                                     10
        positive_result: Result > 0
        factorial_computed : -- Result = n !
    end; -- rfacto
```

Example 2.21

```
belongs_range(low, high: INTEGER; x : T) : BOOLEAN is
        -- whether x belongs to the range [low..high]
        -- Recursive binary search algorithm in O(log(n)).
    require data_is_sorted: is_sorted
    local                                                      5
        m : INTEGER
    do
        if low > high then -- stopping condition for the recursion
            Result := false -- x not found
        else                                                   10
            m := (low + high) // 2
            if x < item(m) then
                -- x cannot be in the upper part
                Result := belongs_range (low,m-1,x)
            elseif x > item(m) then                            15
                -- x cannot be in the lower part
                Result := belongs_range (m+1,high,x)
            else -- x = item(m)
                Result := True
            end -- if                                          20
        end -- if
    ensure
        empty_implies_false: (low > high) implies not Result
    end -- belongs_range
```

2.6 Example: Sorting Data with Eiffel

In this section we try to illustrate the Eiffel constructions presented so far in a more substantial example.

Consider a data structure (typically an array) that is indexed on the interval [lower,upper]. Let us first implement a function that indicates whether a subrange of this array is sorted in increasing order. In the context of *programming by contract*, the first thing that we have to design is the contract of this function—that is, its preconditions and postconditions. We do not know what to do if the parameters of this function are not correct, so we should not try to do anything. Instead, we *must* specify that we expect correct parameters: e.g., the subrange must be included in the index domain of the array: [lower,upper]. This specification gives the preconditions of lines 4–5 in Example 2.22. Like most preconditions, these are almost trivial, but nonetheless extremely useful to specify the routine behavior and also simplify debugging, because a precondition violation will give a precise error message. There is no explicit postcondition here.

The core of this function is a simple loop to check that the i^{th} item is smaller than the next item. The invariant is thus that the subrange [low .. i-1] is sorted, and the termination condition is either that we find an unordered element (such that $item_i < item_{i+1}$), or that we reach the end of the subrange ($i \geq high$). An empty or a single element range is always considered as sorted. Since this is clearly a bounded loop, the variant is only helpful to double-check that we did not forget the incrementing of i within the loop. The full text of this function then can look like Example 2.22

Conversely, consider the problem of sorting this data structure with the *quicksort* algorithm invented by C.A.R. Hoare. It is remotely based on the *bubble sort* algorithm, the principle of which is to consider pairs of neighbor items, and to move the smaller before the larger. This process is repeated until all elements are ordered. The problem with bubble sort is that the number of comparisons it requires is proportional to the square of the number of items being sorted (this algorithm complexity is thus denoted $O(n^2)$).

The *quicksort* algorithm, also called *partition sort,* is a major improvement to this method. It is based on the idea that exchanges should preferably be performed over large distances to be most effective. The idea is to consider the first item (called the pivot) and scan the array from the left until an item i such that $item(i) > item(first)$ is found, and from the right until an item j such as $item(j) <= item(first)$ is also found. Then the two items are exchanged, and this "scan and swap" process is continued until the two scans meet somewhere in the middle of the array (see Figure 2.6). The first item is then exchanged with $item(j)$, and we now get an array

Example 2.22

```
is_sorted_range (low, high: INTEGER) : BOOLEAN is
        -- are the objects in low .. high in (non strict) increasing order?
    require
        low_large_enough: low >= lower
        high_small_enough: high <= upper                                    5
    local
        i: INTEGER
    do
        from i := low + 1; Result := True
        invariant left_subrange_is_sorted: -- [low .. i-1] is sorted        10
        variant high - i + 1
        until (not Result) or i >= high
        loop
            Result := item(i) <= item(i+1)
            i := i + 1                                                      15
        end -- loop
    end -- is_sorted_range
```

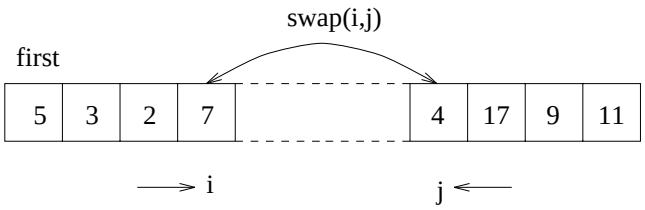


Figure 2.6: The *quicksort* "scan and swap" process.

partitioned into a left part with items smaller than the pivot item, and a right part with items greater than it. The same process then may be applied (recursively) to both partitions, until every partition consists of a single item only (and thus is sorted). The advantage of this algorithm is that it tends to have an algorithmic complexity in $O(n.log(n))$, which brings considerable performance improvement over $O(n^2)$ when sorting large data structures.

The Eiffel implementation of this algorithm closely follows this scheme. First, let us design the precise contract of the procedure *quicksort_range(first, last)* taking as input the range to be sorted. There are two possible policies: either require correct parameters (a nonempty subrange

included in [lower,upper]) and ensure *is_sorted_range(first, last)* at the end of the routine, or accept any parameters and do (and ensure) nothing in case of incorrect parameters. As displayed in Example 2.24, we have chosen the former option.

According to the *quicksort* algorithm, the body of the *quicksort_range* procedure is divided roughly into three parts: the partitioning loop (lines 12–29), the swapping of the first element (*item[m]*) with the pivot (line 30, calling the routine *swap* defined in Example 2.15 on page 59), and the recursive call for both partitions if they are wider than 1 (lines 31–32). This body is surrounded with debug instructions (lines 8–11 and 33–36) are activated with the "Recursion" key to allow the printing of the sequence of recursive calls to the procedure.

To design the partitioning loop, we start with the invariant definition that follows from the method described:

$$\forall k \in [first..i-1] \;\; item(k) \leq item(first)$$

$$\forall k \in [j+1..last] \;\; item(k) > item(first)$$

Eiffel does not feature quantifiers, so these assertions may only be expressed with comments. However, the loop is complex enough to justify some effort to allow the invariant to be actually checked at run time. We thus can design a function to compute these assertions, and use it in the invariant clause of the loop. Instead of building an ad hoc function, we may use (or reuse) more general purpose functions giving the minimum and the maximum items of a subrange of an array. These functions allow for the definition of the invariant as it appears in lines 13–15 of Example 2.24.

The definition of the function max_value_in *is left to the reader.*

The loop variant is that either i or j must move at each step of the iteration, and the termination condition is that they cross themselves. The body of the loop is a mere translation of the core of the method described, and the initializations of i and j follow. The full text of the *quicksort_range* procedure is presented in Example 2.24.

Example 2.23

```
min_value_in(low, high: INTEGER) : T is
        -- the lowest element in the array for the range low..high
    require
        low_large_enough:   low >= lower
        high_small_enough:   high <= upper                              5
        range_not_empty:   low <= high
    local
        i : INTEGER
    do
        from i := low; Result := item(i)                                10
        invariant min_so_far: -- for all j in [low .. i], Result <= item(j)
        variant increasing_i: high - i
        until i = high
        loop
            if item(i) < Result then                                    15
                Result := item(i)
            end -- if
            i := i + 1
        end -- loop
    end -- min_value_in                                                 20
```

Example 2.24

```
quick_sort_range(first, last: INTEGER) is
        -- sort elements first..last into increasing order
        -- (quick sort algorithm)
    require range_not_empty: first <= last
    local                                                          5
        i,j: INTEGER
    do
        debug ("Recursion")
            print("Entering quick_sort_range(")
            print(first); print(','); print(last); print(")%N")    10
        end -- debug
        from i := first + 1; j := last
        invariant
            left_lower: max_value_in(first, i-1) <= item(first)
            right_higher: (j < last)                               15
            implies (min_value_in(j+1, last) > item(first))
        variant ends_converging:  j - i + 2
        until i > j
        loop
            if item(i) <= item(first) then                         20
                i := i + 1          -- advances i rightwards
            elseif item(j) > item(first) then
                j := j - 1          -- advances j leftwards
            else                    -- i & j have been found
                swap(i,j)           -- swap item(i) and item(j)    25
                i := i + 1          -- next i & j
                j := j - 1
            end -- if
        end -- loop
        swap(first, j)              -- places back the pivot       30
        if first < j - 1 then quick_sort_range(first, j - 1) end
        if j + 1 < last then quick_sort_range(j + 1, last) end
        debug ("Recursion")
            print("Exiting quick_sort_range(")
            print(first); print(','); print(last); print(")%N")    35
        end -- debug
    ensure sorted: is_sorted_range(first, last)
    end -- quick_sort_range
```

Chapter 3

Object-Oriented Elements

Until now we have worked within a single class, which was viewed as a module. We now describe how to use the services provided by other classes (with the example of using the class STRING*), and conversely, how to make services available to other classes (for that we design a simple* LISTINT *class). We later introduce how these services may be made* generic *and we show how to use a generic class* ARRAY*[T] and how to build a generic class* LIST*[T]. The inheritance relationship and some of the consequences it has on subtyping, polymorphism, and dynamic binding are described. This chapter ends with a medium-sized case study of a classic software engineering problem: the keyword-in-context (KWIC) system.*

3.1 Working with Modules

The *client* of a module is an object that uses the services provided by the module. The Eiffel notion of module boils down to the class. It is simply a box put around a number of features (attributes and routines). This box has an interface (the client's view of the module), which is like a control panel with a light-emitting diode displaying the value of exported attributes and functions, and buttons allowing the user to call exported procedures. For example, the random number generator in Figure 3.1 has a function named *last_random_real* returning (i.e., displaying) the last random real value computed by the random generator, and two procedures (namely *reset* and *next*) to respectively reset the random generator and make it compute the next random real value. The full text of this class is given in

The interface presented by a class may depend on who is looking at it. This is called subjectivity*. See the selective export clause in Section 3.1.4.*

Example 3.8 on page 79.

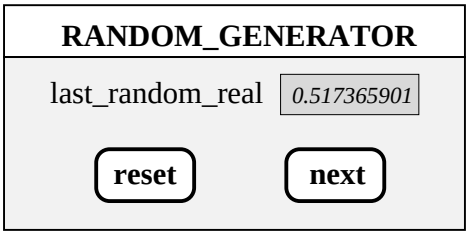


Figure 3.1: A random number generator

3.1.1 Creating Objects

A creation of an expanded object is not prohibited—it just resets the object to its default initial value.

Objects are created as instances of classes. Let *e* be an entity with the declared type SOMECLASS:

- If SOMECLASS is an expanded type (e.g., INTEGER), *e* directly holds the value of the object, and is initialized according to the default initialization rules presented in Section 2.3.4. Thus, no creation is needed.
- If SOMECLASS is a reference type (e.g., RANDOM_GENERATOR), a creation instruction is needed to dynamically allocate a new object attached to *e*.

In the simplest form, if SOMECLASS has no creation clause, a new instance of SOMECLASS is allocated, initialized (that is, its fields are given their default values), and attached to *e* by the means of a *creation* instruction made of two exclamation marks (!!) preceding the entity name (cf. Example 3.1).

Example 3.1

```
local
  e : SOMECLASS
do
  !!e     -- An object of type SOMECLASS is created,
          -- initialized and attached to e                    5
end
```

If SOMECLASS has one or more creation features, one of them must be called when the object is created. For example, the creation routine for the

class RANDOM_GENERATOR is the routine *reset* which takes as argument a new seed value for the generator (see example 3.8). Example 3.2 illustrates how such an object can be created and attached to an entity. The effect of

Example 3.2

```
local
  e : RANDOM_GENERATOR
do
  !!e.reset(123)
        -- An object of type RANDOM_GENERATOR is created,          5
        -- initialized, attached to e, and then the
        -- creation procedure reset is called on e
end
```

this instruction is the same as in Example 3.1; in addition, the *reset* routine is immediately called on the fresh object (thus allowing it to establish the class invariant).

The third form of the creation instruction allows you to create an object instance of a subclass of SOMECLASS instead of a direct instance of SOMECLASS (see Example 3.3). More details on subclassing appear in Section 3.3.

Example 3.3

```
local
  e : SOMECLASS
do
  !SUBCLASS!e.make(...)
        -- An object of type SUBCLASS is created,                  5
        -- initialized, attached to e, and then the
        -- creation procedure make  is called on e
end
```

Clone, *like* print, *is a feature defined in the class* GENERAL, *which is an implicit common ancestor to all classes.*

The last way to get a new object is to clone an existing one. The function *clone* is available in all classes to create a new object that is a field-by-field copy of the original object and has the same type (see Example 3.4).

If e is **Void**, then f is also set to **Void**. There also exists a variant of *clone*, called *deep_clone*, which duplicates the entire structure referenced by the original object (that is, deep clones all its fields).

Example 3.4

```
f := clone(e)          -- now we have equal(e,f)
g := deep_clone(e)     -- now we have deep_equal(e,g)
```

3.1.2 Calling Other Object Features

In the spirit of accessing fields of a Pascal record or a C structure, the dot notation is used to call a feature *foo* of an object *obj* (see Example 3.5).

Example 3.5

```
obj.foo          -- calls the feature foo on the object obj
```

In C++, you would write obj.foo for a value object and obj->foo,equivalent to (*obj).foo*, for reference objects.*

This syntax is valid for any kind of feature call, so in Example 3.5 the feature *foo* also might be an attribute. Whether the object is expanded or not has no more influence on the calling syntax.

An attribute or the result of a function call are entities themselves, so the feature calls may be cascaded as in Example 3.6. If some of the features

Example 3.6

```
obj.foo.bar.etc  -- calls foo on obj, and then
                 -- bar on the result of foo, and then
                 -- etc on the result of bar
```

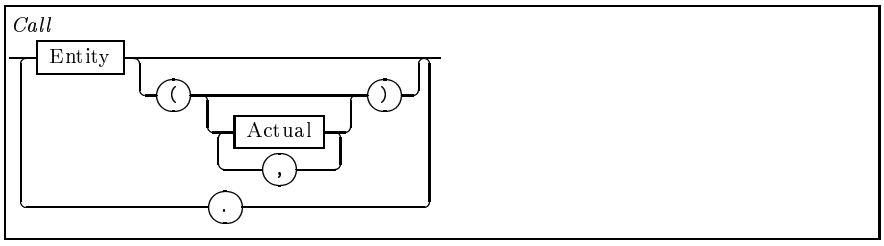
are in fact functions, they may have parameters. The formal syntax of such a call chain is presented in Syntax Diagram 24.

The current object is called this *in C++ or* self *in Smalltalk.*

The form of the feature call itself is the same as the unqualified call defined in Section 2.5.4 on page 60. Actually an unqualified call implicitly refers to the current object (denoted **Current** in Eiffel) and is thus equivalent to `Current.foo` (as far as assertion checking is not concerned).

To illustrate this feature call mechanism, Example 3.7 shows a fragment of code designed to compute the mean value of a number of consecutive pseudo random values generated by a RANDOM_GENERATOR object.

Note the distinction between the command (`gen.next`) asking the generator to produce the next random number, and the query `gen.last_random_real(0,100)`, which is a pure function call. That is, if we would call this function again with the same input parameters (and without calling `gen.next` in between), we would get the same result. This



Syntax Diagram 24: Call chain syntax

Example 3.7

```
test_a_random_generator is
    local
        gen : RANDOM_GENERATOR
        i : INTEGER
        mean : REAL                                          5
    do
        !!gen.reset(1234)
        from i := 1
        until i > 1000
        loop                                                 10
            gen.next
            mean := mean + gen.last_random_real(0,100)
            i := i + 1
        end -- loop
        mean := mean / 1000                                  15
        print("Mean = "); print(mean); io.new_line
    end -- test_a_random_generator
```

strict distinction between commands and queries is a recommended design style in Eiffel [29] (see Section **??**).

3.1.3 Attribute Protection and Information Hiding

By default all features of a class are visible (exported) to every other class. Visible only means that another object can "see" the values of attributes and functions, and "push" the buttons corresponding to the procedures. Thus, an object may not modify an attribute of another object directly: the assignment `obj.attribute := value` is (syntactically) illegal.

In C++ terms, one would say that every routine is "public," whereas attributes are read only. In Smalltalk terms, Eiffel has for each attribute an associated implicit method giving its value.

A group of features appearing after a **feature** keyword can be made *private* to the class if the **feature** keyword is followed by the export clause: {NONE}. The default case in which no class is specified after a **feature** keyword is equivalent to exporting to ANY; thus, the features are made public. Conversely, exporting to the pseudo class NONE allows the total hiding of a set of features: they no longer can be directly called from outside of the class. In Example 3.8, the features *seed, aa, bb, invmaxint* and *uniform* are private to the class RANDOM_GENERATOR; they cannot be directly called by any client class.

Example 3.8

```
indexing
    description: "pseudorandom real number generator"
class RANDOM_GENERATOR
creation
    reset                                                          5
feature
    reset (new_seed:INTEGER) is
            -- Reset this random number generator with a new seed
        require positive: new_seed >= 0
        do                                                         10
            seed := new_seed
        end -- reset
    next is
            -- advance the generator
        do                                                         15
            seed := aa + seed * bb
            if seed < 0 then
                seed := - seed
            end -- if
        end -- next                                                20
    last_random_real(lower,upper:REAL): REAL is
            -- Return an evenly distributed random number over the
            -- interval [lower, upper[.
        require non_empty_interval: lower<upper
        do                                                         25
            Result := lower + uniform * (upper-lower)
        ensure
            in_bounds: Result >= lower and Result < upper
        end -- last_random_real
feature {NONE}                                                     30
    seed : INTEGER
    aa : INTEGER is 987654321
```

```
    bb : INTEGER is  31415821
    invmaxint : REAL is
            -- inverse of the largest positive integer.                35
        once
            Result := 1.0 / (2^(31) - 1)
        end -- invmaxint
    uniform : REAL is
            -- Return an evenly distributed random number over [0.0, 1.0[ 40
        do
            Result :=  seed * invmaxint
        ensure
            normed: Result >= 0.0 and Result < 1.0
        end -- uniform                                                   45
invariant
    non_negative_seed : seed >= 0
end -- class RANDOM_GENERATOR
```

A client of the class RANDOM_GENERATOR is not aware of the way it is implemented. Thus, it may make no assumptions about it: the client only sees the interface displayed in Figure 3.1. If in the future we need to change the RANDOM_GENERATOR implementation (which is probable considering the weakness of its spectral properties, i.e., the mathematical measurement of how random the number sequence is), it could be done without modifying a single character of the client's code. The client's code then is said to be *decoupled* from the actual RANDOM_GENERATOR implementation. This property is called *information hiding*. It is the main benefit of the Eiffel class-based modularity.

3.1.4 Restricted Export and Subjectivity

As a middle term between fully private and fully public features, Eiffel allows you to restrict the visibility of a set of features to a nominative list of classes (and their descendants). This list may be given between brackets after a **feature** keyword. In Example 3.9, the feature *f* is made visible to objects of CLASS1 and CLASS2 only.

Example 3.9

```
feature {CLASS1, CLASS2}
    f is do...end
```

This property is sometimes called *subjectivity* because the view clients get on this kind of class depends on which client is looking [2]. Consider

for example a COFFEE_MACHINE class. Its interface to normal clients is illustrated in Figure 3.2.

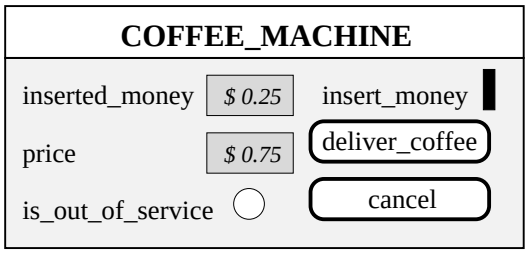


Figure 3.2: The normal client's view of a COFFEE_MACHINE

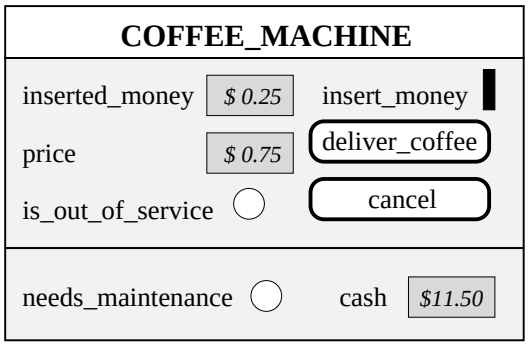


Figure 3.3: The manager's view of a COFFEE_MACHINE

Now consider a class MANAGER dealing with the supervision of coffee machines. It may need to access other features of the class COFFEE_MACHINE, such as the amount of cash deposited within the machine, or whether the machine requires maintenance. The class COFFEE_MACHINE then could be defined as shown in Example 3.10. The MANAGER'S view of this class is displayed in Figure 3.3.

3.1.5 Using Eiffel Strings

This section also serves as an Eiffel string primer.

A module that should be available with any Eiffel implementation is the class STRING. A module may be used in black-box mode by its clients (e.g., without them knowing anything about the way strings are implemented). In the next section we "enter into the box" to show how Eiffel may be used to design such modules.

Example 3.10

```
class interface COFFEE_MACHINE
feature
    is_out_of_service : BOOLEAN
    price : INTEGER -- price of a cup of coffee
    inserted_money : INTEGER -- amount of money currently inserted
    insert_money (amount : INTEGER) is
            -- insert money in the machine. Money is in cents.
        ensure
            money_added: inserted_money = old(inserted_money) + amount
    deliver_coffee is
            -- pilot the hardware to deliver a cup of coffee
        require
            in_service: not is_out_of_service
            paid: inserted_money >= price
        ensure
            cash_added: cash = old(cash) + old(inserted_money)
            reset_money: inserted_money = 0
    cancel is
            -- eject the money already introduced
        ensure
            reset_money: inserted_money = 0
feature {MANAGER}
    cash : INTEGER -- amount of cash stored in the machine
    needs_maintenance : BOOLEAN is
            -- condition upon which some maintenance is needed
end -- COFFEE_MACHINE
```

Eiffel strings (made of finite sequences of characters) are instances of the kernel library class STRING (outlined in Example 3.11). A STRING object is thus a regular Eiffel object. It may be created with the creation procedure *make*, as in Example 3.12.

Alternatively, a STRING object may get its initial value from a manifest string (Example 3.13). In that case no creation is needed, because the manifest string already exists.

The entity *s* attached at runtime to an object declared to be of type STRING has no reason to be the sequence of characters itself. Most probably a STRING object contains information such as the actual string length and a reference to the actual content. The content of the string can be accessed through a set of features. The specification of this feature set is presented in Example 3.14.

Example 3.11

```
class STRING
creation
    make
feature
    make (n: INTEGER)                                          5
            -- Allocate space for at least 'n' characters.
        require
            non_negative_size: n >= 0
        ensure
            empty_string: count = 0                            10
```

Example 3.12

```
  local
    s : STRING
  do
    !!s.make(80)
```

Example 3.13

```
  local
    s : STRING
  do
    s := "Hello World!"
```

Strings can be compared for equality or precedence (lexicographical order) as illustrated in Example 3.15.

Other comparison operators (such as $<=, >, >=, min$, and max) are also available. Beware that if *s1* and *s2* are declared as STRING, (`s1 = s2`) is just testing the two entities *s1* and *s2* against *reference* equality; that is, testing if *s1* is an alias for *s2*.

There are three possible assignment-like operations:

1. `s1 := s2` is a reference assignment *s1* will be attached to the same object as *s2*. This is a classic case of aliasing, which is useful when both entities need to have access to a common underlying sequence of characters (e.g., for message or error strings). Any aliased entity can change the string's characters. The content of the string is shared by all the entities referring to it.

Example 3.14

```
has (c: CHARACTER) : BOOLEAN
  -- Does string include c?
   ensure
      not_found_in_empty: Result implies not empty
index_of (c: CHARACTER; start: INTEGER) : INTEGER                    5
  -- Position of first occurrence of c at or after start; 0 if none
   require
      start_large_enough: start >= 1
      start_small_enough: start <= count
   ensure                                                            10
      non_negative_result: Result >= 0
      at_this_position: Result > 0 implies item (Result) = c
      -- none_before: For every i in start..Result, item (i) /= c
      -- zero_iff_absent:
      -- (Result = 0) = For every i in 1..count, item (i) /= c       15
infix "@",
item (i: INTEGER) : CHARACTER
  -- Character at position i
   require
      good_key: valid_index (i)                                      20
substring_index (other: STRING; start: INTEGER) : INTEGER
  -- Position of first occurrence of other at or after start; 0 if none
```

Example 3.15

```
is_equal (other: STRING) : BOOLEAN
        -- Is 'Current' made of the same character sequence as 'other'?
     require
        other_not_void: other /= void
infix "<" (other: STRING) : BOOLEAN                                  5
        -- Is 'Current' lexicographically less than than 'other'?
     require
        other_not_void: other /= void
     ensure
        asymmetric: Result implies not (other < Current)             10
```

2. `s1 := clone(s2)` attaches to *s1* a new string object that, although consisting of identical characters, is not related to the string attached to *s2*. This is useful when you want to have a *private* copy of the string; i.e., one that is not subject to change by outside code.

3. `s1.copy(s2)` replaces the string object attached to *s1* with a copy of *s2*. It has the same effect as the previous case, except that it allows the reuse of an existing string's object. This is valid if and only if *s2* is not **Void**.

The class STRING also has a set of features to get the string status (Example 3.16). Also available are features to:

- Change the string contents (resize, clear, fill_blank, etc.),
- Append and prepend string representations of objects (Boolean, character, integer, real, double, string) to the current string,
- Convert the string to yield an object value (a boolean, an integer, etc.) or to lower case or uppercase,
- Get a copy of a substring.

Example 3.16

```
empty : BOOLEAN
        -- Is string the empty string?
count : INTEGER
        -- Actual number of characters making up the string.
valid_index (i: INTEGER): BOOLEAN                                    5
        -- Is 'i' within the bounds of the string?
```

3.1.6 Building a Linked List Class

A class may be seen as a black box. Let us "enter into the box" and show how to build it. Consider the notion of a list of integers as it is well known to, for example, LISP users. Basically, such a list has three features:

- The *head* which is the first element of the list (it was known as the *car* in LISP),
- The *tail* of the list, which is also a list of integers (known as the *cdr* in LISP),
- A function *append*, which allows the user to prepend a new head to a list.

We add a fourth one, the feature *has* to check whether a given integer belongs to the list. Completing this class LISTINT with more sophisticated features is left as an exercise to the reader. This list of integers is a recursive data type, because the definition of the type involves the type itself. We do not use the keyword **expanded** in the class header (see Example 3.17), so the class LISTINT is a reference type and this recursive definition does not pose problem.

Example 3.17

```
indexing
    description: "Simple lisp-like linked list of integers"
class LISTINT
creation
    make                                                          5
feature
    head: INTEGER
    tail: LISTINT
    make (new_head: INTEGER; new_tail: LISTINT) is
            -- make a new list prepending 'new_head' to 'new_tail'  10
        do
            head := new_head
            tail := new_tail
        end -- make
    append (new_head: INTEGER): LISTINT is                        15
            -- return a new list with 'new_head' prepended to Current
        do
            !!Result.make(new_head,Current)
        end -- append
    has (v : INTEGER): BOOLEAN is                                 20
            -- does the list contain a value equal to v ?
        do
            Result := head.is_equal(v)
                or else (tail /= Void and then tail.has(v))
        end -- has                                                25
end -- LISTINT
```

To illustrate the use of this class LISTINT, Example 3.18 presents a routine to put the values [7,5,3,2] in such a list, and Example 3.19 prints all the list elements.

Although this list implementation will do the job, it is deficient in several aspects. It is unnecessarily tied to the INTEGER type and its interface merely reflects its implementation. This problems illustrate the fact that

Example 3.18

```
initlist : LISTINT is
    do
        !!Result.make(2,Void)
        Result := Result.append(3)
        Result := Result.append(5)                                    5
        Result := Result.append(7)
    end -- initlist
```

Example 3.19

```
printlist(l : LISTINT) is
    local
        m : LISTINT
    do
        from m := l                                                   5
        until m = Void
        loop
            print(m.head); print('%N')
            m := m.tail
        end -- loop                                                   10
    end -- printlist
```

building reusable components does not come for free once you adopt the object-oriented paradigm. Reusability must be a design goal, and enough resources should be allocated for it. Much better designs for list-like classes are presented in Chapter **??**. Still, this simple example will serve us at several places in the following sections.

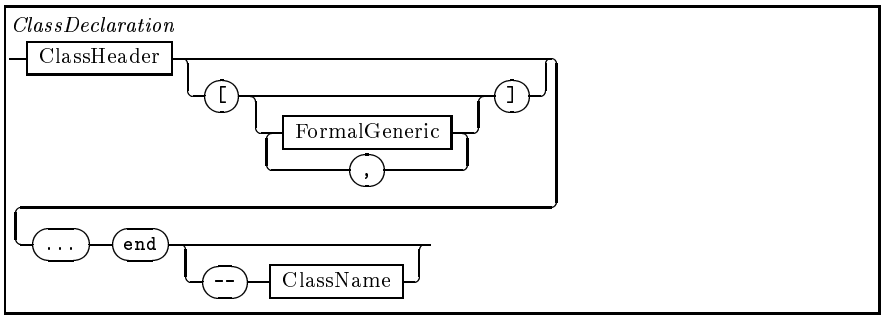
3.2 Genericity

3.2.1 Generic Classes

Building software around implementation of abstract data types (ADTs) such as the class LISTINT is the key to obtaining modular and loosely coupled systems. This technique does not in itself allow the factorization of common behavior for related ADTs (e.g., LISTINT and LISTCHARACTER). Genericity which is available in Eiffel, Ada, or CLU addresses this problem. Genericity is the ability to define modules with parameters, called *generic classes* in Eiffel. A generic class has formal generic parameters represent-

This corresponds to the notion of a class template in C++, except that the Eiffel genericity may be explicitly constrained by a type (see Section 3.7.2).

ing arbitrary types. The syntax of the generic class declaration is given in Syntax Diagram 25.



Syntax Diagram 25: Generic class declaration syntax

We use a French spelling for LISTE *to avoid clashing with the class* LIST *existing in various Eiffel environments.*

Genericity is very useful for classes that store objects (container classes). For example, we can define a generic version of the class LISTINT appearing in Example 3.17 in the previous section: this generic class LISTE[T] now describes a list containing objects of a certain type. This type is the formal generic argument that provides parameters to the class. It is denoted T in the class text of Example 3.20.

A generic class is not directly usable (instantiable), because it is only a class (and a type) pattern.

3.2.2 Generic Class Derivation

To derive a directly usable class from a generic one, you must provide an actual type (i.e., an actual generic parameter) for each formal generic type parameter of the generic class (Example 3.21).

The class LISTE[INTEGER] has exactly the same interface as the class LISTINT from Example 3.17. From a client point of view, they are totally interchangeable.

Genericity is only meaningful in a typed language. In Eiffel, you cannot put anything more than an integer in a LISTE[INTEGER]. In a language such as Smalltalk, there is no way to restrict the types of elements that a list contains, and then genericity would serve no purpose.

Example 3.20

```
indexing
    description: "Simple lisp-like generic linked list"
class LISTE[T]

creation                                                              5
    make
feature
    head: T
    tail: LISTE[T]
    make (new_head: T; new_tail: LISTE[T]) is                        10
            -- make a new list with 'new_head' prepended to 'new_tail'
        do
            head := new_head
            tail := new_tail
        end -- make                                                  15
    append (new_head: T): LISTE[T] is
            -- return a new list with 'new_head' prepended to Current
        do
            !!Result.make(new_head,Current)
        end -- append                                                20
    has (v : T): BOOLEAN is
            -- does the list contain an object equal to v ?
        do
            Result := head.is_equal(v)
                or else (tail /= Void and then tail.has(v))          25
        end -- has
end -- LIST [T]
```

Example 3.21

```
list_of_integers : LISTE[INTEGER]
list_of_characters : LISTE[CHARACTER]
list_of_list_of_integers : LISTE[LISTE[INTEGER]]
```

3.2.3 A Standard Eiffel Generic Class: The ARRAY

Eiffel arrays are finite sequences of generic objects, accessible through integer indices in a contiguous interval. They are instances of the generic kernel library class ARRAY[T]. They are thus regular Eiffel objects. An extract of the interface of the class ARRAY[T] is presented in Example 3.22.

Example 3.22

```
class interface ARRAY [T]
creation
    make
feature -- Initialization
    make (minindex, maxindex: INTEGER)                                    5
    -- Make array empty if minindex > maxindex.
    -- Reallocate if necessary; set all values to default.
        ensure
            no_count: (minindex > maxindex) implies (count = 0)
            count_constraint: (minindex <= maxindex) implies              10
                                    (count = maxindex - minindex + 1)
feature -- Access
    frozen infix "@", frozen item (i: INTEGER) : T
    -- Entry at index i, if in index interval
        require                                                           15
            good_key: valid_index (i)
feature -- Measurement
    count : INTEGER                 -- Number of available indices
    lower : INTEGER                 -- Minimum index
    upper : INTEGER                 -- Maximum index                      20
feature -- Status report
    valid_index (i: INTEGER) : BOOLEAN
    -- Is i within the bounds of the array?
feature -- Element change
    force (v: like item; i: INTEGER)                                      25
    -- Assign item v to i-th entry.
    -- Always applicable: resize the array if i falls out of
    -- currently defined bounds; preserve existing items.
        ensure
            inserted: item (i) = v                                        30
            higher_count: count >= old count
    frozen put (v: like item; i: INTEGER)
    -- Replace i-th entry, if in index interval, by v.
        require
            good_key: valid_index (i)                                     35
        ensure
            inserted: item (i) = v
feature -- Resizing
    resize (minindex, maxindex: INTEGER)
    -- Rearrange array so that it can accommodate indices down to         40
    -- minindex and up to maxindex. Do not lose previously entered items.
        require
            good_indices: minindex <= maxindex
```

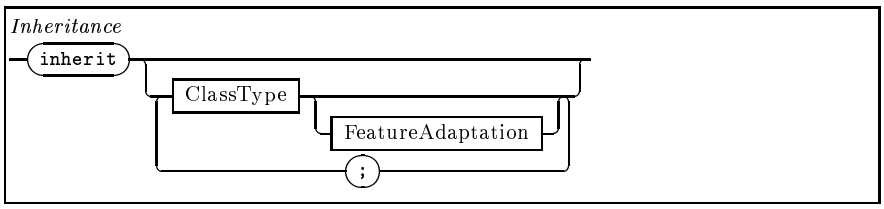
```
feature -- Conversion
    to_c : POINTER                                                    45
            -- Address of actual sequence of values,
            -- for passing to external (non-Eiffel) routines.
invariant
    consistent_size: count = upper - lower + 1
    non_negative_count: count >= 0                                    50
end -- ARRAY [T]
```

An Eiffel ARRAY is dynamic, and can be resized if necessary. This ARRAY class has much in common with the STRING class, which could have been designed as an ARRAY[CHARACTER] augmented with a set of string handling features. As for the class STRING, an entity *a* attached at run time to an object declared of type ARRAY is not the actual sequence of objects but an object that is likely to contain (among others) a reference to the actual contents. Thus, the remarks on assignment-like operations for strings (page 82) also hold for arrays.

3.3 Inheritance

3.3.1 The Dual Nature of Inheritance in Eiffel

Inheritance is a relationship between classes that fosters the definition and implementation of a new class by combination and specialization of existing ones. The new class is then called a subclass (or derived class) of its superclasses (or ancestor classes). The syntax of the inheritance clause is shown in Figure 26.



Syntax Diagram 26: Inheritance clause syntax

Inheritance is characteristic of object-oriented languages. Without it, a language may only be called *object based* (e.g., Ada83 or Modula-2).

Adding inheritance to such languages makes them object oriented (e.g., Ada95, Modula-3).

Inheritance has a dual nature. It provides programming by extension (as opposed to programming by reinvention [24]): a child class extends its parent class by providing more functionality. This is the module view of the class. Inheritance also can be used to represent an is-a-kind-of (or is-a) relationship, thus allowing for *classification*. This gives the type view of the class. The Eiffel type system is based on inheritance; that is, assignment compatibility is defined according to the inheritance relationship.

These two different natures of inheritance are not orthogonal. Most of the time, when you want a class B to be a kind of class A, you also like it to inherit at least a part of the code defined in A. The Eiffel's inheritance allows you to smoothly go from one extreme (the typing side, or interface inheritance, where no code is reused) to the other (the module side, or implementation inheritance, where no part of the interface is reused), or to stay at any intermediate step.

3.3.2 Module Extension

In contrast to genericity, which allows the reuse of closed modules, inheritance allows reusable parts to be customized and combined to build new classes. Inheritance makes a module always open for modification through subclassing. Eiffel gives you full control of this customization through the mechanism of *feature adaption,* which is described in Section 3.4.

As a module extension mechanism, inheritance allows the programmer to reuse a class that is almost what is wanted, and to tailor the class in a way that does not introduce uncontrolled side effects into the other (part of the) software system using the class. It enables an incremental, non disruptive form of programming. If a class B inherits from A, then B has two parts, an *inherited* part and an *incremental* part (see Figure 3.4).

The inherited part provides B with the same services featured by A (unless they are customized; see Section 3.4), whereas the incremental part is the new code, written specifically for B.

Consider for example the class RANDOM_GENERATOR (Example 3.8). If you need random integer (or Boolean) values, you could derive from this class a new class (e.g., MULTI_RANDOM_GENERATOR) having the same features as RANDOM_GENERATOR plus a *last_random_integer* and a *last_random_bit* feature (see Example 3.23).

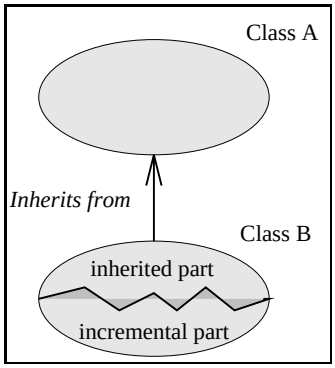


Figure 3.4: Inheritance as a module extension mechanism

3.3.3 Subtyping

Inheritance provides a natural classification for kinds of software objects. It is the software engineering counterpart of the Linnaeus' classification in the natural sciences, or the Bourbaki's systematic efforts to classify mathematical objects. Classification allows us to express and to take advantage of the commonality of objects.

Consider for instance the relationship between a COFFEE_MACHINE and an ESPRESSO_MACHINE. The latter is a specialization of the former: we say that the ESPRESSO_MACHINE is a kind of COFFEE_MACHINE.

When a software system is analyzed and designed using an object-oriented approach, the classifications identified during the analysis are preserved and enriched during design, and then may be directly implemented in code. This seamlessness provides a better continuity between the requirements and the code: a small change in the requirements is likely to produce a small change in the code. Conversely, when a system having already reached the detailed implementation or maintenance phase needs to be modified, it is possible to reflect the changes back to the higher levels of design, specification, and analysis.

In Eiffel, subtyping is defined after subclassing. In the simplest case, if B inherits from A, then B defines a subtype of A. That is, every entity b of type B could be used at any place where an object of type A is expected (substitutability principle). In this case, B is also said to *conform* to A.

Generic classes are special, because they define class (and thus type) templates instead of real classes and types. We have then the following (recursive) rule to define type conformance among generic classes: a generic class B[U] conforms to A[T] only if B conforms to A and U to T. The rule

Example 3.23

```
indexing
    description: "random number generator for REAL, INTEGER and BOOLEAN"
class MULTI_RANDOM_GENERATOR
inherit
    RANDOM_GENERATOR                                                    5
creation
    reset
feature
    last_random_integer(lower,upper:INTEGER) : INTEGER is
            -- Return an evenly distributed random number over the      10
            -- interval [lower, upper[.
        require non_empty_interval: lower<upper
        do
            Result := last_random_real(lower*1.0,upper*1.0).to_integer
        ensure                                                          15
            in_bounds: Result >= lower and Result < upper
        end; -- last_random_integer
    last_random_bit : BOOLEAN is
            -- Return a single random bit.
        do                                                              20
            Result := uniform < 0.5
        ensure
            evenly_distributed: -- Probability of 'Result = True' is 50%.
        end -- last_random_bit
end -- MULTI_RANDOM_GENERATOR                                           25
```

is recursive, because U and T might be generic classes themselves. This conformance rule is easily extended when multiple generic parameters are used.

However, classifications are subjective matters that never quite achieve perfection. Think of the well-known example of ostriches and flying: an ostrich is a bird, a bird is a flying animal, but ostriches don't fly.

> *Classification is humanity's attempt to bring a semblance of order to the description of an otherwise rather chaotic world. (B. Meyer)*

The need to deal with imperfect inheritance structures (where subclasses are not true subtypes) seems universal, and poses several problems that are explored in section **??**.

3.3.4 Inheritance and Expanded Types

The values of entities declared to be of an expanded type are objects rather than references to objects. This is the only consequence of the expansion status of a class, which means that an expanded class may inherit freely from non expanded ones, and conversely. Whether or not the child class is expanded is never decided by parent classes (the expansion status is not inherited) but in the child itself, according to the presence or absence of the keyword **expanded** in its header. Example 3.24 presents the definition of the INTEGER class, which inherits from INTEGER_REF and adds an expanded status.

Example 3.24

```
expanded class INTEGER
inherit
  INTEGER_REF
end -- INTEGER
```

Conversely, Example 3.25 is a possible definition of the MY_INTEGER class, which inherits from INTEGER, and thus wipes out its expanded status.

Example 3.25

```
class MY_INTEGER
inherit INTEGER
feature
  new_feature is
    do                                                    5
      -- something
    end
end
```

3.3.5 Implicit Inheritance Structure

The inheritance structure of an Eiffel system forms a lattice (oriented with the inheritance relation) with a maximal element (the class GENERAL) and a minimal one (the pseudo class NONE); see Figure 3.5. That is, all classes descend from GENERAL and NONE is a descendant of every class.

Any developer-written class (e.g., A, B, C, D in Figure 3.5) without an explicit inheritance clause is considered to directly inherit from the class

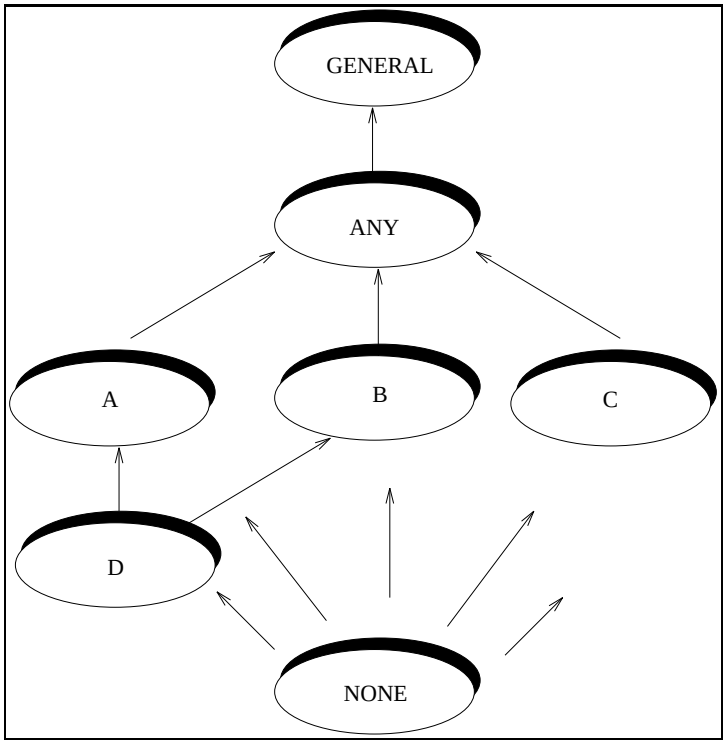


Figure 3.5: Standard Eiffel Inheritance lattice

ANY (as if it included an inheritance clause of the form: inherit ANY). Then by transitivity every class (but GENERAL) inherits from ANY, and thus every type conforms to ANY (hence the name). ANY inherits from GENERAL and may be customized for individual projects or teams, thus providing for project-wide universal properties.

If you remark that a class may inherit from ANY *more than once, don't worry—this is not a problem in Eiffel. It will be explained in Section* ?? *on page* ??.

All the features of ANY are directly available to all Eiffel classes. The features *print*, *clone*, *deep_clone*, *equal*, *deep_equal*, *default_rescue*, *Void*, and *io*, used in several previous examples of Eiffel fragments are some of such universal features that actually belong to the class GENERAL. They are not language keywords, but rather features inherited by all classes through the implicit inheritance link with ANY.

At the other end of the inheritance lattice, the pseudo class NONE is considered to inherit from all classes. The inheritance relation is acyclic, so no class may then inherit from NONE (which is why a feature exported to NONE is private). The other use of NONE is as the type of the feature

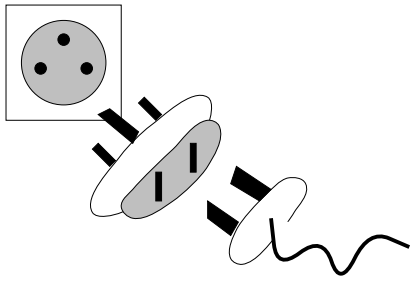


Figure 3.6: Using a plug adaptor

Void (defined in the class GENERAL). *Void* is then a value type-conformant to every class, and is used as the default initialization value for all entities of reference types. The classes GENERAL, ANY, and NONE belong to the Eiffel Library Standard that is described in Section 4.3.

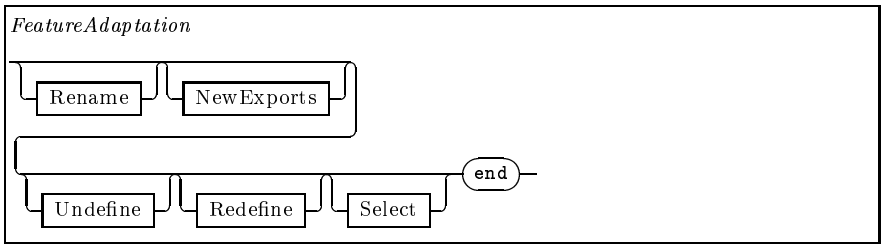
3.4 Feature Adaptation

The reuse of Eiffel classes through inheritance may be customized on a by-feature basis. A useful analogy for this mechanism is the problem of plugging an electrical razor cord into a foreign wall outlet. The solution usually is to use a plug adaptor to convert the size and form of the connectors (and sometimes the voltage or the frequency, or both) of your cord to fit the foreign wall outlet (see Figure 3.6). What is done with this adaptor is functionally equivalent to changing the foreign wall outlet interface. As a client, you then may use the adaptor interface instead of the original one.

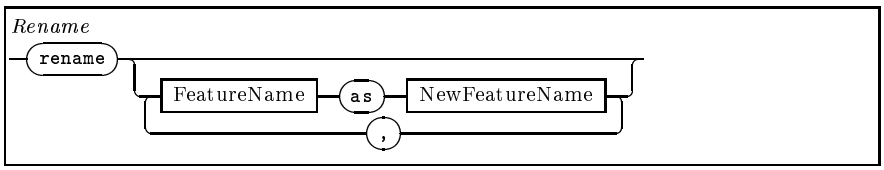
In Eiffel, for each individual feature inherited from a parent class, you may either inherit the feature as it is in the parent class (same name, specification [including signature, preconditions and postconditions], body, and export status [this is the default behavior]), or change any of these components with the mechanisms described in the following sections. The syntax of the feature adaptation is described in Syntax Diagram 27.

3.4.1 Renaming

Renaming is giving a new name to an inherited feature. The syntax of the *rename* subclause of the *inheritance* clause is described in Syntax Diagram 28. This subclause is useful to present a more convenient interface to clients (for example, if the class VECTOR inherits from ARRAY, the feature *count* might be renamed *dimension*). Explicit renaming is also the



Syntax Diagram 27: Feature adaptation



Syntax Diagram 28: Rename clause syntax

mechanism used to solve name conflicts in multiple inheritance. Consider for instance a class VISUAL_RANDOM_BAG inheriting from the class RANDOM_BAG, which has a feature *draw*, and from the class PICTURE which also has a feature *draw*. We thus have a name conflict for the feature *draw* for objects of type VISUAL_RANDOM_BAG. In contrast to some artificial intelligence–based languages using sophisticated heuristics to solve such name clashes, Eiffel requires you to do it explicitly through the renaming of one of the conflicting versions, as in Example 3.26.

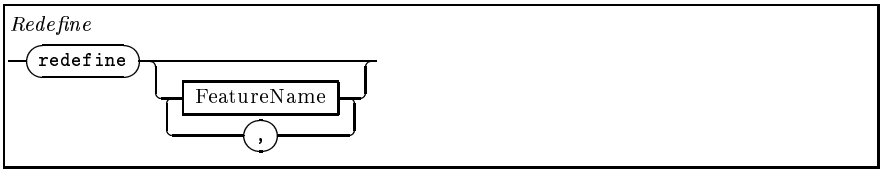
3.4.2 Redefining

Redefining the feature is changing its specification or body, or both. If you want to redefine a feature, you must declare it in the *redefine* subclause of the *inheritance* clause.

For example, the process by which the coffee is produced is different depending on whether you have a COFFEE_MACHINE or an ESPRESSO_MACHINE. If the class ESPRESSO_MACHINE inherits from the class COFFEE_MACHINE, the *deliver_coffee* routine must be redefined, as shown in Example 3.27. This change is, however, constrained to respect

Example 3.26

```
class VISUAL_RANDOM_BAG
inherit
    RANDOM_BAG
        rename
            draw as random_draw                                    5
        end
    PICTURE
creation
    ...
end -- VISUAL_RANDOM_BAG                                          10
```



Syntax Diagram 29: Redefine clause syntax

Example 3.27

```
class ESPRESSO_MACHINE
inherit
    COFFEE_MACHINE
        redefine
            deliver_coffee, needs_maintenance                      5
        end
feature
    deliver_coffee is
            -- pilot the hardware to deliver a cup of espresso
        do                                                        10
            -- new method to produce the espresso
        end
    needs_maintenance : BOOLEAN is
            -- condition upon which some maintenance is needed
        do                                                        15
            -- new criteria to evaluate if maintenance is needed
        end
end -- ESPRESSO_MACHINE
```

the semantics of the original feature. The signature must be covariantly compatible with the original, the precondition can only be weakened, and the postcondition must be strengthened.

Covariant Signature Redefinition

The implications of the type system of the covariance rule for redefinitions are discussed in Section ?? on page ??.

Syntactically the covariant signature redefinition constraint translates to the following rule. If the original version of a feature *f* takes an argument of type A and/or returns a type A (if the feature is an attribute or a function), then the redefined version may only take an argument of type B (or returns a type B), such that B is a descendant of A or A itself. The inheritance varies for both in the same direction, hence the name *covariance*.

Weakening Preconditions

If the precondition is to be changed, the new precondition follows the syntax:

```
require else new_cond
```

The precondition of the redefined routine is actually the new_cond or else the precondition of the original routine.

Strengthening Postconditions

In the same spirit, the postcondition clause of a redefined routine must follow the syntax:

```
ensure then new_cond
```

The postcondition of the redefined feature is actually the new_cond and then the postcondition of the original routine. These constraints on a feature redefinition provide the necessary semantics to ensure the safety of the Eiffel inheritance mechanism according to most recent works in this domain [27].

These constraints leave open the possibility of redefining a function without parameters to become an attribute (but not conversely).

3.4.3 Anchored Declarations

Covariance redefinition is used frequently in the Eiffel world, so a mechanism called *anchored declaration* has been designed to save a considerable amount of tedious redeclarations when dealing with signature redefinitions.

An anchored declaration has a syntax as shown in Syntax Diagram 30.

Syntax Diagram 30: The anchored declaration

The *Anchor* is either **Current** or an attribute. The meaning of the declaration `x : like anchor` is that whenever the anchor is redeclared in a descendant class, `x` follows automatically. Consider for example the feature *is_equal*, as defined in the root of the Eiffel class hierarchy, the kernel library class GENERAL. This anchored definition (see Example 3.28) allows

Example 3.28

```
is_equal (other: like Current): BOOLEAN is
        -- Is 'other' attached to an object considered equal to 'Current'?
    require
        other_not_void: other /= Void
    ensure                                                              5
        symmetric: Result implies other.is_equal (Current)
    end
```

an automatic redeclaration of the signature of the feature *is_equal* in any class FOO to

```
is_equal (other: FOO): BOOLEAN
```

This *is_equal* feature may be redefined to fit special equality semantics, whereas the feature *equal* (see Section 2.4.1) may not. For example, the sets $A = \{1, 2, 3\}$ and $B = \{1, 3, 2\}$ are mathematically equal, but not Eiffel $equal(A, B)$. The feature *is_equal* may be redefined in a class SET in such a way that *A.is_equal(B)*.

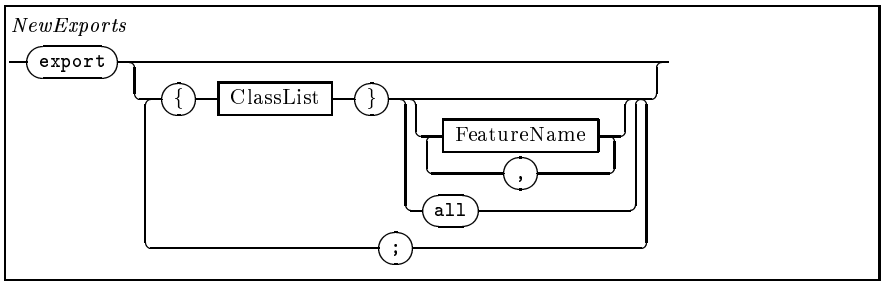
3.4.4 Changing the Export Status

The export status of an inherited feature may be changed in several ways with the *export* subclause of the *inheritance* clause.

With this mechanism, you may either:

- Extend the export status of a set of features, or make them available to more client classes than in the parent class ("ANY" means visible by all classes),

The implications type of system of this *are discussed in Section ?? on page ??.*



Syntax Diagram 31: The "new export" syntax

- Reduce it, hiding a set of features to some clients (or even to all clients if NONE is used). This option may be useful if these features are inherited for implementation purposes only.

Consider for example a class LEASED_COFFEE_MACHINE representing coffee machines leased to special clients (instances of a class called LEASEHOLDER), who pay a rental fee, but get in exchange the money deposited in the LEASED_COFFEE_MACHINE. It is then possible for a LEASEHOLDER to get access on the feature *cash* with the declaration shown in Example 3.29. Any feature not specified in the new *export* clause keeps the exact export

Example 3.29

```
class LEASED_COFFEE_MACHINE
inherit
    COFFEE_MACHINE
        export
            {MANAGER,LEASEHOLDER} cash                    5
        end
end -- LEASED_COFFEE_MACHINE
```

status it had in the superclass. The special keyword **all** counts for all features of the superclass.

3.4.5 Other Feature Adaptations

You may also:

- Undefine a feature (*undefine* clause), which is mainly useful for choosing one of a set of competing implementations for a feature name in some cases of multiple inheritance (see Section 3.6.3),
- Select a feature (*select* clause) as the target for dynamic binding when there are ambiguities in some cases of repeated inheritance (see Section ?? on page ??).

3.5 Polymorphism and Dynamic Binding

3.5.1 Polymorphic Entities

Polymorphism is defined in Webster's dictionary as "the quality or state of being able to assume different forms" [28]. Polymorphic referencing is the way inheritance polymorphism appears in Eiffel. The association of a reference with an object is constrained by the type conformance rule (see Section 3.3.3): an entity x declared as being of a non expanded type T can be used to refer to an object of type S, provided the class S is a descendant of the class T. The entity x is then said to be of *static* type T, and capable of assuming the *dynamic* type S. More generally, an attribute declared as being of static type T can be used to refer to any object with a dynamic type that *conforms to* type T.

Expanded types are dealt with in Section 3.5.3.

Consider the declaration in Example 3.30.

Example 3.30

```
e : T          -- T being a non expanded type
```

This declaration specifies that the entity e may only be *Void* or attached to objects conforming to T, that is instances of T itself or subclasses of T. During its lifetime, a non expanded entity may refer to various objects, not necessarily having the same type (provided they conform to the entity static type), hence its polymorphic aspect. On the contrary, an expanded type entity is not polymorphic, because it is really an object and not a reference to an object. In other words, objects are not polymorphic, only references are.

The *dynamic type* of an entity is the type of the object it refers to at a given point in its lifetime. A non expanded entity may acquire a new dynamic type either:

- Through a creation instruction with the third form of object creation described in Section 3.1.1 on page 74. For example, if the class S is

a descendant of the class T, the creation instruction of Example 3.31 creates an object of type S, attaches it to e, and initializes it with the creation procedure *make*,

Example 3.31

```
!S!e.make        -- S being a subclass of T
```

- Or through any assignment instruction, including actual to formal mapping of routine parameters and assignment attempt (see Section ??). In Example 3.32, d is a reference to an object of type S and r is a routine with a formal argument e of type T.

Example 3.32

```
e := d   -- e is now attached to an object of type S
r(d)     -- inside r, the formal argument e is attached
         -- to an object of type S
```

In both cases, the static type of the entity e is T, and its dynamic type is S.

3.5.2 Dynamic Binding

Consider a system in which instances of the classes COFFEE_MACHINE (Example 3.10) and ESPRESSO_MACHINE (Example 3.27) coexist. Let be m an entity declared of type COFFEE_MACHINE that may assume the dynamic type ESPRESSO_MACHINE at some step of the program execution. Assume that m is asked to deliver more coffee (`m.deliver_coffee`). The problem is that we cannot always know at compile time which "*deliver_coffee*" feature must be called on m—is it the one defined in COFFEE_MACHINE or in ESPRESSO_MACHINE?

The rule known as *dynamic binding* states that the dynamic type of an entity determines which version of the operation is applied. Dynamic binding allows the choice of the actual version of a feature to be delayed until run time. Whenever more than one version of a feature might be applicable, it ensures that the most directly adapted to the target object is selected. The static constraint on the entity's type ensures that there is at least one such version.

Dynamic binding of routines to entities is the default rule in Eiffel. It is

In C++ only virtual methods are subject to dynamic binding.

a run-time mechanism (basically a table lookup) that is a priori more costly than a simple procedure call. In modern Eiffel compilers, the appropriate routine is always found in constant time, whatever the complexity of the inheritance hierarchy. This time overhead tends to be small (or even negligible) for real applications. Furthermore, there are two cases in which the dynamic-binding mechanism may be bypassed in favor of a static binding:

- If a feature is declared to be **frozen**, it may not be redefined in subclasses. Dynamic binding is thus unnecessary.
- When compiling an Eiffel system (that is, a set of classes needed to produce an executable program), the compiler may become aware that a feature is never redefined in used subclasses, or may statically know the dynamic type of an entity (e.g., through data flow analysis). In both cases it is well founded to replace the dynamic binding of the relevant features with a mere procedure call (or even its in-line expansion).

More details about compiler technology and performances of an Eiffel program are given in Section **??** (Implementation Efficiency). Dynamic binding bypassing remains a compiler optimization that is transparent to the Eiffel programmer.

3.5.3 Type Conformance and Expanded Types

As discussed in the previous section, entities that denote expanded types are not polymorphic, because they are only *values*. If x is an entity of an expanded type A, only expanded or regular instances of class A may be assigned to x. Consider the set of declarations in Example 3.33 (with the definitions of INTEGER and MY_INTEGER Examples 3.24 and 3.25).

Example 3.33

```
i : INTEGER_REF      -- reference to an integer value
j : INTEGER          -- an integer value
k : MY_INTEGER       -- reference to an integer
                     -- (MY_INTEGER inherits from INTEGER)
```

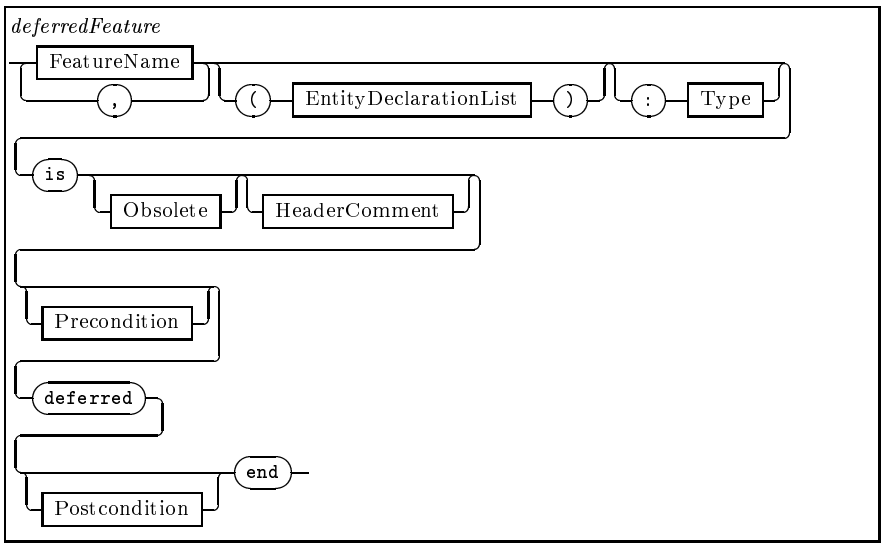
The assignment `i := j` is legal, and involves a copy of the value of j in i. On the other hand, the assignment `j := k` would not be legal, because although MY_INTEGER is a subclass of INTEGER, k does not conform to INTEGER, because INTEGER is of an expanded type.

An interesting consequence of this limitation is that operations involving expanded types such as INTEGER or REAL may be implemented as efficiently as in procedural languages, because they are never subject to dynamic binding.

3.6 Deferred Classes

3.6.1 Deferred Routines

A *deferred class* is a class with at least one *deferred feature*, (that is, a feature with an implementation that is left unspecified). The syntax to declare a deferred feature is presented in Syntax Diagram 32.



Syntax Diagram 32: Deferred Feature syntax

A deferred feature is equivalent to a pure virtual function in C++.

This deferred feature has a specification (a name, a signature, preconditions and postconditions) but no implementation. By opposition, a non deferred feature is called an effective feature (it has a specification *and* an implementation). An attribute may not be deferred, because it already has an implementation (formally the function returning its value).

Example 3.34

```
infix "<" (other : like Current) : BOOLEAN is
        -- Is 'Current' less than 'other'?
    require
        other_not_void : other /= void
    deferred
    ensure
        asymmetric : Result implies not (other < Current)
    end
```

3.6.2 Deferred Classes

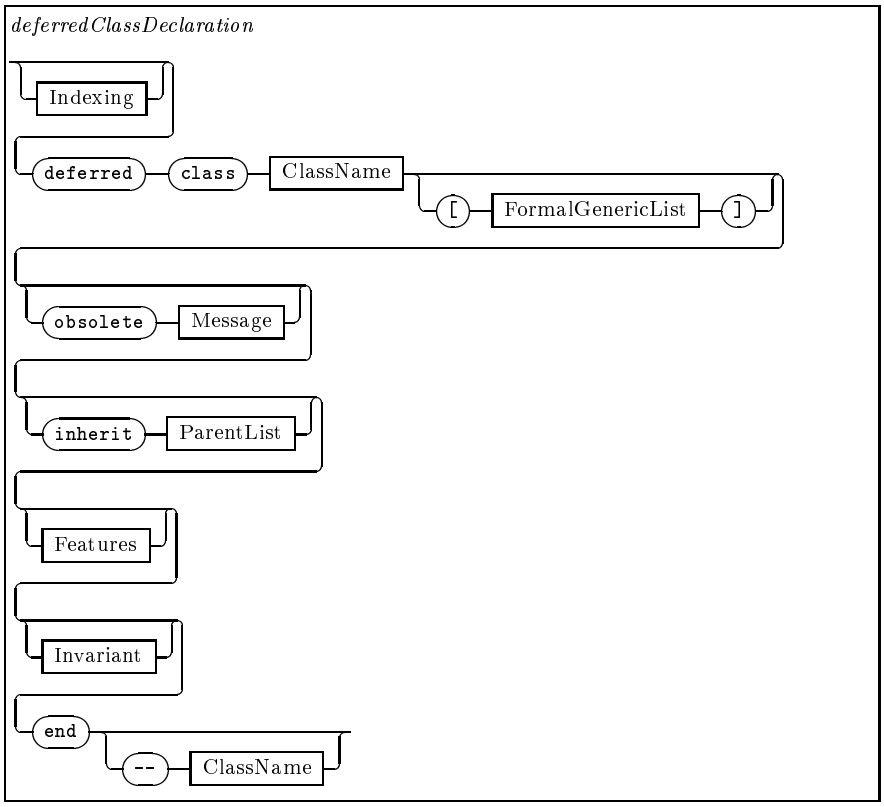
As soon as a feature is left deferred, the enclosing class must be declared deferred. The deferred class syntax is presented in Syntax Diagram 33.

Conversely, a deferred class must have at least one deferred feature. Consider for example the kernel library class COMPARABLE encapsulating the notion of objects that may be compared according to a total order relation. It has a deferred function (*infix* " $<$") and a number of effective features defined after this function.

For the sake of conciseness, preconditions and postconditions are mostly omitted in this listing.

Example 3.35

```
indexing
    description: "Objects that may be compared according to a
        total order relation"
    note: "descendants need only define the behavior of infix <"
deferred class COMPARABLE
feature
    infix "<" (other : like Current) : BOOLEAN is
        -- Is 'Current' less than 'other'?
    require
        other_not_void : other /= void
    deferred
    ensure
        asymmetric : Result implies not (other < Current)
    end
    infix "<=" (other: like Current): BOOLEAN is
        -- Is current object less than or equal to 'other'?
    do
        Result := not (other < Current)
    end
    infix ">" (other: like Current): BOOLEAN is
```



Syntax Diagram 33: Deferred class Syntax

```
        -- Is current object greater than 'other'?
    do
        Result := other < Current
    end
infix ">=" (other: like Current): BOOLEAN is                    25
        -- Is current object greater than or equal to 'other'?
    do
        Result := not (Current < other)
    end
min (other : like Current) : like Current is                    30
        -- The smaller of 'Current' and 'other'.
    do
```

```
            if Current < other then
                Result := Current
            else
                Result := other
            end -- if
        end
    max (other : like Current) : like Current is
            -- The larger of 'Current' and 'other'.
        do
            if other < Current then
                Result := Current
            else
                Result := other
            end -- if
        end
invariant
    irreflexive_comparison: not (Current < Current)
end -- class COMPARABLE
```

This is related to the notion of abstract class found in various object-oriented language.

Whereas a class is the implementation of an ADT, a deferred class is a partial implementation of such an ADT (or even the ADT itself). Hence a deferred class may not be instantiated (see Example 3.36). It merely describes the common properties of a group of classes descendant of it. A deferred class may not have a **creation** part. Still, a deferred class defines a type (or a type pattern if it is a generic deferred class), and entities may be declared with this type.

Example 3.36

```
x : COMPARABLE  -- legal
!!x             -- illegal: COMPARABLE may not be instantiated
x := "Hello world" -- legal: STRING inherits from COMPARABLE
```

Deferred classes are useful in structuring systems (see Section 3.6.4), but ultimately only their effective subclasses may be instantiated.

3.6.3 Inheritance and Deferred Classes

Beyond the feature adaptation mechanisms (renaming, exportation status, and redefinition; see Section 3.4) that are available for all kinds of features, a child class can change the deferred status of its parent features.

Effecting a Routine

This operation provides the inherited deferred feature with an implementation. No new syntax is required. The feature only needs to be present in a feature clause of the child class, with the same specification (name, signature, precondition, and postcondition) as inherited from its parent and with an implementation. Like any other function with no parameter, a deferred function may be implemented as an attribute.

If the deferred feature specification needs to be changed in the child class, the redefinition mechanism (as described in section 3.4.2) must be used instead. Since routine overloading (*à la* Ada or C++) is not available in Eiffel, if you mess up the feature signature, the compiler tells you what to do (either redefine the feature to take into account the new signature or correct it if it was a typo).

Merging Features through undefinition

Undefining is possible provided the routine was not **frozen**.

Undefining a feature allows a child class to wipe out the implementation of an effective routine (but not of an attribute) inherited from one of its parents. The syntax of an undefine clause is similar to that of a redefine clause, and just precedes it in the feature adaptation clause (Syntax Diagram 27).

Syntax Diagram 34: Undefine

Example 3.37

```
deferred class CHILD
inherit
    PARENT
        undefine f
    end                                                    5
...
```

In this example, the feature *f* becomes a deferred feature in the class CHILD (which makes it a deferred class) while keeping the specification

(name, signature, and preconditions, and postconditions) it had in the class PARENT.

The usefulness of this undefining mechanism mainly concerns the merging of multiple inherited features, because a merging is valid only if at most one merged feature is effective.

3.6.4 Deferred Classes: A Structuring Tool

Deferred classes may be used to factor out common properties of a set of classes into *abstract classes*. Some abstract classes appear naturally in the application domain, whereas other abstract classes are artificially manufactured as a convenient means for promoting code reuse. Consider for example the problem (discussed in depth in Chapter ??) of designing a linear algebra library around the notions of matrices and vectors.

A fundamental principle applied when designing this library is that the abstract specification of an entity is dissociated from any kind of implementation detail. Although all matrices, for example, share a common abstract specification, they do not necessarily require the same implementation layout. Obviously dense and sparse matrices deserve different internal representations. The same remark goes for vector entities.

The classes MATRIX and VECTOR are deferred classes: they provide no details about the way matrices and vectors shall be represented in memory. The specification of their internal representation is thus left to descendant classes. This does not imply that all features are kept deferred. Representation-dependent features are simply declared, whereas other features are defined—i.e., implemented—directly in MATRIX and VECTOR, as shown in Chapter ??.

3.7 Genericity and Inheritance

Genericity and inheritance are orthogonal concepts in Eiffel [29]. It is then interesting to see how they can be combined to foster versatile new possibilities for reusing software.

3.7.1 Heterogeneous Containers

A container class is a class that is able to store generic elements. Examples are the classes LIST[T], STACK[T], and ARRAY[T], etc. found in most Eiffel data structure libraries. This constraint is very difficult to express in a typeless language like Smalltalk. There is no simple way to forbid a particular kind of object from going into containers.

Back to our coffee machine example, imagine that a number of such machines are to be monitored to determine whether they need maintenance. We could design a class MANAGER (as in Example 3.38) that has a list of coffee machines (LISTE[COFFEE_MACHINE]; see Example 3.20). The ESPRESSO_MACHINE is a kind of COFFEE_MACHINE, so one can store both kinds of coffee machines in the list. It is thus a heterogeneous container, restricted to contain objects that are instances of COFFEE_MACHINE (or subclasses of COFFEE_MACHINE).

Example 3.38

```
class MANAGER
feature
    machines : LISTE[COFFEE_MACHINE]
    add (new : COFFEE_MACHINE) is
        require                                                    5
            exist: new /= Void
        do
            if machines = Void then
                !!machines.make(new,Void)
            else                                                   10
                machines := machines.append(new)
            end -- if
        end -- add
    total_needing_maintenance : INTEGER is
        local m : like machines                                    15
        do
            from m := machines
            until m = Void
            loop
                if machines.head.needs_maintenance then            20
                    Result := Result + 1
                end -- if
                m := machines.tail
            end -- loop
        end -- total_needing_maintenance                           25
end -- MANAGER
```

Thanks to dynamic binding, the right version of the feature *needs_maintenance* is applied to each element of the array in order to compute the total number of coffee machines that require maintenance. This routine (lines 14–25 of Example 3.38) may be written *without precise knowledge* of which objects really are in the heterogeneous container, provided

they conform to COFFEE_MACHINE.

In a procedural language (such as C, Pascal, or Ada83), this function would have been written using a case statement discriminating among case selector values (encoding on the actual type of the considered object) to make the correct procedure call (the one from either COFFEE_MACHINE or ESPRESSO_MACHINE). The same kind of processing then would be repeated for each routine dealing with the heterogeneous container. The object-oriented solution, here based on the dynamic binding of the routine *needs_maintenance*, makes it possible to avoid such a duplication of code.

Once again, much more than the savings obtained during the design and the coding, the real savings appear during the maintenance phase of the software. Imagine that at this stage (maybe 10 years after the software is running in production mode), a new kind of COFFEE_MACHINE is adopted (e.g., a CAPPUCCINO_MACHINE), with a new means to compute its *needs_maintenance*.

With the procedural language solution, we would have to locate and modify every routine dealing with the explicit type of a COFFEE_MACHINE (e.g., *needs_maintenance*). Then every modified module would have to be retested (both for the new functionality and for nonregression).

In contrast, in the object-oriented solution, once the CAPPUCCINO_MACHINE class has been implemented and tested, no other part of the software has to be modified (except perhaps for the routine dealing with the initialization of the container). Most notably, the class MANAGER does not need to be changed, and hence retested. The maintenance phase is clearly where the object-oriented approach is the big winner.

3.7.2 Constrained Genericity

The other mechanism for combining genericity and inheritance is constrained genericity, which makes it possible to specify that a generic parameter must be a descendant of a certain class. For example, a generic MATRIX class would require its generic parameter T to be a descendant of the NUMERIC class with the declaration:

classMATRIX [T− >NUMERIC]

If the NUMERIC class features an infix "+" operation, then a generic MATRIX addition operation could be defined, based on the addition of the individual elements of the MATRIX.

The notion of constrained genericity encompasses the previously described notion of genericity (Section 3.2), which is actually an abbreviation for genericity constrained by ANY. Thus a LISTE[T] is equivalent to a LISTE[T − > ANY]. . This constrained genericity mechanism enables the generic definition of powerful abstraction, such as a SORTED_LIST[T − >

The class COMPARABLE *has been presented in Example 3.35*

Example 3.39

```
feature
    add (other: MATRIX [T]) is
            -- add other to Current matrix
        require
            not_Void: (other /= Void);                                    5
            same_size: (nrow = other.nrow) and (ncolumn = other.ncolumn)
        local
            i, j : INTEGER
        do
            from i := 1                                                   10
            until i > nrow
            loop
                from j := 1
                until j > ncolumn
                loop                                                      15
                    put(item(i,j) + other.item(i,j), i, j)
                    j := j + 1
                end -- loop on j
                i := i + 1
            end -- loop on i                                              20
        end -- add
```

COMPARABLE]—that is, a sorted list of generic elements. The mere notion that the list is sorted implies that its elements can be compared, hence the constrained genericity.

Another example is to define an ARRAY in such a way that it can be sorted (e.g., to encapsulate the *quicksort* algorithm presented in the Example 2.24 of Section 2.6).

Some examples of the use of this mechanism are described in the Keyword-in-context (KWIC) index problem presented in Section 3.8.

3.8 Case Study: The KWIC System

We now illustrate the concepts introduced in this chapter by considering the KWIC index problem, inspired by R. Wiener [41] or B. Liskov [26]. It is a well-known case-study in the software engineering literature, so the reader is advised to compare the Eiffel solution to this problem to the CLU or the Ada83 one.

Example 3.40

```
class SORTABLE_ARRAY [T -> COMPARABLE]
inherit
  ARRAY [T]
creation
   make -- inherited from ARRAY                                          5
feature
   -- dealing with the sortable properties of the array
   is_sorted : BOOLEAN is
         -- is the array sorted
      do                                                                 10
         Result := is_sorted_range(lower, upper)
      end -- is_sorted
   quick_sort is
         -- sort the array in nondescending order in O(n log(n)).
      do                                                                 15
         if lower < upper then -- at least 2 items
            quick_sort_range(lower, upper)
         end -- if
      ensure
         sorted: is_sorted                                               20
      end -- quick_sort

   quick_sort_range(first, last: INTEGER) is
         -- sort elements first..last into increasing order
         -- (quick sort algorithm)                                       25
      require range_not_empty: first <= last
   -- etc.
end -- class SORTABLE_ARRAY
```

3.8.1 Presentation of the KWIC System

A KWIC index is a list of titles of books, research articles, and so on, arranged so that each title that contains a "key" word can be found easily. Associated with the title is the inventory number of the book. For example, consider the following titles:

```
The Hitch-Hiker's Guide to the Galaxy G. Adams 4242
Software Engineering with Ada and Modula-2 R. Wierner and R. Sincovec 6543
Object-Oriented Software Engineering with Eiffel J.-M. Jezequel 6789
The Evolution of the Universe: Part 1 C. Charlie 9834
```

The KWIC index for this list with the "key" words in the first column would be:

```
Ada and Modula-2                                Software Engineering with   6543
Eiffel                          Object-Oriented Software Engineering with   6789
Engineering with Ada and Modula-2                                Software   6543
Engineering with Eiffel                          Object-Oriented Software   6789
Evolution of the Universe: Part 1                                     The   9834
Galaxy                                     The Hitch-Hiker's Guide to the   4242
Guide to the Galaxy                                   The Hitch-Hiker's   4242
Hiker's Guide to the Galaxy                                   The Hitch-   4242
Hitch-Hiker's Guide to the Galaxy                                     The   4242
Modula-2                                  Software Engineering with Ada and   6543
Object-Oriented Software Engineering with Eiffel                          6789
Oriented Software Engineering with Eiffel                          Object-  6789
Software Engineering with Ada and Modula-2                                  6543
Software Engineering with Eiffel                             Object-Oriented   6789
Universe: Part 1                                  The Evolution of the      9834
```

In each title, words that are articles, prepositions, or trivial are called *nonkeywords*. In the KWIC index, each title appears as often as there are keywords matching it. The titles are aligned so that all the keywords occur in the first column. The portion of the title that appears before the keyword has been shifted to the end of the line. The inventory number of the title is printed to the right of the title.

Such a KWIC index is useful in finding books and articles. To find titles on a particular subject, you just have to search the KWIC index for keywords related to the subject and get the corresponding title by reading first the right part, and then the left one. You also have the inventory number. A typical KWIC index may contain thousands of titles.

3.8.2 The KWIC Object-Oriented Software

A software system to produce a KWIC index is given book descriptions (basically a title, list of authors, and inventory number, see the class BOOK of Example 2.2 on page 36) and the list of nonkeywords as input. The system identifies possible keywords and creates entries for the KWIC index, alphabetizes the entries according to the keywords, and then prints the KWIC index. The list of nonkeywords may be modified from run to run by adding or deleting words.

An object-oriented analysis (presented in [41]) allows the identification of the following classes:

BOOK as described in Example 2.2.

KWIC_ENTRY This class is basically a line of the KWIC index.

KWIC This class is made of KWIC entries. It can be built after a list of books and a list of nonkeywords.

WORDS This class is a representation of a book title broken into an iterable sequence of words.

DRIVER The user interface to the KWIC.

In the following, we only present the core of a solution: the classes KWIC_ENTRY, KWIC and WORDS, and a simple class DRIVER. These classes only use classes from the Eiffel Standard Library (e.g., ARRAY, STRING) and classes previously defined in this book (e.g., LISTE, SORTABLE_ARRAY), so they should run in any existing Eiffel environment. Still, the reader is welcome to complete the system to get more insight into his or her Eiffel environment, by using a HASH_TABLE of nonkeywords instead of a LISTE (for evident efficiency reasons) or by designing a real user interface.

3.8.3 The Class KWIC_ENTRY

The KWIC_ENTRY is made of a book title broken into two parts around a cutting point. These two parts are called *left* and *right* (see the 9^{th} line in Example 3.41). We want to produce an alphabetized list of the KWIC entries, so they should be comparable to each other. Hence KWIC_ENTRY inherits from COMPARABLE (5^{th} line), and we have to define the *infix* "<" operator according to a lexicographical order (lines 27–34 in Example 3.41).

Example 3.41

```
indexing
    description: "a line of the KWIC index"
class KWIC_ENTRY
inherit
    COMPARABLE                                                          5
creation
    make
feature
    left, right: STRING -- left and right part of the kwic entry
    inventory : INTEGER -- inventory number of the corresponding book  10
    make (a_title : STRING; cutting_point, invent : INTEGER) is
            -- kwic entry with the title broken around the cutting_point
        require
            not_void: a_title /= Void
            positive_cutting_point: cutting_point > 0                  15
            bounded_cutting_point: cutting_point <= a_title.count
        do
            inventory := invent
```

```
            left := a_title.substring(cutting_point,a_title.count)
            if cutting_point > 1 then                                    20
                right := a_title.substring(1,cutting_point-1)
            else
                !!right.make(0)          -- empty right part
            end -- if
        ensure                                                           25
            -- right prepended to left is equal to a_title
        end -- make
    infix "<" (other : like Current) : BOOLEAN is
            -- does this entry alphabetically precede other?
        require else                                                     30
            not_void: other /= Void
        do
            Result := (left < other.left)
                or else ((left.is_equal(other.left)) and right < other.right)
        end -- infix "<"                                                 35
end -- KWIC_ENTRY
```

3.8.4 The Class KWIC

The class KWIC is made of KWIC entries that may be alphabetized. We will implement it as a SORTABLE_ARRAY[KWIC_ENTRY] (6^{th} line in Example 3.42) that features a *quick_sort* procedure (see Example 3.40). A KWIC index is built after a list of books and a list of nonkeywords, so its creation procedure (*make*) accepts such parameters (13^{th} line). It stores the nonkeyword list in a private attribute (called *trivial_words*) , and then for each book in the *library* book list, it makes the corresponding KWIC entries (lines 18–23).

The procedure *make_entries_from* (lines 25–45) creates as many KWIC entries as there are keywords matching words in the book title. It is based on a loop asking for successive words of the title and checking whether their lowercase forms are keywords. If so, the corresponding KWIC entry is made. The last procedure, *print_index*, is just a loop for printing all KWIC entries in the format defined in Section 3.8.1.

Example 3.42

```
indexing
    description: "Collection of KWIC entries"
    implementation: "sortable array, exporting the feature quick_sort"
class KWIC
```

```
inherit                                                                    5
    SORTABLE_ARRAY[KWIC_ENTRY]
        rename
            make as array_make
        end
creation                                                                   10
    make
feature
    make (library: LISTE[BOOK]; non_keywords: LISTE[STRING]) is
        local l : like library
        do                                                                 15
            trivial_words := non_keywords
            array_make(1,0) -- initialized to empty
            from l := library
            until l = Void
            loop                                                           20
                make_entries_from(l.head)
                l := l.tail
            end -- loop
        end -- make
    make_entries_from(a_book: BOOK) is                                     25
            -- make the KWIC entries corresponding to a book title
        require exist: a_book /= Void
        local
            new_entry : KWIC_ENTRY
            seq : WORDS -- the sequence of words in the book title         30
            w : STRING  -- the current word of the sequence
        do
            from !!seq.init(a_book.title) -- initialize the sequence
            until seq.off
            loop                                                           35
                w := seq.word           -- get the current word
                w.to_lower              -- convert to lowercase
                if not trivial_words.has(w) then -- it is a keyword
                    !!new_entry.make(a_book.title,seq.start_position,
                                     a_book.inventory)                     40
                    force(new_entry,upper+1) -- append the new entry
                end -- if
                seq.next -- advance the sequence of words
            end -- loop
        end -- make_entries_from                                           45
    print_index is
        -- print the KWIC index, with the "key" words in the first column
        local
            i, j : INTEGER
```

```
        do                                                              50
            from i := lower
            until i > upper
            loop
                print(item(i).left)
                from j := item(i).left.count                            55
                until j > 70 - item(i).right.count
                loop
                    print(' ')
                    j := j + 1
                end -- loop                                             60
                print(item(i).right)
                print("  "); print(item(i).inventory); print("%N")
                i := i + 1
            end -- loop
        end -- print_index                                              65
feature {NONE} -- private features
    trivial_words : LISTE[STRING]
end -- KWIC
```

3.8.5 The Class WORDS

The class WORDS is an iterable sequence of the words present in a STRING. A *word* is a substring made of the letters *a* to *z* and *A* to *Z* only (see the function *is_letter*, lines 51–55 of Example 3.43).

For its implementation, it simply keeps a reference (*ref*) on the original string. The sequence must be initialized with the creation procedure *init*. It may be iterated with the procedure *next*, which advances to the next word (feature *word*) in the string.

Example 3.43

```
indexing
    description: "An iterable sequence of words in a STRING"
    implementation: "keeps a reference on the original string"
class WORDS
creation                                                                5
    init
feature
    ref : STRING
    init (s : STRING) is
            -- init the iterable sequence of words                     10
        require
```

```
        exist: s /= Void
    do
        ref := s
        next -- set the sequence on the first word
    end -- init
word : STRING is
        -- a copy of the current word in the sequence
    require
        not_off: not off
    do
        Result := ref.substring(start_position,end_position)
    end -- word
off : BOOLEAN is
        -- is the sequence of words exhausted?
    do
        Result := start_position > ref.count
    end -- off
next is
        -- advance to the next word in ref
    require not_off: not off
    do
        from start_position := end_position + 1
        until off or else is_letter(ref.item(start_position))
        loop
            start_position := start_position + 1
        end -- loop
        if not off then
            from end_position := start_position
            invariant on_letter: is_letter(ref.item(end_position))
            variant ref.count - end_position + 1
            until end_position = ref.count or else
                not is_letter(ref.item(end_position + 1))
            loop
                end_position:= end_position+ 1
            end -- loop
        end -- if
    ensure
        progress:  start_position > old(end_position)
    end -- next
is_letter(c:CHARACTER): BOOLEAN is
        -- is c a lowercase or uppercase letter, a-z or A-Z ?
    do
        Result := (c>='a' and c<='z') or (c>='A' and c<='Z')
    end -- is_letter
start_position : INTEGER -- of the current word within ref
```

```
    end_position : INTEGER -- of the current word within ref
invariant
    ref_exist: ref /= Void
    end_after_start: (not off) implies end_position >= start_position
    start_on_letter: (not off) implies is_letter(ref.item(start_position))
    end_on_letter: (not off) implies is_letter(ref.item(end_position))
    next_not_letter: end_position < ref.count implies
                          not is_letter(ref.item(end_position+1))
end -- WORDS
```

3.8.6 The Class DRIVER

The class DRIVER is a simple example to exercise the KWIC problem. A real driver to this problem should provide a real user interface, deal with a book database, and have an intelligent way to manage the list of nonkeywords. In Section 4.3.3 we illustrate the input/output facilities available in the Eiffel Standard Library with a modified version of this class to deal with data read from a disk or standard input.

Instead, Example 3.44 uses hard-coded data to produce the KWIC index listing of Section 3.8.1. Despite the high level of the language, the generated code is kept small and efficient; e.g., a stripped version of the executable code for this KWIC system (optimized compilation with the TowerEiffel compiler 1.4.3.0b) on a SPARC workstation is only 40960 bytes large. See Appendix ?? for instructions on getting the full source code of this example.

Example 3.44

```
indexing
    description: "exercise the KWIC problem with a simple example"

class DRIVER
creation
    make
feature
    library : LISTE[BOOK]
    non_kw : LISTE[STRING]
    make is
        local
            k : KWIC
        do
            read_library              -- read the list of books
            read_non_keywords         -- read the list of non_keywords
```

```
        !!k.make(library,non_kw) -- build the KWIC
        k.quick_sort             -- sort it
        k.print_index            -- print it
    end -- make
  read_library is                                                    20
    -- put a number of books in the 'library' list.
    local
        b : BOOK
    do
        !!b.make("The Hitch-Hiker's Guide to the Galaxy",            25
                 "G. Adams", 4242)
        !!library.make(b,Void)
        !!b.make("Software Engineering with Ada and Modula-2",
                 "R. Wierner and R. Sincovec", 6543)
        library:=library.append(b)                                   30
        !!b.make("Object-Oriented Software Engineering with Eiffel",
                 "J.-M. Jezequel", 6789)
        library:=library.append(b)
        !!b.make("The Evolution of the Universe: Part 1",
                 "C. Charlie", 9834)                                 35
        library:=library.append(b)
    end -- read_library
  read_non_keywords is
        -- put a number of non_keywords in the 'non_kw' list.
    do                                                               40
        !!non_kw.make("and",Void)
        non_kw := non_kw.append("of")
        non_kw := non_kw.append("or")
        non_kw := non_kw.append("part")
        non_kw := non_kw.append("s")                                 45
        non_kw := non_kw.append("the")
        non_kw := non_kw.append("to")
        non_kw := non_kw.append("using")
        non_kw := non_kw.append("with")
    end -- read_non_keywords                                         50
end -- DRIVER
```

Chapter 4

The Eiffel Environments

At this point you know enough about the Eiffel language to understand probably more than 95% of any Eiffel system. Eiffel programs do not exist in the void, so this chapter brings in environment matters: system configuration and monitoring, an overview of the Eiffel kernel library, interfacing with external software, and controlling garbage collection.

4.1 System Assembly and Configuration

4.1.1 Assembling Classes

An Eiffel software system is the *assembly* of a number of software components, usually a mixture of on-the-shelf and *ad hoc* classes, with occasionally a pinch of external software.

Assembling a system consists of telling the compiler where the relevant classes (and potential external software) are located, which one among them is the "root" class of the Eiffel program, and which creation feature of the root class is the program entry point. System assembly can be customized according to many options such as assertion monitoring level, debugging level, optimization level, garbage collection status, and tracing. These options may be specified at a system-wide level or, except for the garbage collection status, on a per cluster or even on a per class basis. This kind of specification of what to do with many software components is called an assembly of classes for Eiffel, (ACE). In the next sections of this chapter, we describe in further detail the various concepts involved in ACE. However, we need a notation to describe ACE. There are several variants of such notation, none of which is standardized yet by NICE. Two of them are:

They contain roughly the same information, so some translators are even available.

LACE is the *language for assembling classes in Eiffel*. It has an Eiffel-like syntax, and deals with all the items mentioned previously. It is used in the environments sold by ISE or Tower Technology.

PDL/RCL provides the same kind of functionality, but it is divided into two parts: the *program description language* that deals with the compilation management itself, and the *run-time control language* that deals with execution options (assertion monitoring level, debugging level, etc.). PDL/RCL is used in the SiG Computer GmbH environments.

You don't need to be very proficient in their respective syntaxes, because usually the environment provides you with a template with all the default fields that you simply customize for your application.

We overview the most frequently available mechanisms to configure Eiffel systems. Compiler vendors sometimes add several useful features (e.g., the ability to generate various executables at once, or the possibility of sharing precompiled code among many systems) that are not described here.

4.1.2 Generating an Application

The first part of an ACE deals with the notion of an application, that is, an Eiffel system. It allows you to specify the name of the generated executable program (or possibly the library) and where its entry point is located.

For example, if we consider the ACE corresponding to Example 2.1 (the "*Hello world*" program), we would specify that the executable file should be called *hello*, that the root class is the class HELLO (found by default in a file called *hello.e*, and that the entry point is the feature *make*. This translates in LACE to:

```
system hello
root hello : make
```

which corresponds to the following PDL:

```
program hello
root hello : make
```

4.1.3 Specifying Clusters

An ACE description should allow you to specify which clusters contain the sets of classes used in your application. A cluster is not an Eiffel language-level notion. Any mechanism that allows you to group several

related classes might be used. In most UNIX, Windows or DOS systems, a cluster is usually a directory containing a set of files, themselves containing class descriptions.

For both LACE and PDL, cluster descriptions come after the keyword **cluster**. For example, here is a LACE fragment specifying that two clusters are used, a *local* one, which is the current working directory, and the *kernel* one, which is probably mandatory for most compilers, because it contains the Eiffel Standard Library classes.

$EIFFEL refers to the value of an environment variable.

```
cluster
   local: ".";
   kernel: "$EIFFEL/clusters/kernel";
```

The difference in the PDL version is that clusters are not given symbolic names:

```
cluster
   "."
   end
   "$EIFFEL/library/basic"
   end
```

4.1.4 Excluding and Including Files

By default, all class texts contained in files suffixed with ".e" are considered by the compiler for each specified cluster. This default rule may be modified in two ways:

- If some ".e" suffixed files do not contain relevant Eiffel classes, they can be *excluded* from the cluster, which translates to LACE with:

  ```
  exclude "file1"; "file2"; .. "filen";
  ```

  and to PDL with:

  ```
  exclude "file1", "file2", .. "filen".
  ```

- Conversely, if some relevant Eiffel classes are not stored in conventional files (e.g., because of some OS filename limitations), they still can be included in the cluster. With LACE, you may use the **include** keyword:

  ```
  include "file1"; "file2"; .. "filen";
  ```

whereas with PDL you may use a *find* clause to let the compiler look for specific classes in some files:

```
find classname1 in "file1", classname2 in "file2".
```

4.1.5 Dealing with Class Name Clashes

In large projects in which you use several third-party libraries, it is highly possible that a class name clash will eventually happen. In some cases (e.g., if you don't have the sources of the libraries), it will be beyond your reach to change the source code to avoid such clashes. Both LACE and PDL provide an external way to *rename* conflicting classes, in much the same way as conflicting features are renamed with multiple inheritance.

In LACE, you may use an *adapt* clause within a cluster specification:

```
local : "."
  adapt
    cluster1:
      rename C as C1,
             D as D1;
    cluster2:
      rename C as C2;
```

Thus for all the classes of the *local* cluster, the classes C and D belonging to the cluster *cluster1* are known under the names C1 and D1, whereas the class C belonging to the cluster *cluster2* is known as C2.

In PDL, the renaming occurs within the original clusters of the conflicting classes, and a *use* clause specifies for which classes the new name is to be used (this is either an enumeration, or the keyword **all** to specify it for all classes).

```
"$DVP/cluster1":
  rename C as C1,
         D as D1
  use C1, D1 for [list of class]

"$DVP/cluster2":
  rename C as C2
  use C2 for [list of class]
```

4.2 Assertion Monitoring

4.2.1 Rationale

Remember that the Eiffel modularity is based on the "programming by contract" principle. When you violate the preconditions of a class feature, you don't respect your part of the contract: there is an error in *your* code. It could be called a *domain error*. It is up to you whether you want a clean error message at run time (telling you where and why you have an error), or maximum efficiency, risking a dirty crash if your program contains a domain error.

There are thus two mutually exclusive approaches to dealing with assertion checking:

- In the first case, you don't use assertion checking to bring you security at run time (i.e., software fault tolerance), but just to help you test your classes.

  Domain errors are much like type errors, because what you can do (and what you can't) with a class (the implementation of an ADT) is defined by the feature signatures *and* the class assertions (preconditions and invariants). However, the problem of checking domain errors is not (generally) achievable at compile time, and thus run time checking is the poor man's solution.

  If your compiler is clever enough to detect some domain errors at compile time, then it saves you *testing* time (the same as for type errors). When you are confident that your program is correct and if you badly need a full efficiency, just disable assertion checking.

- The second approach is to use assertions as a variant of defensive programming, to bring some kind of software fault tolerance. Software then may be built with the hypothesis that at least precondition checking is always enabled, and may be prepared to handle exceptions raised by precondition violations (see Section ??).

  In this case you should be careful about the performance penalties of assertion checking, at least until Eiffel compilers are clever enough to check assertions at compile time whenever possible.

4.2.2 Enabling Assertion Checking with LACE

Assertion checking can be enabled with `assertion(level)`, where *level* is one of the following keywords representing the level of monitoring:

no means that no assertion is to be checked,

require means that only preconditions are to be checked,

ensure means that *both* preconditions and postconditions are to be checked,

invariant means that class invariants also should be checked (see Section 2.5.3 on page 57),

loop means that loop variants and invariants also should be checked (see Section 2.4.7 on page 48),

check means that check instructions also should be executed (see Section 2.4.8 on page 53),

all means everything should be checked (actually, it is equivalent to the level **check**).

Assertion monitoring is a subset of the compilation options that can be specified with LACE. Like other options, they may appear at any of the three following levels:

- At system-wide level with a top-level *default* clause,
- At cluster level with a cluster-level *default* clause,
- On a per class basis, with an *option* paragraph in the cluster specification.

Remember that most Eiffel environments offer a user-friendly way to generate these ACEs.

Example:

```
system hello
root hello : make

default
  assertion(ensure); debug(no);
cluster

   local: "."
     default
       assertion(all); debug(all);
     end;

   kernel: "$EIFFEL/clusters/kernel"
     option
       assertion(invariant): ARRAY;
     end;
```

The same mechanism is used to enable *tracing* (a message is printed each time a routine is entered or exited) and *debug* code (that is, code enclosed in a *debug* instruction; see Section 2.4.9 on page 54). The LACE keywords **trace** and **debug** are used in much the same way as the **assertion** keyword. The argument *no* means that no debug code is to be executed. A list of keys may be used to enable the execution of the corresponding debugging code:

```
debug("TRACE","RECURSIVITY").
```

The argument *all* allows all debugging code to be executed, whatever its key.

4.2.3 Enabling Assertion Checking with Run-time Control Language

Enabling assertion checking and debugging code are not compile-time options specified with PDL, but run-time options specified with *run-time control language* (RCL). The advantage of this approach is that you don't have to recompile your Eiffel system if you just want to modify such options. The final code you get is not fully optimized, however, unless you specify "optimization: on" in your PDL file. In that case, you can no longer have assertion checking.

An RCL file is simply a list of *tags* (corresponding to the various assertion checking levels above) followed with a ":" and a list of class names or the keywords **none** and **all**. For example:

```
precondition: all
postcondition: ARRAY, STRING
loop_variant: none
loop_invariant: none
invariant: ARRAY
debug: none
debug_key: "TRACE","RECURSIVITY"
trace: KWIC_ENTRY
```

4.3 Overview On the Eiffel Standard Library

4.3.1 Purposes of the Eiffel Standard Library

Eiffel has a precise language definition that guarantees a first level of interoperability among various compilers. In practice, however, the compilers

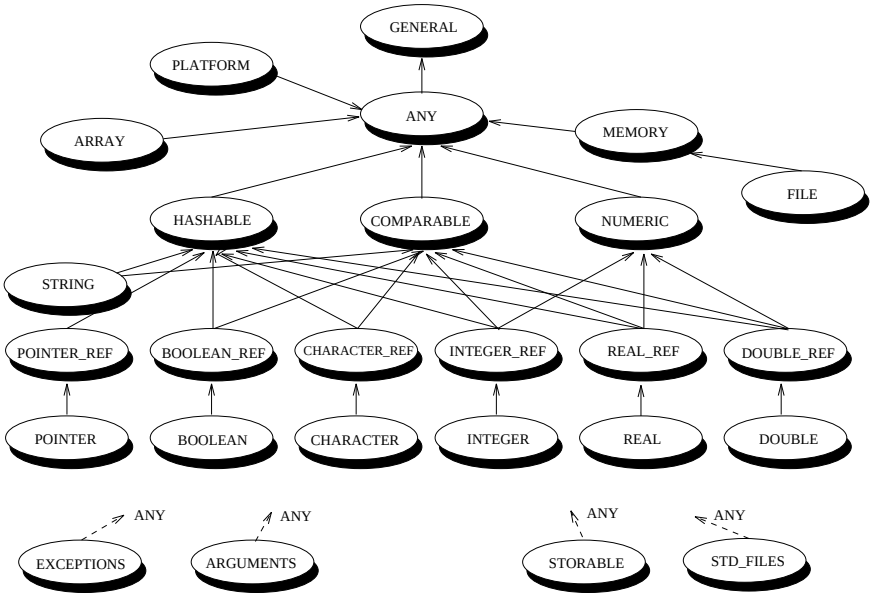


Figure 4.1: The inheritance relationships among standard classes

are dependent on a small set of classes called the Eiffel standard library in areas such as array and string manipulation, object copying and cloning, input-output, object storage, basic types, etc. For libraries and applications to be portable across compilers, these classes should present the same interface in each implementation. This is the purpose of the Eiffel Library Standard, which is standardized by NICE. This standard is revised on a yearly base, in the spirit of preserving the technology investment of Eiffel users, while allowing for improvements. Each successive version of it being known as a Vintage.

In this section we give an overview of the Vintage 95 of the Eiffel Library Standard. Full details are available from NICE, or from your Eiffel compiler vendor (see Appendix **??**).

4.3.2 Required Standard Classes

Twenty six classes belong to the Eiffel standard library. These classes are partially ordered with the inheritance relationships according to Figure 4.1. Here is a brief description of them:

1. GENERAL, which encapsulates all platform-independent universal properties of objects (see Section 3.3.5 on page 94).

2. ANY, which is an ancestor to all developer-written classes. ANY inherits from GENERAL and may be customized for individual projects or teams. (see Section 3.3.5 on page 94).

3. COMPARABLE, encapsulating the notion of objects that may be compared according to a total order relation. It has a deferred function ($infix$ " $<$") and a number of effective features defined after this function. (see Section 3.6.2 on page 106 and Example 3.35).

4. HASHABLE, representing values that may be hashed into an integer index, for use as keys in hash tables. It has a deferred function *hash_code*. Among others, BOOLEAN, CHARACTER, INTEGER, POINTER, and STRING are descendants of HASHABLE, thus hash tables have keys derived from these types.

5. NUMERIC is a deferred class encapsulating the notion of objects to which numerical operations (available in a commutative ring) are applicable ($+, -, *, /, etc.$). (see Section 3.7.2 on page 112).

6. BOOLEAN, representing truth values, with the usual Boolean operations, as described in Section 2.5.7 on page 62, with some of its features presented in Examples 2.18 and 2.19.

7. CHARACTER, representing ASCII characters, with comparison operations. It was outlined in Section 2.3.4 on page 40. This class inherits from COMPARABLE and HASHABLE.

8. INTEGER, for integer values (outlined in Section 2.5.7 on page 62). This class inherits from COMPARABLE, NUMERIC and HASHABLE.

9. REAL, floating-point values (single precision) as outlined in Section 2.5.7. This class inherits from COMPARABLE, NUMERIC, and HASHABLE.

10. DOUBLE, floating-point values (double precision), also outlined in Section 2.5.7. This class inherits from COMPARABLE, NUMERIC, and HASHABLE.

11. POINTER, representing references to objects meant to be exchanged with non-Eiffel software (see Section 4.4.2 on page 136). Pointers are HASHABLE.

12. ARRAY, implementing sequences of values, all of the same type or of a conforming one, accessible through integer indices in a contiguous interval. (see Section 3.2.3 on page 88, and Example 3.22 on page 90).

13. STRING, implementing sequences of characters, accessible through integer indices in a contiguous range. It was described in Section 3.1.5 on page 80, with some of its features presented in Example 3.11 on page 82 to 3.16. This class inherits from COMPARABLE and HASHABLE.

14. STD_FILES, encapsulating commonly used input and output mechanisms. Most notably, the feature *io* of the class GENERAL is an instance of STD_FILES that may be used by any class to deal with basic input/output such as reading or writing integers, reals, or strings from or to standard input or standard output or standard error.

15. FILE, viewed as persistent sequences of characters (see Section 4.3.3).

16. STORABLE, encapsulating the notion of objects that may be stored and retrieved along with all their dependents. This class may be used as an ancestor by classes needing persistency properties. It thus provides a primitive connection toward object-oriented databases.

17. MEMORY, encapsulating facilities for tuning up the garbage collection mechanism. This class may be used either as an ancestor or as a supplier by classes that require its facilities (this will be described in Section 4.5.3 in relation to garbage collection). Its interface appears as Example 4.6 on page 141.

18. EXCEPTIONS, encapsulating facilities for adapting the exception handling mechanism. It is described in Section ?? and its interface appears as Example ?? on page ??.

19. ARGUMENTS, encapsulating facilities for accessing command-line arguments (see Section 4.3.3).

20. PLATFORM, encapsulating platform-dependent properties such as the number of bits in an INTEGER or a REAL.

In addition, the six classes BOOLEAN_REF, CHARACTER_REF, INTEGER_REF, REAL_REF, DOUBLE_REF, and POINTER_REF are available as the reference classes corresponding to the six basic types.

4.3.3 Using I/O Classes: An Example

The classes ARGUMENTS, FILE, and STD_FILE can be used to build a new driver for the KWIC system described in Section 3.8. This new driver inherits from the class DRIVER of Example 3.44 and redefines the procedures *read_library* and *read_nonkeywords* to deal respectively with a file of books given as the first argument on the command line, and a file of nonkeywords given as the second argument on the command line or the standard input if the second argument is "-". These procedures deal with abnormal cases in reading their input files through exceptions.

Example 4.1

```
indexing
    description: "exercise the KWIC problem on a small database"
class FILEDRIVER
inherit
    DRIVER                                                          5
        redefine read_library, read_non_keywords
        end
    ARGUMENTS
creation
    make                                                            10
feature
    read_library is
        -- read book descriptions from the file whose name is given
        -- as argument(1). The format of a book description is:
        -- title %T authors %T inventory_number                     15
        -- (%T is the <TAB> character in Eiffel)
        local
            f : FILE
            title, authors : STRING
            inventory : INTEGER                                     20
            b : BOOK
            tab_position1, tab_position2 : INTEGER
            new_head : like library
        do
            from !!f.make_open_read(argument(1))                    25
            until f.end_of_file
            loop
                f.readline      -- read a line from f and leave it in 'laststring'
                tab_position1 := f.laststring.index_of('%T',1)
                                        -- get position of the first %T   30
                title := f.laststring.substring(1,tab_position1-1)
```

```
                                    -- extract title
            tab_position2 := f.laststring.index_of('%T',tab_position1+1)
                                    -- get position of the second %T
            authors := f.laststring.substring(tab_position1+1,
                tab_position2-1) -- extract authors
            inventory := f.laststring.substring(tab_position2+1,
                                f.laststring.count).to_integer
            !!b.make(title,authors,inventory) -- make a new book
            !!new_head.make(b,library) -- add it to the library
            library := new_head
        end -- loop
        rescue
            io.error.putstring(argument(1))
            io.error.putstring(": read error%N")
            -- print error message on STDERR & propagate the exception
    end -- read_library
read_non_keywords is
    -- read a list of non keywords from the file whose
    -- name is given as argument(2). If the name is '-',
    -- then read from STDIN
    local
        f : FILE                -- source of data
        word : STRING           -- to store a non_keywords
        new_head : like non_kw
    do
        if argument(2).is_equal("-") then -- read from STDIN
            f := io.input
        else -- read from file argument(2)
            !!f.make_open_read(argument(2))
        end -- if
        from
        until f.end_of_file
        loop
            f.readword -- read a word and leave it in f.laststring
            word := clone(f.laststring) -- clone the last word read
            !!new_head.make(word,non_kw) -- add it to the list
            non_kw := new_head
        end -- loop
        rescue
            io.error.putstring(argument(2))
            io.error.putstring(": read error%N")
            -- print error message on STDERR & propagate the exception
    end -- read_non_keywords
end -- FILEDRIVER
```

4.4 Interfacing with Other Languages

4.4.1 Declaring external Routines

Eiffel promotes the construction of software systems through the assembly of reusable software components, so it has to be possible to interface new Eiffel software with existing pieces written in other languages. Also, if Eiffel is to be used within the context of embedded systems, the problem of interfacing with the low-level functions (usually written in assembly language or C) must be dealt with.

As seen in Section 2.5, the routine body of an effective routine may be *external*, that is, not implemented within an Eiffel class. To specify that a routine has an external implementation, one must use the keyword **external** instead of **do** or **once**. This keyword must be followed by a manifest string (that is, a string between double quotes) indicating the language in which the routine is written (e.g., "C"). The declaration of Example 4.2 can be used to make the UNIX system call *sync* available to an Eiffel system.

Other effective routines are internal, *that is, they begin with* **do** *or* **once**; *see page 54.*

Example 4.2

```
sync is
    -- forces pending output to be written out immediately to the disk
    external
        "C"
    end                                                              5
```

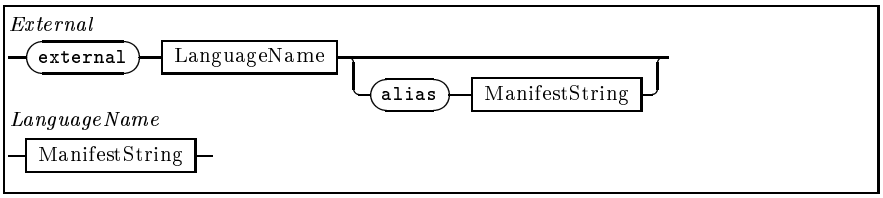
The actual implementation language is not that important, as long as it follows calling conventions that are compatible with the language specified in the manifest constant.

It is possible to refer to an external routine through a name other than its original name. This renaming is useful for giving a routine a new Eiffel name to follow naming conventions (see Section **??**), or simply because the foreign name contains uppercase letters, or otherwise does not match Eiffel syntax (e.g., leading underscore). In such a case you may use an alias, that is a syntactic clause introduced with the keyword **alias** and followed with a manifest string containing the original (foreign) name of the routine (e.g., *getpid* in Example 4.3).

The syntax of an external routine body is presented in Syntax Diagram 35.

Example 4.3

```
pid : INTEGER is
    -- the id of the process executing this program
    external
        "C"
    alias                                                    5
        "getpid"
    end
```



Syntax Diagram 35: External routine declaration syntax

4.4.2 Calling External Routines

Once they are given an Eiffel specification (signature, and possibly preconditions and postconditions), external routines may be called exactly like internal ones. The only caveat lies in the argument and result transmission: their type must be common to Eiffel and the external language. Clearly, this depends on both the Eiffel implementation and the foreign language one.

If you are not using this case (e.g., you use DOS or Windows 3.1), check with your compiler vendor.

In the simplest case where the foreign language is C on a 32-bit computer (which is the case of most UNIX systems, Windows'95 or Windows NT), a given implementation of Eiffel might provide the following mappings:

Eiffel type	C type
INTEGER	int
REAL	float
DOUBLE	double
CHARACTER	char
POINTER	(void*)

The standard library classes STRING and ARRAY also provide (not yet standardized) features to facilitate the transmission of string and array parameters between Eiffel and C.

Consider for example the function *system* found in most UNIX, Windows and DOS environments. It takes as its parameter a C string (a pointer to a memory zone ending with a NUL character), and gives it to the shell as input. In the TowerEiffel environment, a feature called *to_c* can be used

Example 4.4

```
system (c_string:POINTER) is
    -- gives the c_string to the shell as input,
    -- just as if it had been typed as a command from a terminal
    external
        "C"                                                    5
    end
```

to convert an Eiffel STRING to a C string. This feature is named *to_pointer* in the ISE environment. An Eiffel feature might then make use of features such as *pid* or *system* as shown in Example 4.5.

Example 4.5

```
foo is
    -- print the pid and list the
    -- directory
    local
        command : STRING                                       5
    do
        io.putstring("This process number is: ")
        io.putint(pid); io.new_line
        command := "ls"
        io.putstring("This directory contains:%N")            10
        system(command.to_pointer) -- to_c for TowerEiffel
    end -- foo
```

Routines that deal with reference type may be declared on the external side as expecting a pointer on anything (void* in C). As long as the foreign routine only stores or forwards the reference, no major problem will arise.

To do anything more, the routine must access the internal structure of the object, using for instance the C Eiffel call-in library (CECIL) library (provided with some Eiffel implementations) for the C language.

The CECIL library contains macros, functions, types, and error codes dealing with the view C programs have on Eiffel objects, and how to call (back) features on them. Their actual contents are not yet standardized

by NICE, so you should check your vendor-specific documentation for more information.

4.4.3 The Address Operator

Eiffel provides an address operator $ with which you can obtain the address of features (both attributes and routines) from the enclosing class. This operator returns an object of type POINTER that is useful only for transmitting attribute or routine addresses to external software.

This mechanism can be used to implement callback to Eiffel routines from a user interface. The problem with this approach is that because the address of a feature is static by nature, you lose all the advantages of dynamic binding and jeopardize your program integrity.

4.4.4 Linking with External Software

Unless all of the external routines belong to the standard C library (*libc*), you must specify in your ACE file where to find their code.

With LACE, you may specify under an *external* clause the names of the object files (".o") and library files (".a") containing the external routines you need to link with:

```
external
  object: "filename.o $MYSOFT/mylib.a -ltermcap".
```

With PDL, you use two separate clauses for object files and library files:

```
link "filename.o"
lib  "$MYSOFT/mylib.a -ltermcap".
```

Most vendor environments allow their ACE files to drive one or several *makefiles* that could be needed to deal with the external file dependency management.

4.5 Garbage Collection

4.5.1 Definition

With many languages, programmers must explicitly reclaim heap memory at some point in the program by using a *free* or a *dispose* statement. Eiffel frees the programmer from this burden, by means of a garbage collector. The garbage collector's function [22] is to:

In practice, these two phases may be interleaved, because the reclamation technique is strongly dependent on the garbage detection one.

1. Find data objects that are no longer in use or distinguish the live objects from the garbage in some way (garbage detection),

2. Make their space available for reuse by the running program or reclaim the garbage objects' storage (garbage reclamation),

3. Optionally move objects in memory to enable memory compaction (i.e., defragmentation), thus improving the locality of reference.

An object is considered garbage and hence subject to reclamation if it is not reachable by the running program via any path of reference traversals. Live (that is reachable) objects are preserved by the collector, ensuring that the program can never traverse a dangling reference into a deallocated object.

The set of live objects is thus the set of objects on any directed path of references from the root object, or the instance of the root class created when the program starts. Any object that is not reachable from the root object is garbage, (useless) because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore can't affect the future course of the computation, and their space may be safely reclaimed.

4.5.2 Interest for Software Correctness

A comprehensive overview of garbage collection is available in [42].

Garbage collection is necessary for fully modular programming to avoid introducing unnecessary intermodule dependencies. If objects must be deallocated explicitly, some module must be responsible for knowing when other objects are no longer interested in a particular object. Thus, many modules must cooperate closely (liveness is a global property). This cooperation leads to a tight binding between supposedly independent modules, and introduces nonlocal bookkeeping into routines that might otherwise be locally understandable and flexibly composable. This bookkeeping inhibits abstraction and reduces extensibility, because when new functionality is implemented, the bookkeeping code must be updated.

The unnecessary complications and subtle interactions created by explicit storage allocation and deallocation are especially troublesome because programming mistakes often break the basic abstractions of the programming language, making errors hard to diagnose. Failing to reclaim memory at the proper point may lead to memory leaks; in other words, unreclaimed memory gradually accumulates until the program terminates or the swap space is exhausted. Reclaiming memory too soon can lead to very strange behavior, because an object's space may be reused to store completely different data while an old reference still exists.

These programming errors are particularly dangerous because they often fail to show up repeatably, making debugging very difficult (although products are available that can help). In the worst case, they don't show up at all until the program is stressed in an unusual way. For instance, if the allocator happens not to reuse a particular object's space, a dangling pointer may not cause a problem during the testing phase of the software. Later, perhaps long after delivery, the application may crash because of a special memory access pattern, or because of any other apparently undetermined reason. Similarly, some memory leaks (called slow leaks) may not be noticeable while a program is being used in normal ways—perhaps for many years—because the program terminates before too much extra space is used. If the code is incorporated into a long-running server program, however, the server eventually will exhaust the available memory and crash.

These problems lead many applications programmers to implement some form of application-specific garbage collection (e.g., reference counting) within a large software system. This obvious need for a reusable, bullet-proof garbage collection subsystem explains why garbage collection has to be part of any Eiffel implementation, and not left on the programmer responsibility.

4.5.3 The Cost of Garbage Collection

It was once widely believed that garbage collection was quite expensive relative to explicit heap management, but several studies [1, 46, 39] have shown that garbage collection is sometimes cheaper than explicit deallocation, and is usually competitive with it.

A well-implemented garbage collector should not slow running programs down by more than 10% (with respect to explicit heap deallocation).

Some programmers regard such a cost as unacceptable, but many others believe it to be a small price for the benefits in convenience, development time, and reliability. In the long run, poor program structures induced by manual garbage collection may incur extra development and maintenance costs, and may cause programmer time to be spent on maintaining inelegant code rather than optimizing time-critical parts of applications. Even for C and C++, several add-on packages exist to retrofit them with garbage collection [4].

Recent advances in garbage collection technology make automatic storage reclamation affordable for use in high-performance systems. Generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically observed tendency of objects to die young. Incremental techniques may even make garbage collection relatively attractive

for real-time systems [11, 23, 38, 43].

Most Eiffel implementations come with such an incremental garbage collector, which can be activated and suspended at will.

4.5.4 Controlling the Garbage Collector

To control the garbage collector, you may inherit from (or use an instance of) the standard standard library class MEMORY (the interface of which is presented in Example 4.6), and call *collection_off* at the beginning of your application. Then, once in a while or each time you have some time to waste (e.g., during asynchronous I/O or in a reactive system main loop when no event is waiting to be processed), you may call the routine *collection_on* to ask for a partial garbage collection. The more frequently you call it, the less time it takes. An example of this process is presented in Chapter **??**, along with actual performance results in Section **??**.

Example 4.6

```
indexing
    description: "Facilities for tuning the memory management system"
class interface MEMORY
feature
    full_collect                                                    5
        -- Force an immediate garbage collection if garbage collection is
        -- enabled; otherwise do nothing.
    collection_off
        -- Disable the garbage collector.
    collection_on                                                  10
        -- Enable the garbage collector.
    collecting : BOOLEAN
        -- Is garbage collecting enabled?
feature
    -- Finalization.                                               15
    dispose
        -- Called just before the garbage collector reclaims the object.
        -- This is only intended to enable cleaning of external resources.
        -- The object should not do remote calls on other objects since
        -- those may also be dead and have already been reclaimed.  20
        -- The current object is freed after the 'dispose' routine returns.
end -- MEMORY
```

4.5.5 Finalization

Finalization is the ability to perform actions automatically when an object is reclaimed [15]. It is especially useful when an object manages an external resource such as a file or a network connection. For example, it may be important to close a file when the object dealing with it is reclaimed. Finalization thus can generalize the garbage collector so that other resources are managed in much the same way as heap memory and with similar program structure. This generalization makes it possible to write more general and reusable code, rather than having to treat certain kinds of objects very differently than normal objects.

An Eiffel class that requires finalization facilities can inherit from the standard library class MEMORY (see its interface in Example 4.6). It then may redefine the procedure *dispose* (which does nothing by default) to implement the finalization actions. However, because finalization occurs asynchronously (i.e., whenever the collector notices the object is unreachable and does something about it), the finalization code should be written with care. It should concentrate on cleaning external resources and should not do remote calls on other objects because those also may be dead and may have been reclaimed already.

Bibliography

[1] A. W. Appel, J. R. Ellis, and Kai Li. – Real–time concurrent collection on stock multiprocessors. – In *SIGPLAN'88 Conf. on Prog. Lang. Design and Implementation*, pages 11–20, June 1988.

[2] R. Bielak and J. McKim. – The many faces of a class: Views and contracts. – In *Proc. TOOLS 11*, pages 153–161, August 1993.

[3] B. W. Boehm. – The high cost of software. – In Ellis Horowitz, editor, *Practical Strategies for Developing Large Software Systems.* Addison-Wesley, 1975.

[4] Hans-Juergen Boehm and Mark Weiser. – Garbage collection in an uncooperative environment. – *Software — Practice and Experience*, 18(9):807–820, September 1988.

[5] Jean-Pierre Briot and Pierre Cointe. – Programming with explicit metaclasses in Smalltalk-80. – In *Proceedings OOPSLA'89, ACM SIGPLAN Notices*, pages 419–432, October 1989.

[6] P. P. S. Chen. – The entity-relationship model: Toward a unified view of data. – *ACM TODS*, 1(1):9–36, March 1976.

[7] O. J. Dahl and C. A. R. Hoare. – *Structured Programming.* – Academic Press, 1972.

[8] O .J. Dahl, B. Myhrhaug, and K. Nygaard. – Simula 67 common base language. – Technical report, Oslo:Norwegian Computing Centre, 1970.

[9] E.W. Dijkstra. – *A Discipline of Programming.* – Prentice-Hall, 1976.

[10] Geoff Dromey. – *The Development of Programs From Specifications.* – Addison-Wesley, 1987.

[11] J. R. Ellis, K. Lil, and A. W. Appel. – Real-time concurrent collection on stock multiprocessors. – Technical Report 25, Digital Systems Research Center, Palo Alto, CA, February 1988.

[12] B. H. Liskov et al. – *CLU Reference Manual.* – Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., memo 161 edition, July 1978.

[13] A. Goldberg and D. Robson. – *Smalltalk-80: The Language and its Implementation.* – Addison-Wesley, 1983.

[14] John Guttag. – Abstract data types and the development of data structures. – *CACM*, 20(6):396–404, June 1977.

[15] B. Hayes. – Finalization in the collector interface. – In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 277–298, Saint-Malo (France), September 1992. Springer-Verlag.

[16] C. A. R. Hoare. – Monitors: An operating systems structuring concept. – *CACM*, 17(10):549–557, 1974.

[17] Watts Humphrey. – *Managing the Software Process.* – Addison Wesley, 1989.

[18] J. Ichbiah. – *Reference Manual for the ADA Programming Language.* – ANSI/MIL-STD 1815a, January 1983.

[19] M.A. Jackson. – *System Development.* – Prentice-Hall International, Series in Computer Science, 1985.

[20] C. Jard and M. Raynal. – The rudiments of object distribution in distributed systems. – In 2^{th} *Int. Conf. on computers Inf. Sciences, Istambul*, 1987.

[21] Ian Joyner. – A critique of C++. – In *TOOLS Pacific, International Conference on Technology of Object-Oriented Languages and Systems, Australia.* An extended version is also available as ftp://ftp.desy.de/pub/c++/misc/c++.critique.ps, 1992.

[22] D.E. Knuth. – *The Art of Computer Programming.* – Addison–Wesley, 1968. – Vol. 1 : Fundamental Algorithms.

[23] E. K. Kolodner and W. E. Weihl. – Atomic incremental garbage collection. – In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 365–387, Saint-Malo (France), September 1992. Springer-Verlag.

[24] W. LaLonde and John Pugh. – Subclassing ≠ subtyping ≠ is-a. – *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[25] B. H. Liskov and S. N. Zilles. – Programming with abstract data types. – *SIGPLAN Notices*, 9(4):50–59, April 1974.

[26] Barbara Liskov and John Guttag. – *Abstraction and Specification in Program Development.* – MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1986.

[27] Barbara Liskov and Jeannette M. Wing. – A new definition of the subtype relation. – In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[28] Merriam-Webster. – *Webster's 7th Collegiate Dictionary, in Electronic Form (xwebster, by Niels Mayer).* – Webster, 1963.

[29] B. Meyer. – *Object-Oriented Software Construction.* – Prentice-Hall, 1988.

[30] B. Meyer. – *Eiffel: The Language.* – Prentice-Hall, 1992.

[31] M. Minsky. – A framework for representing knowledge. – Artificial Intelligence Memo 252, MIT Laboratory for Computer Science, 1974.

[32] D. E. Monarchi and G. I. Puhr. – A research typology for object-oriented analysis and design. – *Communications of the ACM*, 9(35):35–47, September 1992.

[33] Peter Naur et al. – Report on the algorithmic language ALGOL 60. – *Communications of the ACM*, 3(5):299–314, May 1960.

[34] D. L. Parnas. – *Structured Analysis and Design.* – Infotech International Limited, 1978.

[35] D.L. Parnas. – Software aspects of strategic defence systems. – *Comm. of the ACM*, 28(12), December 1985.

[36] D. T. Ross. – Structured analysis (SA): A language for communicating ideas. – *IEEE Trans. Software Eng.*, SE-6(1):16–33, January 1977.

[37] W. Royce. – Managing the development of large software systems. – In *Proceedings of IEEE WESCON*, August 1970.

[38] Ravi Sharma and Mary Lou Soffa. – Parallel generational garbage collection. – In *Proceedings OOPSLA'91*, pages 16–32, November 1991. – Published as ACM SIGPLAN Notices, volume 26, number 11.

[39] Robert A. Shaw. – *Empirical Analysis of a LISP System.* – PhD thesis, Stanford University, February 1988. – Available as Technical Report CSL-TR-88-351.

[40] Philippe Stephan. – Building financial software with object technology. – *Object Magazine*, 5(4), July 1995.

[41] R. Wiener and R. Sincovec. – *Software Engineering with Modula-2 and Ada.* – John Wiley & Sons, 1984.

[42] Paul R. Wilson. – Uniprocessor garbage collection techniques. – In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.

[43] Paul R. Wilson and Thomas G. Moher. – Design of the opportunistic garbage collector. – In *Proceedings OOPSLA'89*, pages 23–36, October 1989. – Published as ACM SIGPLAN Notices, volume 24, number 10.

[44] Niklaus Wirth. – The programming language Pascal. – *Acta Informat.*, 1:35–63, 1971.

[45] Niklaus Wirth. – *Programming in Modula-2.* – Springer-Verlag, Berlin, 1983.

[46] Benjamin Zorn. – Comparing mark-and-sweep and stop-and-copy garbage collection. – In *Proceedings of the 1990 ACM Conference on Lisp and functional rogramming*, June 1990.