TOUCH OF CLASS

Learning to program *well* with objects and contracts

Bertrand Meyer

Draft of ongoing work by Bertrand Meyer

TOUCH OF CLASS:

Learning to program *well* with objects and contracts

This draft is a private version used to prepare the publication of the book. Do not copy or distribute it without the author’s permission.

Short contents

The full table of contents appears on page **xlix**.

Community resources

Touch of Class rests (at the time of publication) on six years of teaching the "Introduction to Programming" course at ETH Zurich, taken by all entering computer science students. In connection with the course and the book we have developed a considerable amount of pedagogical material. Instructors are welcome to use this material for their own teaching. On the Web page for both this book and the course

http://touch.ethz.ch

you will find links to:

- The full set of our course slides (PowerPoint + PDF) in its latest version.
- Streamable and downloadable video recordings of our lectures.
- Supplementary material.
- Exercises.
- Slides for exercise sessions (tutorials).
- Mailing list and Wiki page for instructors using *Touch of Class* as their textbook.
- *Traffic* software for download (Windows, Linux, ...)
- Published articles and technical reports on our pedagogical work in connection with the course, and our other work on computer science education including the TrucStudio course development framework.
- Information about courses using the textbook in other universities.
- Errata.
- An instructor's corner (requiring registration), for instructors of courses having adopted the book, with suggestions for homeworks and exams and some solutions.

All this material is freely available for academic use in connection with the present textbook (see license terms on the site). For other uses please contact us.

Most of the material, in particular the course slides and video recordings, is in English. German versions are available for some of the exercise session slides. We expect to add material in other languages as it becomes available; if you translate slides or other elements into another language, we will be happy to include the translations.

More generally we welcome all community contributions.

Dedication

This book is dedicated to two pioneers of computer science, as thanks for their unending influence and in recognition of their many brilliant insights:

C.A.R. Hoare, on the occasion of his 75th birthday.

Niklaus Wirth, with special gratitude for his development of computing science (informatics) at ETH.

Prefaces

```
note
    description:"[
        This book has two prefaces, one for instructors and one for students, as stated
        here through a contrived but correct use of its own programming notation.
                ]"
class PREFACING inherit
    KIND_OF_READER
create
    choose
feature -- Initialization
    choose
            -- Get the preface that's right for you.
        do
            if is_student then
                student_preface.read
            elseif is_instructor then
                instructor_preface.read
            else
                pick_one_or_both
            end

                check
                    -- You learn about dynamic binding
                note
                    why: "You will express this more elegantly"
                end
        end
end
```

Student_preface*

*The preface for instructors is on page xxiii.

Programming is fun. Where else can you spend your days devising machines of your own imagination, build them without ever touching a hammer or staining your clothes, make them run as by magic, and get paid — not too bad, thanks for asking — at the end of the month?

Programming is tough. Where else do products from the most prestigious companies fail even in ordinary use? Where else does one find so many users complaining so loudly? Where else do engineers routinely work for hours or days trying to understand why something that should work doesn't?

Get ready for the mastery of programming and its professional form, software engineering; get ready for both the toughness and the fun.

Software everywhere

By going into computing science you have chosen one of the most exciting and fast-moving topics in science and technology. Fifty years ago it was not even recognized as a scientific subject; today hardly a university in the world is without a CS department. Thousands of books, journals, magazines and conferences cover the field. The global revenues of its industry — called information technology or IT — are in the trillions. No other field, in the history of technology, has undergone growth of either such magnitude or such speed.

And we have made a difference. Without software there would be no large-scale plane travel, and in fact no modern planes (or modern cars, or high-speed trains) since their design requires sophisticated "Computer-Aided Design" software. To pay its workers, any large corporation would employ hundreds of people just to write the paychecks. A phone would still be a device tied to the wall by a cable. After taking a picture, you still could not see the result until the roll of film came back from processing. There would be no video games, no camcorders, no iPods and no iPhones, no Skype, no GPS to guide you to your destination even when there is no one around to ask. To produce a report you would still hand-write a draft, give it to a typist, and go through rounds of correction requests. A sudden itch to know the name of the captain in *The Grand Illusion*, or the population of Cape Town, or the author of a familiar citation, would require (rather than typing a couple of search words and getting the answer in a blink) a trip to the library. The list goes on; at the heart of countless practices that now pervade our daily life lie programs — increasingly sophisticated programs.

All this does not happen by itself. While computers may have become a commodity, programs — without which computers would be useless — definitely are not. Programming, the task of constructing new programs or improving existing ones, is a challenging intellectual pursuit that requires programmers possessing creativity and experience. Through this book you will become familiar with the world of programs and programming, with a view to becoming a professional in the field.

CASUAL AND PROFESSIONAL SOFTWARE DEVELOPMENT

Although more and more people are acquiring basic computing proficiency, being able to program at a professional level is another matter, and is what a curriculum in computing science will bring you.

For comparison, consider mathematics. A few centuries ago, just being able to add and subtract 5-digit numbers required a university education, and in return provided qualifications for such good jobs as accountant. Nowadays these skills are taught in grade school; if you want to become an engineer or a physicist, or just a stock trader, you need to study more advanced mathematical topics, such as calculus, in a university. The boundary between basic training and university-level education has moved up.

Computing is following the same evolution, only much faster — the scale is in decades, not centuries. Not so long ago, being able somehow to program a computer was enough to land a job. Do not expect this today; an employer will not be much more impressed if your résumé states “I have written programs” than if you say you can add numbers.

What increasingly counts is the difference between having some basic programming experience and being a software engineer. The former skill will soon be available to anyone who has gone through a basic education; but the latter is a professional qualification, just like advanced mathematics. Studying this book is a step towards becoming a computing professional.

Factors that distinguish professional software development from casual programming include **size**, **duration** and **change**. In professional software development, you may become involved in programs that reach into the millions of lines of program text, must remain in operation for years or decades, and will undergo many changes and extensions in response to new circumstances. Many an issue that seems trivial or irrelevant when you are working on a medium-size program, meant only to solve a problem of immediate interest, becomes critical when you move to the scale of professional development.

With this book I'll try to prepare you for the real world of software, where systems are complex, solve serious problems (often affecting human life or property), stay around for a long time, and must lend themselves gracefully to requests for change.

Prior experience — or not

This book does not assume any prior programming knowledge.

If you *did* program before, that experience will help you master the concepts faster. You will recognize some of the ideas, but you should also expect to be surprised at times, since the professional study of any topic is different from its use by the general public. Once in a while, for example, you may find that I belabor a seemingly simple matter. If so, you will (I think) discover after a while that the topic is not as simple as it seems at first, just as addition is more challenging to the mathematician than to the accountant. While you must be prepared to question some of your previous practices if they do not match the professional software engineering principles developed here, you can and should take advantage of everything you know. Learning to program well takes a lot of effort: every bit — every angle from which you can approach the problem — helps. In particular, the discussion relies, as explained below, on a supporting software system, Traffic. If you are familiar with programming and some programming languages, you will be able to discover some of Traffic by yourself, perhaps ahead of the official assignments. Do not hesitate to do so: one learns programming in part by reading existing programs for inspiration and imitation. You may have to do some guessing for elements of Traffic that rely on techniques and language constructs you have not formally studied yet, but this is where your experience will help you move faster.

On the other hand, if you have *not* done any programming, you're OK too. You might progress more slowly at the beginning, but should just study all the material carefully and do all the exercises. In particular, even though this book includes little actual mathematics, you will feel more comfortable if you have a mathematical mindset and the practice of logical reasoning. This is just as beneficial as programming experience, and will compensate for any handicap you feel relative to those fellow students in the back row who look like they typed their first program before they lost their baby teeth.

Programming, like the rest of computing science, is at the confluence of engineering and science. Success requires both a hands-on attitude (the "hacker" side, in the positive sense of the word), useful in technology-oriented work, and an ability to perform abstract, logical reasoning, required in mathematics and other sciences. Experience with programming helps you with the first goal; a logical mind helps you with the second. Wherever your strength lies, take advantage of it, and use this book to make up for any initial deficiency on the other side.

MODERN SOFTWARE TECHNOLOGY

Becoming a software professional requires more than one course or one book: it takes a multi-year curriculum in which — in addition to mathematical foundations such as logic and statistics — you will learn about software engineering, theory of computation, data structures, algorithms, operating systems, artificial intelligence, databases, hardware, networking, project management, software metrics, numerical computation, graphics and many other topics. But to prepare for these other computer science courses it is essential to use the best of what is known in software technology.

In recent years two major ideas, holding the potential for producing software of much better quality than was available before, have made their way into the software field: **object-oriented software construction** and **formal methods**. Both of these ideas, but especially the first, can be used to make the introductory study of computing more exciting and more profitable. Along with other concepts from modern software technology, they play a major role in this book. Let's have a quick advance look at both.

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

Object-oriented ("O-O") software construction follows from the realization that proper systems engineering must rely on a large inventory of high-quality reusable components, as in the electronic or construction industries. The O-O approach defines what form these components should have: each of them must be based on a certain *type of objects*. The term "object", which gives its name to the method, does not just refer to objects of the application domain, such as circles or polygons in a graphics program, but also to objects that are purely internal to the software, such as a list. If you do not quite see what this all means, that's normal; I hope that if you read this preface again in a few months it will all be crystal-clear!

A previous book ("Object-Oriented Software Construction", 2nd edition, Prentice Hall,1997) covers object technology in depth and at a more advanced level.

Object technology (the shorter name for object-oriented software construction) is quickly changing the software industry. Becoming familiar with it from the very beginning of your computing studies is an excellent insurance policy against technical obsolescence.

FORMAL METHODS

Formal methods are the application of systematic reasoning techniques, based on mathematical logic, to the construction of reliable software. Reliability, or rather the lack of it, is a vexing problem in software; errors, or the fear of error, are the programmer's constant companion. Anyone who uses computers has some anecdote about bugs.

Formal methods can help improve this situation. Learning formal methods in their full extent requires more knowledge than is available at the beginning of a university education. But the approach used in this book shows a significant influence of formal methods, in particular through the idea of *Design by Contract*, which considers the construction of software systems as the implementation of a number of individual contractual relations between modules, each characterized by a precise specification of obligations and benefits. I hope that you will understand the importance of these ideas and remember them for the rest of your career. In industry, everyone knows the difference between a programmer who just "hacks code" and one who is able to produce correct, robust and durable software elements.

Learning by doing

This book is not a theoretical presentation; it assumes that as you go along you practice what you learn on a computing system. The associated Web site provides links to the necessary software, in versions for Windows, Linux and other platforms, which you can download. Your school may also have the equivalent facilities available on its computers. In fact, the text prompts you, in some cases, to do the practical work with the software *before* learning the theoretical concepts.

touch.ethz.ch.

The system that you will use for this course is an advanced object-oriented environment: EiffelStudio, an implementation of the Eiffel analysis, design and programming language. Eiffel is a simple, modern language, used worldwide in large, mission-critical industrial projects (banking and finance, health care, networking, aerospace etc.) as well as for teaching and research in universities. The EiffelStudio version that you will use is exactly the same as the professional version, with the same graphical development environment and fundamental reusable components such as the EiffelBase, EiffelVision and EiffelMedia libraries. Your school may also have an academic license providing for maintenance and support.

Appendices present an introduction to four other languages widely used in industry: Java, C#, C++ and C. Any good software engineer must be fluent in several programming languages, including at least some of these; learning Eiffel will be a plus on your résumé (a mark of professionalism) and will help you master other object-oriented languages.

From the consumer to the producer

Because from day one of the course you will have the whole power of EiffelStudio at your fingertips, you will be able to skip many of the “baby” exercises that have traditionally been used to learn programming. The approach of this book is based on the observation that to learn a technique or a trade it is best to start by looking at the example of excellent work produced by professionals, and taking advantage of it by (in order) using that work, understanding its internal construction, extending it, improving it — and starting to build your own. This is the time-honored method of apprenticeship, which places newcomers under the guidance of experts.

The expertise is represented here by software, more specifically *library classes*: software elements from the Traffic library, specially developed for this book. As you write your first software examples, you will use these classes to produce results which are already impressive even though you have not had much to write; you will simply be relying on the mechanisms defined by the classes, acting, through your own software, as a *consumer* of existing components. Then, as someone who knows how to drive but is studying to become an automobile engineer, you will be encouraged to lift the hood and see how these classes are made, so that you can later on write extensions to the classes, improve them perhaps, and write your own classes.

The Traffic library, as its name suggests, provides mechanisms for dealing with traffic in a city — cars, pedestrians, metros, trams, taxis … — with graphical visualization, simulations, route computation, route animation etc. It is a rich reservoir of applications and extensions: you can build on it to write video games, solve optimization problems and try out many new algorithms.

The built-in examples use Paris as the sample city, because it is a popular tourist destination; you can easily adapt them to another city without touching the Traffic software, since all the location information is provided separately in a file (using a standard format, XML). It suffices to provide such a file representing your chosen city. For example, the course as taught at ETH Zurich uses the Zurich tram system, replacing the Paris metro.

ABSTRACTION

Basing your work on existing components has another important consequence for your education as a professional software engineer. The program modules that you reuse are a substantial piece of software, embodying a lot of knowledge. It would be very difficult to use them for your own applications if you had to read the full program text of each one you need. Instead, you will rely on a description of their *abstract interfaces*, which are extracted from their text (by automatic software mechanisms, part of EiffelStudio) but retain only the essential information that you need as a consumer. An abstract interface is a description of the purpose of a software module that only states its functions, not *how* the module's code realizes these functions. In software terminology it is also called the *specification* of the module, excluding the module's *implementation*.

This technique will help you learn one of the professional software developer's key skills: *abstraction*, meaning here the ability to distinguish the purpose of any piece of software from the details, often numerous, of its implementation. Every software development professor and textbook preaches the virtues of abstraction, and for good reason; here you will get the occasional bit of preaching too, but mostly you will be encouraged to learn abstraction by example, experiencing its benefits through the reuse of existing components. When you get to build your own software, you should apply the same principles; that is the only way to tame the ogre of software complexity.

The benefits of abstraction are quite concrete; you will experience them right from the beginning. The first program you will write is only a few lines long, but already produces a significant result (an animated itinerary on a city map). It can do this only by using modules from Traffic; and it can use them only because they are available through an abstract specification. If you had to examine the text of these modules (their *source code*), then the text of the modules they rely on themselves, directly or indirectly, you would quickly drown in an ocean of details and could not produce anything.

→"A class text", 2.1, page 15.

Throughout your work with software, abstraction is the lifevest that will save you from drowning in the sea of complexity.

DESTINATION: QUALITY

This book teaches not only techniques but methodology. Throughout the presentation you will encounter design principles and rules on programming style. Sometimes you may think that I am being fussy and that you could write the program just as well without the rules. Well, often you could. But the methodological rules make the difference between an amateurish program, which sometimes works, sometimes not, and the kind of production-quality software that you will want to produce. You should apply these rules not just because this book and your teachers say so, but because the power and speed of computers magnify any deficiency, however small, and require that the programmer pay attention to both the big picture and every detail. They are also good job insurance for your future career: there are many programmers around, and what really differentiates them in the eyes of an employer is the long-term quality of the software they produce.

Do not fool yourself with the excuse that "this is only an exercise" or "this is only a small program":

- Exercises are precisely where you need to learn the best possible techniques; when Airbus hires you to write the control software for their next plane, it will be too late.
- Calling a program "small" is often more hope than guarantee. In industry, many big programs are small programs that grew, since a good program tends to give its users endless ideas for requesting new functionalities.

So you should apply the same methodological principles to all the programs you develop, whether small or large, educational or operational.

Such is the goal of this book: not just to take you through the basics of software engineering and to let you experience the fun and thrill of producing software that works, but also to develop — along with a sense of beauty for the principles, methods, algorithms, data structures and other techniques that define the discipline — a sense for what makes good software stand out, and a determination to produce programs of the highest possible quality.

BM
Zurich / Santa Barbara, April 2009

November 2012: for this second printing all known errors have been corrected.

Instructor_preface*

The preface for students is on page xiii.

Right from its subtitle, this book shows its colors: it is not just about learning to program but about "Learning to Program *Well*". I am trying to get the students started on the right track so that they can enjoy programming — without enjoyment one does not go very far — and have a successful career; not just a first job, but a lifelong ability to tackle new challenges.

To help them reach this goal, the book applies innovative ideas detailed in the rest of this preface:

- **Inverted curriculum**, also known as the "outside-in" approach, relying on a large library of reusable components.
- Pervasive use of **object-oriented** and model-driven techniques.
- **Eiffel** and **Design by Contract**.
- A moderate dose of **formal methods**.
- Inclusion, from the very beginning, of **software engineering** concerns.

These techniques have for several years been applied to the "Introduction to Programming" course at ETH Zurich, taken by all entering Computer Science students. *Touch of Class* builds on this course and draws from its lessons. This also means that teachers using it as a textbook can rely on the teaching material developed for the course: slides, lecture schedules, exercises, self-study tutorials, student projects, even video recordings of our lectures.

← *See* "Community resources", page vii.

THE CHALLENGES OF A FIRST COURSE

This section is based on reference [12].

Many computer science departments around the world are wondering today how best to teach introductory programming. This has always been a difficult task, but new challenges have added themselves to the traditional ones:

- Adapting to ever higher stakes.
- Identifying the key knowledge and skills to teach.
- Coping with fads and outside pressures.
- Addressing a broad diversity of initial student backgrounds and abilities.
- Meeting high expectations for examples and exercises.
- Introducing the real challenges of professional software development.
- Teaching methodology and formal techniques without scaring off students.

The stakes are getting ever higher. When educating future software professionals, we must teach durable skills. It is not enough to present immediately applicable technology, for which in our globalized industry a cheaper programmer will always be available somewhere.

We must **identify the key knowledge and skills** to teach. Programming is no longer a rare, specialized ability; a large proportion of the population gets exposed to computers, software and some rudimentary form of programming, for example through spreadsheet macros or Web site development with Python, Ruby on Rails or ASP.NET. Software engineers need more than the ability to program; they must master software development as a professional endeavor, and by this distinguish themselves from the masses of occasional or amateur programmers.

It is important to keep a cool head in the presence of **fads and outside pressures**. Fads are a given of our field, and they are not always bad — structured programming, object technology and design patterns were all fads once — but we must make sure an idea has proved its mettle before inflicting it on our students. Outside pressures can be more delicate to handle. Student families have more say nowadays; this too is not necessarily bad, but sometimes results in inappropriate demands that we teach the specific technologies required in the job advertisements of the moment. What this attitude misses is that four years later some of the fashionable acronyms will be different, and that competent industry recruiters look for problem-solving skills, not narrow knowledge. It is our duty to serve the very interests of the students and their families by teaching them the fundamental matters, which will give them not just a first job but a rewarding career.

> This obsession with learning the right résumé-filling buzzwords for fear of not landing a job is silly anyway. It is a worldwide phenomenon, likely to last for decades, that a decent software developer has no trouble finding a good job. For all the gloom that the media have spread after the “burst of the Internet bubble”, and the fears that “all the jobs have gone to Bangalore”, no end is in sight to the challenges and excitement of our field, including of course for our colleagues in Bangalore. But there is a qualification: people who get and *keep* good jobs are not the narrow-minded specialists having been taught whatever filled the headlines of the day; they are the competent developers possessing a wide and deep understanding of computing science, and mastery of many complementary technologies.

The **broad diversity of student backgrounds** complicates the task. Among the students in the lecture hall on the first day of the introductory course, you will find some who have barely touched a computer, some who have already built an e-commerce site, and the full range in-between. What can the teacher do?

- It is tempting to assume a fair amount of prior programming experience and teach to the most advanced students only; but this shuts out students who simply have not had the opportunity or inclination to work with computers yet. In my experience, they include some who can later turn out to be excellent computer scientists thanks to excellent abstraction skills, which they have so far applied to topics such as mathematics rather than computing. The nerdy image still widely associated with computers may have prevented them from realizing that it is not about late-night video game sessions fueled by home-delivery pizza (a picture which, in particular, turns off many girls with excellent computer science potential) but about cogent thinking applied to solving some of the most exciting intellectual challenges open to humankind.
- We must not either — at the other extreme — bring everyone down to the lowest level: we need a way to catch and retain the attention of the more experienced students, letting them use and expand the insights they have already gained.

Reliance on reusable components, discussed below, is a central part of this book's solution to the issue. By giving students access to high-quality libraries, we let the novices take advantage of their functionality through abstract interfaces without needing at first to understand what's inside. The more advanced and curious students can, ahead of the others, start to peek into the internals of the components and use them as guidance for their own programs.

For this to work we need **high-quality examples**. Students today, having lived most of their lives in a world awash in the visual and auditory marvels of software-powered multimedia, expect to see and build more than small academic programs of the "Compute the 7-th Fibonacci number" kind. We must meet these expectations of the "Nintendo Generation" [3], without of course letting technological dazzle push aside the teaching of timeless skills.

> A variant of this issue is what we may call the "Google-and-paste" phenomenon, the name I use for what colleagues (generally using Java or C++ as their teaching language) report as follows: you give an exercise that calls for, say, a 100-line program solution. Internet-savvy students quickly find on the Web some Java code that does the job, except that it does much more as part of, maybe, a 10,000-line program. Now it does not take long for beginners to hit upon a key piece of programming wisdom from the ages: that if you see a program that works you mess with it as little as you can. You hold your breath when coming anywhere close to it. Following this insight, the student will just switch off (rather than remove) the parts he or she does not need, through a minimal set of changes. So the teacher gets a 10,000-line solution to an elementary question. Of course one may impose, if not a full prohibition of Web use (which in a computer science curriculum would be bizarre), precise rules that would exclude such a "solution". But how exactly? "Google-and-paste" is, after all, a form of reuse, even if not exactly the kind advocated by software engineering textbooks.

> The approach of this book goes one step further. Not only do we encourage reuse, we actually provide a large amount of code (150,000 lines of Eiffel at the time of writing) for reuse, and also for imitation since it is available in source form and explicitly designed as a model of good design and implementation. Reuse is one of the "best practices" promoted by the course from the beginning; but it is a form of reuse in line with principles of software engineering, based on abstract interfaces and contracts.

These questions contribute to the next issue on our list: **introducing the real challenges of professional software development**. In a university-level computer science or software engineering program, we cannot just teach programming in the small. We have to prepare students for what matters to professionals: programming in the large. Not all techniques that work well for small programs will scale up. The very nature of an academic environment, especially at an introductory level, makes it hard to introduce students to the actual challenges of today's industrial software: software developed by many people, expanding to many lines of code, adapted to many categories of uses and users, maintained over many years, and undergoing many changes.

This concern for scalability gives particular urgency to the last issue: **introducing methodology and formal reasoning without disconnecting from the students**. Methodological advice — injunctions to use information hiding, contracts and software engineering principles in general — can sound preachy and futile to beginners. Introducing some formal (mathematically-based) techniques, such as axiomatic semantics, can widen this potential gap between teacher and student. Paradoxically, the students who have already programmed a bit and stand to benefit most from such admonitions and techniques may be most tempted to discard them since they know from experience that it is possible — at least for small programs — to reach an acceptable result without strict rules. The best way to instill a methodological principle is pragmatic: by showing that it empowers you to do something that would otherwise be unthinkable, such as building impressive programs with graphics and animation. Our reliance on powerful libraries of reusable components is an example: right from the beginning of the course, students can produce significant applications, visual and all, thanks to these components; but they would never proceed beyond a few classes if as a prerequisite they had to read the code. The only reuse that works here is through abstract interfaces.

Rather than pontificating on abstraction, information hiding and contracts, it is better to let the students use these techniques and discover that they work. If an idea has saved you from drowning, you will not discard it as sterile theoretical advice.

OUTSIDE-IN: THE INVERTED CURRICULUM

The order of topics in programming courses has traditionally been bottom-up: start with the building blocks of programs such as variables and assignment; continue with control and data structures; move on if time permits — which it often does not in an introductory course — to principles of modular design and techniques for structuring large programs.

This approach gives the students a good practical understanding of the fabric of programs. But it fails to teach the system construction concepts that software engineers must master to be successful in professional development. Being able to produce programs is no longer sufficient; many non-professional software developers can do this honorably. What distinguishes the genuine professional is the mastery of system skills for the development and maintenance of possibly large and complex programs, open for adaptation to new needs and for reuse of some of their components. Starting from the nuts and bolts, as in the traditional "CS1" curriculum, may not be the best way to teach these skills.

Rather than bottom-up — or top-down — the order of this book is **outside-in**. It relies on the assumption that the most effective way to learn programming is to use good existing software, where "good" covers both the quality of the code — since so much learning happens through imitation of proven models — and, almost more importantly, the quality of its program *interfaces* (APIs).

From the outset we provide the student with powerful software: a set of libraries, called Traffic, where the top layers have been produced specifically for this book, and the basic layers on which they rely (data structures, graphics, GUI, time and date, multimedia, animation…) are widely used in commercial applications. All this library code is available in source form, providing a repository of high-quality models to imitate; but in practice the only way to use them for one's own programs, especially at the beginning, is through API specifications, also known as *contract views*, which provide the essential information abstracted from the actual code. By relying on contract views, students are right from the start able to produce interesting applications, even if the part they write originally consists of just a few calls to library routines. As they progress, they learn to build more elaborate programs, and to understand the libraries from the inside: to "open up the black boxes". By the end of the course they should be able, if needed, to produce such libraries by themselves.

This Outside-In strategy results in an “Inverted Curriculum” where the student starts as a *consumer* of reusable components and learns to become a *producer*. It does not ignore the teaching of standard low-level concepts and skills, since in the end we want students who can take care of everything a program requires, from the big picture to the lowest details. What differs is the order of topics and particularly the emphasis on architectural skills, often neglected in the bottom-up curriculum.

The approach is intended to educate students so that they will master the key concepts of software engineering, in particular *abstraction*. In my career in industry I have repeatedly observed that the main quality that distinguishes good software developers is their ability to abstract: to separate the essential from the accessory, the durable from the temporary, the specification from the implementation. All good introductory textbooks duly advocate abstraction, but the result of such exhortations is limited if all the student knows of programming is the usual collection of small algorithmic examples. I can lecture on abstraction too, but in the end, as noted earlier, the most effective way to convey the concepts is by example; by showing to the student how he or she can produce impressive applications through the reuse of existing software. That software is large at least by academic standards; trying to reuse it by reading the source code would take months of study. Yet students can, in the first week of the course, produce impressive results by reusing it through the contract views.

Here abstraction is not just a nice idea that we ask our students to heed, another parental incitation to be good and do right. It is the only way to survive when faced with an ambitious goal which you can only reach by standing on someone else’s shoulders. Students who have gone early and often through this experience of building a powerful application through contract-based reuse of libraries do not need much more haranguing about abstraction and reuse; for them these concepts become a second nature.

Teaching is better than preaching, and if something is better than teaching it must be the demonstration — carried out by the students themselves — of the principles at work, and the resulting “Wow!”.

The supporting software

Central to the Outside-In approach of this book is the accompanying Traffic software, available for free download. The choice of application area for the library required some care:

From touch.ethz.ch.

- The topic should be immediately familiar to all students, so that we can spend our time studying software issues and solutions, not the problem domain. (It might be fun to take, say, astronomy, but we would end up discussing comets and galaxies rather than inheritance structures and class invariants.)

- The area should provide a large stock of interesting algorithm and data structure examples, applications of fundamental computer science concepts, and new exercises that each instructor can devise beyond those in the book. This should extend beyond the introductory course, to enable our colleagues teaching algorithms, distributed systems, artificial intelligence and other computer science topics to take advantage of the software if they wish.
- The chosen theme should call for graphics and multimedia development as well as advanced graphical user interfaces.
- Unlike many video games, it must not involve violence and aggression, which would be inappropriate in a university setting (and also would not help correct the gender imbalance which plagues our field).

The application area that we retained is *transportation in a city*: modeling, planning, simulation, display, statistics. The supporting Traffic software is not just an application, doing a particular job, but a *library*, providing reusable components from which students and instructors can build applications. Although still modest, it has the basic elements of a Geographical Information System and the supporting graphical display mechanisms.

For its examples the book uses Paris, with its streets and transportation system; since the city's description comes from XML files, it is possible to retarget the example to any other city. (In the second week of the first session of the course at ETH a few students spontaneously provided a file representing the Zurich transportation network, which we have been using ever since.)

The very first application that the student produces takes up twelve lines. Its execution displays a map, highlights the Paris Metro network on the map, retrieves a predefined route, and shows a visitor traveling that route through video-game-style graphical animation. The code is:

```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Show city info and route.
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

The algorithm includes only four instructions, and yet its effect is impressive thanks to the underlying Traffic mechanisms.

In spite of the reliance on an extensive body of existing software, I stay away from giving any impression of "magic". It is indeed possible to explain everything, at an appropriate level of abstraction. We should never say "*just do as you are told*, *you'll understand when you grow up*". This attitude is no better at educating students than it is at raising one's own children. In the first example as shown above, even the **inherit** clause can be explained in a simple fashion: I do not go into the theory of inheritance, of course, but simply tell the students that class *TOURISM* is a helper class introducing predefined objects such as *Paris*, *Louvre*, *Metro* and *Route1*, and that a new class can "inherit" from such an existing class to gain access to its features. They are also told that they do not need to look up the details of class *TOURISM*, but may do so if they feel the born engineer's urge to find out "how things work".

The rule, allowing our students to approach the topics progressively, is always to abstract and never to lie.

From programming to software engineering

Programming is at the heart of software engineering, but is not all of it. Software engineering concerns itself with the production of systems that may be large, are developed over a long time, undergo many changes, and meet strong constraints of quality, timeliness and cost. Although the corresponding techniques are usually not taught to beginners, it is important to provide at least a first introduction, which appears in the last chapter. The topics include requirements analysis (the programmers we educate should not just be techies focused on the machinery but should also be able to talk to customers and understand their needs), facets of software quality, an introduction to lifecycle models, the concept of agile development, quality assurance techniques and Capability Maturity Models.

An earlier chapter complements this overview by presenting software engineering tools, including compilers, interpreters and configuration management systems.

Terminology

Lucid thinking includes lucid use of words. I have devoted particular attention to consistent and precisely defined terminology. The most important definitions appear in call-out boxes, others in the main body of the text.

At the end of each chapter a "New vocabulary" section lists all the terms introduced, and the first exercise asks the student to provide precise definitions of each. This is an opportunity to test one's understanding of the ideas introduced in the chapter.

TECHNOLOGY CHOICES

The book relies on a combination of technologies: an object-oriented approach, Design by Contract, Eiffel as the design and programming language. It is important to justify these choices and explain why some others, such as Java as the main programming language, were not retained.

Object technology

Many introductory courses now use an object-oriented language, but not necessarily in an object-oriented way; few people have managed to blend genuine O-O thinking into the elementary part of the curriculum. Too often, for example, the first programs rely on static functions (in the C++ and Java sense of routines not needing a target object). There sometimes seems to be an implicit view that before being admitted to the inner chambers of modern technology students must suffer through the same set of steps that their teachers had to travel in their time. This approach retains the traditional bottom-up order, only reaching classes and objects as a reward to the students for having patiently climbed the *Gradus ad Parnassum* of classical programming constructs.

There is no good reason for being so fussy about O-O. After all, part of the pitch for the method is that it lets us build software systems as clear and natural *models* of the concepts and objects with which they deal. If it is so good, it should be good for everyone, beginners included. Or to borrow a slogan from the waiters' T-shirts at Anna's Bakery in Santa Barbara, whose coffee played its part in fueling the writing of this book: *Life is uncertain — Eat dessert first*!

Classes and objects appear at the very outset and serve as the basis for the entire book. I have found that beginners adopt object technology enthusiastically if the concepts are introduced, without any reservations or excuses, as the normal, modern way to program.

One of the principal consequences of the central role of object technology in this presentation is that the notion of *model* guides the student throughout. The emergence of "model-driven architecture" reflects the growing recognition of an idea central to object technology: that successful software development relies on the construction of models of physical and conceptual systems. Classes, objects, inheritance and the associated techniques provide an excellent basis to teach effective modeling techniques.

Object technology is not exclusive of the traditional approach. Rather, it subsumes it, much as relativity yields classical mechanics as a special case: an O-O program is made of classes, and its execution operates on objects, but the classes contain routines, and the objects contain fields on which programs may operate as they would with traditional variables. So both the *static* architecture of programs and the *dynamic* structure of computations cover the traditional concepts. We absolutely want the students to master the traditional techniques such as algorithmic reasoning, variables and assignment, control structures, pointer manipulation (whose coverage here includes algorithms to reverse a linked list, a tricky task seldom covered in introductory courses), procedures and recursion; they must also be able to build entire programs from scratch.

Eiffel and Design by Contract

We rely on Eiffel and the EiffelStudio environment which students can download for free from www.eiffel.com. Universities can also install this free version (and purchase support if desired). This choice directly supports the pedagogical concepts of this book:

- The Eiffel language is uncompromisingly object-oriented.
- Eiffel provides a strong basis to learn other programming languages such as Java, C#, C++ and Smalltalk (as demonstrated by appendices which introduce the essentials of the first three of these languages, in about 30 pages each, by building on the concepts developed in the rest of the book).

→Appendices A (Java), B (C#), C (C++).

- Eiffel is easy for beginners to learn. The concepts can be introduced progressively, without interference between basic constructs and those not yet studied.
- The EiffelStudio development environment uses a modern, intuitive GUI, with advanced facilities including sophisticated browsing, editing, a debugger with unique reverse execution capabilities, automatic documentation (HTML or otherwise), software metrics, and leading-edge automatic testing mechanisms. It produces architectural diagrams automatically from the code; the other way around, it lets a user draw diagrams from which the environment will produce the code, with round-trip capabilities.
- EiffelStudio is available on many platforms including Windows, Linux, Solaris and Microsoft .NET.
- EiffelStudio includes a set of carefully written libraries, which support the reuse concepts of this book, and serve as the basis of the Traffic library. The most relevant are: *EiffelBase*, which by implementing the fundamental structures of computer science supports the study of algorithms and data structures in part III: *EiffelTime* for date and time; *EiffelVision*, for portable graphics; and *EiffelMedia* for multimedia and animation.

- Unlike tools designed exclusively for education, Eiffel is used commercially for mission-critical applications handling tens of billions of dollars in investments, managing health care systems, performing civil and military simulations, and tackling other problems across a broad range of application areas. This is in my opinion essential to effective teaching of programming; a tool that is really good should be good for professionals as well as for novices.
- The Eiffel language is specified by a standard of the International Standards Organization. For the teacher relying on a programming language, an international standard, especially an ISO standard, is a guarantee of sustainability and precise definition.

For the text of the standard see tinyurl.com/y5abdx or the ECMA version (same contents, free access) at tinyurl.com/cq8gw.

- Eiffel is not just a programming language but a *method* whose primary aim — beyond expressing algorithms for the computer — is to support *thinking* about problems and their solutions. It enables us to teach a **seamless approach** that extends across the software lifecycle, from analysis and design to implementation and maintenance. This concept of seamless development, supported by the round-trip Diagram Tool of EiffelStudio, is in line with the modeling benefits of object technology.

To support these goals, Eiffel directly implements the concepts of **Design by Contract**, which were developed together with Eiffel and are closely tied to both the method and the language. By equipping classes with preconditions, postconditions and class invariants, we let students use a much more systematic approach than is currently the norm, and prepare them to become successful professional developers able to deliver bug-free systems.

One should also not underestimate the role of syntax, for beginners as well as for experienced programmers. Eiffel's syntax — illustrated by the earlier short example — facilitates learning, enhances program readability, and fights mistakes:

← Class PREVIEW, page xxix.

- The language avoids cryptic symbols.
- Every reserved word is a simple English word, unabbreviated (*INTEGER*, not int).
- The equal sign =, rather than doing violence to hundreds of years of mathematical tradition, means the same as in math.
- Semicolons are not needed. In most of today's languages, program texts are peppered with semicolons terminating declarations and instructions. Most of the time there is no reason for these pockmarks; even when not consciously noticed, they affect readability. Being required in some places and illegal in others, for reasons obscure to beginners, they can be a source of errors. In Eiffel the semicolon as separator is optional, regardless of program layout. This leads to a neat program appearance, as you may see by picking any example in the book.

Encouraging such cleanliness in program texts should be part of the teacher's pedagogical goals. Eiffel includes precise style rules, explained along the way to show students that good programming requires attention to both the high-level concepts of architecture and the low-level details of syntax and style: quality in the large and quality in the small.

More generally, a good language should let its users focus on the concepts rather than the notation. This is one of the goals of using Eiffel for teaching: that students should think about their problems, not about Eiffel

Why not Java?

Since courses in recent years have often used Java, or a Java variant such as C#, it is useful to explain why we do not follow this practice. Java is important for a computer scientist to know — indeed, as mentioned, the book provides an appendix describing Java, along with others on C#, C++ and C — but not suitable as a first teaching language. There is simply too much baggage to be learned before the student can start to think about the problems. This is apparent from the first program attempts; a Java "Hello World" reads

```
class First {
   public static void main(String args[])
   { System.out.println("Hello World!"); } }
```

This is full of irrelevant concepts, each an obstacle to learning. Why "public", "static", "void"? (Sure, I'll make my program *public* if you insist, but do you mean my efforts are *void* of any value?) These keywords have nothing to do with the purpose of the program, and the student won't begin to understand what they mean for a few months at least, yet he or she must include them, like magic incantations, for their programs to work. For the teacher this means repeatedly engaging in injunctions to use certain constructions without understanding what they mean. As noted earlier, this "*You'll understand when you grow up*" style is not good pedagogy. Eiffel protects us from it: we can explain every programming language construct that we use, right from the first example.

The object-oriented nature of Eiffel and the simplicity of the language play a role. It is ironic that every Java program, starting with the simplest example as shown above, uses a static function as its main program, departing from the object-oriented style of programming. There are of course people who do not like the idea of using O-O for the first course; but if you do choose objects, you should be consistent. At some point the students will realize that this fundamental scheme — the one you told them to use, from the first example to every subsequent one — is not object-oriented after all; how can you answer their inevitable question with a straight face?

Syntax, as noted, matters. In this first example the student must master strange symbol accumulations, like the final ""); } }", disconcerting to the eye and with no obvious role. In this accumulation the precise order of the symbols is essential, but is hard to explain and to remember. (Why a semicolon between a closing parenthesis and a brace? Is there a space after that semicolon, and if so how important is it?) Such aspects are troubling to beginners; inevitably, much time and effort are consumed learning them and recovering from trivial mistakes causing mysterious results, just when the student should be concentrating on the concepts of programming.

Another source of confusion is the use of "=" for assignment, inherited from Fortran through C and hard to justify in the twenty-first century. How many students starting with Java have wondered what value *a* must have for a = a + 1 to make sense, and, as noted by Wirth [15], why a = b does not mean the same as b = a ?

Inconsistencies are troubling: why, along with full words like "static", use abbreviations such as "args" and "println"? Students will retain from that first exposure to programming that it is not necessary to be consistent, and that saving keystrokes is more important than choosing clear names. (In the basic Eiffel library the operation to go to the next line is called *put_new_line*.) If indeed we later introduce methodological advice urging students to choose clear and consistent names, we can hardly expect them to take us seriously. "*Do as I say, not as I do*" is another dubious pedagogical technique.

To cite another example: when describing the need for a mechanism for treating operations as objects, like Eiffel's agents or the closures of other languages, I had to explain how one addresses the issue in a language such as Java that does not have these mechanisms. Since I used iterators as one of the motivating examples, I was at first happy to find that the original Sun page describing Java's "inner classes" also had code for an iterator design, which it would have been nice to use as a model. But then it includes declarations such as

→ Chapter 17.

See tinyurl.com/c4oprq (archive of java.sun.com/docs/ books/tutorial/java/ javaOO/ innerclasses.html, Oct. 2007; the page now uses a different example).

```
public StepThrough stepThrough() {
            return new StepThrough();
       }
```

I can perhaps try to justify this to seasoned programmers, but there is no way I can explain it to beginning students — and I admire anyone who can. Why does StepThrough appear three times? Does it denote the same thing each time? Is the change of letter case (StepThrough vs stepThrough) relevant? What does the whole thing mean anyway? Very quickly the introductory programming course turns into painful exegesis of the programming language, with little time left for real concepts. In Alan Perlis's words, "*A programming language is low-level when its programs require attention to the irrelevant*".

Epigram #8, available at www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

Also contributing to the difficulties of using Java in an introductory course are the liberties that the language takes with object-oriented principles. For example:

- If *x* denotes an object and *a* one of the attributes of the corresponding class, you may by default write the assignment *x*.*a* = *v* to assign a new value to the *a* field of the object. This violates information hiding and other design principles. To rule it out, you must shadow every attribute with a "getter" function. For the teacher, the choice is between forcing students early on to add such noise to their programs, or let them acquire bad design habits which are then hard to unlearn.
- Java strictly distinguishes fully abstract modules, called *interfaces*, from fully implemented ones — classes. One of the benefits of the class mechanism, available as early as Simula 67, is to offer a full range of possibilities between these extremes. This idea is at the core of teaching the object-oriented method, in particular teaching design: you can express a notion, when you first identify it, as a fully *deferred* (abstract) class; then you refine it progressively, through inheritance, into a fully effective class. Classes at intermediate levels in this process are partially deferred and partially effective. Java does not let you use this approach if you may need to combine two or more abstractions through inheritance: all but at most one of the combined modules must be interfaces.

There are many more examples of such influences of Java on the teaching process; a new Eiffel user expressed a typical reaction by writing on a mailing list that "*I have written a lot of C++ and Java*; *all my brain power went on learning loads of nerdy computer stuff. With Eiffel I do not notice the programming and spend my time thinking about the problem*."

A reason often invoked for using Java or C++ in introductory programming is the market demand for programmers in these languages. This is a valid concern, but it applies to the computer science curriculum as a whole, not to the first course. Programming at the level required of a CS graduate today is hard enough; we should use the best pedagogical tools. If market demand had been the determinant, we would never in the past have used Pascal (for many years the introductory language of choice), even less Scheme. Following the trends reflected in the latest ads for programmers we would in turn have imposed Fortran, Cobol, PL/I, Visual Basic, maybe C — and trained programmers who, a few years after graduation, would have found their skills obsolete when the great wheel of fashion turned. Our duty is to train problem-solvers who can quickly adapt to the evolutions of our discipline.

We should not let short-term market considerations damage pedagogical principles. In other words: if you think Java or C++ are ideal teaching tools, use them; you probably will not like this book very much anyway. But if you agree with its approach, do not let yourself be scared that some student or parent will complain that you use an "academic" approach. Explain to them that you are teaching programming in the best way you know, that someone who understands programming will retain that skill for life, and that any half-decent software engineer can pick up a new programming language at breakfast — in case he or she has not already picked it up from other courses of your curriculum. As to the "academic" qualification (assuming that in a university context, it is meant as derogatory!), point them to eiffel.com and its long list of mission-critical systems in Eiffel in the financial industry, aerospace, defense, networking, computer-aided design, health care and other areas, successfully deployed by major companies, often after attempts in other languages had failed.

Java, C#, C++ and C are, for the next few years, an important part of any software engineer's baggage; it is important, as reflected by this book's four language-specific appendices, to ensure that the students know them. This goal is, however, unrelated to the question of what techniques to use in the introductory course. Students will most likely be exposed to these languages at some point; it would be a rare curriculum these days where no course uses at least one of them. In any case, no introductory course that I know covers *all* of them, so students need to learn more regardless of the initial teaching language.

In our surveys [13], about 50% of students have used Java or C++ before *they reach the introductory course.*

Programming languages and the programming culture associated with each of them are interesting objects of study. Our group at ETH, which teaches introductory programming in Eiffel, has introduced courses for the third year and beyond, devoted to specific languages: "Java in Depth", "C# in Depth" etc.

Once you understand the concepts of programming, you are well prepared to master diverse languages. Eiffel is a benefit here too: as many people have noted, having learned Eiffel and its object model helps you become a better C++ or Java programmer.

> As a potential employer in both academia and industry I see dozens of CVs every month. They all boast of the same skills, including C++ and Java. Other than as checkboxes to be ticked, this will not impress anyone. What recruiters do watch for is any skill that sets out an applicant from the hordes of others with similar backgrounds. An example of such a distinctive advantage is that the applicant knows a fully object-oriented approach with support for software engineering, as evidenced by a curriculum using Eiffel and Design by Contract. It is possible to survive a C++-based curriculum without ever understanding O-O concepts in any depth; with Eiffel that is less likely. Competent employers know that what counts, beyond immediate skills, is depth of understanding of software issues and aptitude for long-term professional development. All the effort deployed through this book and its use of Eiffel is directed at these goals.

It may be appropriate here to cite Alan Perlis again: *A language that doesn't affect the way you think about programming is not worth knowing*.

Epigram #19.

How formal?

One of the benefits of the Design by Contract approach is to expose the students to a gentle dose of "formal" (mathematics-based) methods of software development.

The software world needs, among other advances, more use of formal methods. Any serious software curriculum should devote at least one course entirely to mathematics-based software development, based on a mathematical specification language. In addition — although not as a substitute for such a course — the ideas should influence the entire software curriculum, even though as discussed below it is not desirable today to subject beginners to a fully formal approach. The challenge is not only to include an introduction to formal reasoning along with practical skills, but to present the two aspects as complementary, closely related, and both indispensable. The techniques of Design by Contract, tightly woven into the fabric of object-oriented software architecture, permit this.

Teaching Design by Contract awakens students to the idea of mathematics-based software development. Almost from the first examples of interface specifications, routines possess preconditions and postconditions, and classes possess invariants. These concepts are introduced in the proper context, treated — as they should, although many programmers still fear them, and most programming languages offer no support for contracts — as the normal, obvious way to reason about programs. Without intimidating students with a heavy-duty formal approach, we open the way for the introduction of formal methods, which they will fully appreciate when they have acquired more experience with programming.

→ *In chapter* 4.

In no way does the use of a mathematical basis imply a stiff or intimidating manner. Some formality in the concepts goes well with a practical, hands-on approach. For example the text introduces loops as an *approximation* mechanism, to compute a solution on successively larger subsets of the data; in this view the notion of *loop invariant* comes naturally, at the very beginning of the discussion of loops, as a key property stating the approximation obtained at every stage.

This emphasis on practicality distinguishes Design by Contract from the fully formal approaches used in some introductory courses, whose teachers hold that students should first learn programming as a mathematical discipline. Sometimes they go so far as to keep them away from the computer for a semester or a full year. The risk of such dogmatism is that it may produce the reverse of its intended effect.

Students, in particular those who have programmed before, realize that they can produce a program — not a perfect program, but a program — without a heavy mathematical apparatus; if you tell them that it's not possible they will just disconnect: they may from then on reject any formal technique as irrelevant, including both simple ideas which can help them now and more advanced ones later. As Leslie Lamport — not someone to be suspected of underestimating the value of formal methods — points out [6]:

> [*In American universities*] *there is a complete separation between mathematics and engineering. I know of one highly regarded American university in which students in their first programming course must prove the correctness of every tiny program they write. In their second programming course, mathematics is completely forgotten and they just learn how to write C programs. There is no attempt to apply what they learned in the first course to the writing of real programs.*

Our experience confirms this. First-year students, who react well to Design by Contract, are not ready for a fully formal approach. To develop a real appreciation for its benefits you must have encountered the difficulties of industrial software development. On the other hand, it also does not work to let students develop a totally informal approach first and, years later, suddenly reveal that there is more to programming than hacking. The appropriate technique, I believe, is incremental: introduce Design by Contract techniques right from the start, with the associated idea that programming is based on a mathematical style of reasoning, but without overwhelming students with concepts beyond their reach; let them master the practice of software development on the basis of this moderately formal approach; later in the curriculum, bring in courses on such topics as formal development and programming language semantics. This cycle can be repeated, as theory and practice reinforce each other.

Such an approach helps turn out students for whom correctness concerns are not an academic chimera but a natural, ever-present component of the software construction process.

In the same spirit, the discussion of high-level functional objects (agents, chapter 17, and their application to event-driven programming in chapter 18) provides the opportunity of a simple introduction to **lambda calculus**, including currying — mathematical topics that are seldom broached in introductory courses but have applications throughout the study of programming.

OTHER APPROACHES

Looking around at university curricula, talking to teachers and examining textbooks leads to the observation that four main approaches exist today for introductory programming:

1 Language-focused.

2 Functional (in the sense of functional programming).

3 Formal.

4 Structured, Pascal or Ada-style.

It is important to understand the benefits of these various styles — indeed we retain something from each of them — and their limitations.

The first approach is probably the most common nowadays. It focuses on a particular programming language, often Java or C++. This has the advantage of practicality, and of easily produced exercises (subject to the Google-and-Paste risk), but gives too much weight to the study of the chosen language at the expense of fundamental conceptual skills. Relying on Eiffel helps us teach the concepts, not the specifics of a language.

The second approach is illustrated in particular by the famous MIT course based on the Scheme functional programming language [1], which has set the standard for ambitious curricula; there also have been attempts using Haskell, ML or OCaml. This method is strong on teaching the logical reasoning skills essential to a programmer. We strive to retain these benefits, as well as the relationship to mathematics, present here through logic and Design by Contract. But in my opinion object technology provides students with a better grasp of the issues of program construction. Not only is an O-O approach in line with the practices of the modern software industry, which has shown little interest in functional programming; more importantly for our pedagogical goals, it emphasizes system building skills and software architecture, which should be at the center of computer science education.

While, as noted, the curriculum should not be a slave to the dominant technologies just because they are dominant, using techniques too far removed from practice subjects us to the previously mentioned risk of disconnecting from the students, especially the most advanced ones, if they see no connection between what they are being taught and what their incipient knowledge of the discipline tells them. (Alan Perlis put this less diplomatically: *Purely applicative languages are poorly applicable*.)

Epigram #108.

I would argue further that the operational, imperative aspects of software development, downplayed by functional programming, are not just an implementation nuisance but a fundamental component of the discipline of programming, without which many of the most difficult issues disappear. If this view is correct, we are not particularly helping students by protecting them from these aspects at the beginning of their education, presumably abandoning them to their own resources when they encounter them later. (Put in a different way: functional programming seems to require monads these days and, given a choice, I'd rather teach assignment than category theory.)

> It is useful to point out that O-O programming is as mathematically respectable — through the theory of abstract data types on which it rests and, in Eiffel, the reliance on contracts — and as full of intellectual challenges as any other approach. Recursion, one of the most fascinating tools of functional programming, receives extensive coverage in the present book.

→ *Chapter* 14.

Some of the comments on functional programming also apply to the third approach, reliance on formal methods. As discussed above, a fully formal approach is, at the introductory programming level, premature. The practical effect may be to convince students that academic computer science has nothing to do with the practice of software engineering, and lead them to a jaded, method-less approach to programming.

The fourth commonly used approach, pioneered at ETH, draws its roots in the structured programming work of the seventies, and is still widespread. It emphasizes program structure and systematic development, often top-down. The supporting programming language is typically Pascal, or one of its successors such as Modula-2, Oberon or Ada. The approach of this book is heir to that tradition, with object technology viewed as a natural extension of structured programming, and a focus on programming-in-the-large to meet the challenges of programming in the new century.

TOPICS COVERED

The book is divided into five parts.

Part I introduces the basics. It defines the building blocks of programs, from objects and classes to interfaces, control structures and assignment. It puts a particular emphasis on the notion of contract, teaching students to rely on abstract yet precise descriptions of the modules they use, and to apply the same care to defining the interface of the modules they will produce. A chapter on "Just Enough Logic" introduces the key elements of propositional calculus and predicate calculus, both essential for the rest of the discussion. Back to programming, subsequent chapters deal with object creation and the object

→ *Chapter* 5.

structure; they emphasize the modeling power of objects and the need for our object models to reflect the structure of the external systems being modeled. Assignment is introduced, together with references and the tricky issues of working with linked structures, only after program structuring concepts.

Part II, entitled "How things work", presents the internal perspective. It starts with the basics of computer organization (covered from the viewpoint of a programmer and including essential concepts only), syntax description methods (BNF and its applications), programming languages and programming tools. The two chapters that follow cover core topics: syntax and how to describe it, including BNF and an introduction to the theory of finite automata; and an overview of programming languages, programming tools and software development environments.

Part III examines fundamental data structure and algorithm techniques. It is made of three chapters:

- Fundamental data structures — not a substitute for the "Data Structures and Algorithms" course which often follows the introductory course, but introducing genericity, algorithm complexity, and several important data structures such as arrays, lists of various kinds and hash tables.
- Recursion, including binary trees (in particular binary search trees), an introduction to fixpoint interpretations, and a presentation of techniques for implementing recursion.
- A detailed exploration of one interesting algorithm family, topological sort, chosen for its many instructive properties affecting both algorithm design and software engineering. The discussion covers the mathematical background, the progressive development of the algorithm for efficient execution, and the engineering of the API for convenient practical use.

Part IV goes into the depth of object-oriented techniques. Its first chapter covers inheritance, addressing many details seldom addressed in introductory courses, such as the Visitor pattern (which complements basic inheritance mechanisms for the case of adding operations to existing types). The next chapter addresses a technique that is increasingly accepted as a required part of modern object-oriented frameworks: function objects, also known as closures, delegates and *agents* (the term used here). It includes an introduction to lambda calculus. The final chapter in this part applies agent techniques to an important style of programming: event-driven computation. This is the opportunity to review another design pattern, Observer, and analyze its limitations.

Part V adds the final dimension, beyond mere programming, by introducing concepts of software engineering for large, long-term projects.

Appendices, already mentioned, provide an introduction to programming languages with which students should be familiar: Java, C#, C++ — a bridge between the C and O-O worlds — and C itself.

ACKNOWLEDGMENTS

A number of elements of this Instructor's Preface are taken from earlier publications: [7], [8], [9], [10], [12].

This book has its source, as noted, in the "Introduction to Programming" course at ETH Zurich and would not have been possible without the outstanding environment provided by ETH. Both the course and the book exist as a result of Olaf Kübler's trust (or wager) that in addition to entrepreneur I could also be a professor. Specific thanks go to the Rectorate (which financed the initial development of the Traffic library), to the Rector himself, Konrad Osterwalder, and to the computer science department, particularly Peter Widmayer who, as then department head, first asked me whether I would like to teach introductory programming, and made the effort of coordinating his own course with mine.

I have taught the course every Fall since 2003 and am indebted to the outstanding assistant team that has built an effective operation for handling exercise sessions, supporting students, devising exercises and exams, grading them, organizing student projects, writing supplementary documents and teaching aids, and on the odd occasion substituting for me in lectures. This has enabled me to concentrate on developing the pedagogical concepts and the core material, reassured that the logistics would work. I am also grateful to the hundreds of students who have taken this course, put up with my trials and errors, and provided feedback, including the best kind of feedback one can hope for: excellent software projects.

See e.g. games.ethz.ch.

The course assistants, 2003-2008, have been: Volkan Arslan, Stephanie Balzer, Till Bay, Karine Bezault (Karine Arnout), Benno Baumgartner, Rolf Bruderer, Ursina Caluori, Robert Carnecky, Susanne Cech Previtali, Stephan Classen, Jörg Derungs, Ilinca Ciupa, Ivo Colombo, Adam Darvás, Peter Farkas, Michael Gomez, Sebastian Gruber, Beat Herlig, Matthias Konrad, Philipp Krähenbühl, Hermann Lehner, Andreas Leitner, Raphael Mack, Benjamin Morandi, Yann Müller, Marie-Hélène Nienaltowski (Marie-Hélène Ng Cheong Vee), Piotr Nienaltowski, Michela Pedroni, Marco Piccioni, Conrado Plano, Nadia Polikarpova, Matthias Sala, Bernd Schoeller, Wolfgang Schwedler, Gabor Szabo, Sébastien Vaucouleur, Yi (Jason) Wei and Tobias Widmer. While I should cite virtually all members of the ETH Chair of Software Engineering for their support and ideas I must at least single out Manuel Oriol for his participation in our education research, Till Bay (for his development of the EiffelMedia library, the basis for so many student projects, of the EiffelVision drawables of Traffic in his diploma thesis, and of the Origo project hosting site at origo.ethz.ch as part of his PhD thesis), Karine Bezault, Ilinca Ciupa, Andreas Leitner, Michela Pedroni and Marco Piccioni (all of them head assistants at some point and helpful in many other ways). Claudia Günthart provided excellent administrative support.

The Traffic software has a particularly important role in the approach of this book. The current version was developed over several years by Michela Pedroni, starting from an original version written by Patrick Schönbach under the management of Susanne Cech Previtali; a number of students contributed to the software, supervised by Michela in various semester and master's projects, in particular (in approximate chronological order) Marcel Kessler, Rolf Bruderer, Sibylle Aregger, Valentin Wüstholz, Stefan Daniel, Ursina Caluori, Roger Küng, Fabian Wüest, Florian Geldmacher, Susanne Kasper, Lars Krapf, Hans-Hermann Jonas, Michael Käser, Nicola Bizirianis, Adrian Helfenstein, Sarah Hauser, Michele Croci, Alan Fehr, Franziska Fritschi, Roger Imbach, Matthias Loeu, Florian Hotz, Matthias Bühlmann, Etienne Reichenbach and Maria Husmann. Their role was essential in bringing the user perspective to the product, as most of them had previously taken the introductory course with early versions of Traffic. Michela Pedroni was also instrumental in reconciling the software with the book and the other way around and, more generally, in helping develop the underlying pedagogical approach — inverted curriculum, outside-in, tool support (see trucstudio.origo.ethz.ch). Marie-Hélène Nienaltowski also participated in our pedagogical work, provided the TOOTOR system to help students master the material, and tried out the approach at Birkbeck College, University of London.

I am grateful to my colleagues in the Computer Science Department (Departement Informatik) at ETH for many spirited discussions about the teaching of programming; I should acknowledge in particular the criticism and suggestions of Walter Gander (who also helped me improve an important numerical example), Gustavo Alonso, Ueli Maurer, Jürg Gutknecht, Thomas Gross, Peter Müller and Peter Widmayer. Beyond ETH, I benefited from many discussions with educators including Christine Mingins, Jonathan Ostroff, John Potter, Richard E. Pattis, Jean-Marc Jézéquel, Vladimir Billig, Anatoly Shalyto, Andrey Terekhov and Judith Bishop.

Like all my work of recent years, this book has a huge debt to the outstanding work of developing the EiffelStudio environment and libraries at Eiffel Software under the leadership of Emmanuel Stapf and with the participation of the entire development team. I am also grateful to the willingness of the ECMA International TC49-TG4 standard committee, in charge of the ISO Eiffel standard, to take into consideration the needs of beginning students when discussing improvements and extensions to the language design; the debt here is to Emmanuel Stapf again, Mark Howard, Éric Bezault, Kim Waldén, Zoran Simic, Paul-Georges Crismer, Roger Osmond, Paul Cohen, Christine Mingins and Dominique Colnet. Discussions on the Eiffel Software user list have also been most enlightening.

groups.eiffel.com.

Listing even a subset of the people whose work has influenced the present one would take many pages. Many are cited in the text itself but one is not: the presentation of recursion owes some of its ideas to the online record of Andries van Dam's lectures at Brown.

Many people provided comments on drafts of the book; I should in particular note Bernie Cohen (although his principal influence on this book occurred many years earlier, when he proposed the concept of inverted curriculum), Philippe Cordel, Éric Bezault, Ognian Pishev and Mohamed Abd-El-Razik, as well as ETH students and assistants Karine Bezault, Jörg Derungs, Werner Dietl, Moritz Dietsche, Luchin Doblies, Marc Egg, Oliver Jeger, Ernst Leisi, Hannes Röst, Raphael Schweizer and Elias Yousefi. Hermann Lehner contributed several exercises. Trygve Reenskaug contributed important and perceptive comments on the event-driven design chapter. I am particularly grateful for the extensive reading and error reporting that Marco Piccioni and Stephan van Staden performed on chapters of the last drafts.

Special thanks are due to the originators of the material from which the language-specific appendices is drawn: Marco Piccioni (Java, appendix A), Benjamin Morandi (C#, appendix B) and Nadia Polikarpova (C++, appendix C). I obviously remain responsible for any deficiency in the resulting presentations.

I cannot find strong enough words to describe the value of the extremely diligent proofreading of the final version by Annie Meyer and Raphaël Meyer, resulting in hundreds (actually thousands) of corrections and improvements.

Since so many people have helped I am afraid I am forgetting some, and will keep a version of this section online, correcting any omissions. I do want to end, however, by acknowledging the help and advice of Monika Riepl, from le-tex publishing services in Leipzig, on typesetting issues, and the warm and efficient support, throughout the publishing process, of Hermann Engesser and Dorothea Glaunsinger from Springer Verlag.

See touch.ethz.ch/ acknowledgments.

BM
Santa Barbara / Zurich, April 2009

Bibliography

[1] Harold Abelson and Gerald Sussman: *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, 1996.

[2] Bernard Cohen: *The Inverted Curriculum*, Report, National Economic Development Council, London, 1991.

[3] Mark Guzdial and Elliot Soloway: *Teaching the Nintendo Generation to Program*, in *Communications of the ACM*, vol. 45, no. 4, April 2002, pages 17-21.

[4] Joint Task Force on Computing Curricula: *Computing curricula 2001* (final report). December 2001, tinyurl.com/d4uand.

[5] Joint Task Force for Computing Curricula 2005: *Computing Curricula 2005*, 30 September 2005, www.acm.org/education/curric_vols/CC2005-March06Final.pdf.

[6] Leslie Lamport: *The Future of Computing*: *Logic or Biology*; text of a talk given at Christian Albrechts University, Kiel on 11 July 2003, research.microsoft.com/users/lamport/pubs/future-of-computing.pdf.

[7] Bertrand Meyer: *Towards an Object-Oriented Curriculum*, in *Journal of Object-Oriented Programming*, vol. 6, no. 2, May 1993, pages 76-81. Revised version in *TOOLS 11* (*Technology of Object-Oriented Languages and Systems*), eds. R. Ege, M. Singh and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 585-594.

[8] Bertrand Meyer: *Object-Oriented Software Construction*, *2nd edition*, Prentice Hall, 1997, especially chapter 29, "*Teaching the Method*".

[9] Bertrand Meyer: *Software Engineering in the Academy*, in Computer (IEEE), vol. 34, no. 5, May 2001, pages 28-35,se.ethz.ch/~meyer/publications/computer/academy.pdf.

[10] Bertrand Meyer: *The Outside-In Method of Teaching Introductory Programming*, in Manfred Broy and Alexandre V. Zamulin, eds., Ershov Memorial Conference, volume 2890 of Lecture Notes in Computer Science, pages 66-78. Springer, 2003.

[11] Christine Mingins, Jan Miller, Martin Dick, Margot Postema: *How We Teach Software Engineering*, in *Journal of Object-Oriented Programming* (JOOP), vol. 11, no. 9, 1999, pages 64-66 and 74.

[12] Michela Pedroni and Bertrand Meyer: *The Inverted Curriculum in Practice*, in *Proceedings of SIGCSE 2006* (Houston, 1-5 March 2006), ACM, se.ethz.ch/~meyer/publications/teaching/sigcse2006.pdf.

[13] Michela Pedroni, Manuel Oriol and Bertrand Meyer: *What do Beginning CS students know?*, submitted for publication, 2009.

[14] Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*, in *Proceedings of Tenth Australasian ComputingEducation Conference (ACE2008)*, Wollongong, January 2008), crpit.com/confpapers/CRPITV78Lister.pdf.

[15] Niklaus Wirth: *Computer Science Education*: *The Road Not Taken*, opening address at ITiCSE conference, Aarhus, Denmark, June 2002, www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm.

Web addresses come and go. All URLs appearing in this bibliography and the rest of the book were operational on April 19, 2009.

Note to instructors: what to cover?

To provide flexibility for the instructor, the book has more material than will typically be covered in a one-semester course. The following is my view of what constitutes essential material and what can be viewed as optional. It is based on my experience and will naturally need to be adapted to every course's specifics and every instructor's taste.

- Chapters 1 to 4 should probably be covered in their entirety, as they introduce fundamental concepts.

- Chapter 5 on logic introduces fundamental concepts. If students are also taking a logic course the material can be covered briefly, with a focus on relating computer scientists' and logicians' notations and conventions. I find it useful to insist on the properties of implication, initially counter-intuitive to many students ("Getting a practical feeling for implication", page 86); also, the course should discuss **semistrict boolean operators** (5.3), which logicians usually do not cover.

- Chapter 6 on object creation is necessary for the rest of the presentation.

- So is chapter 7 on control structures up to 7.6; the remaining sections present details of the low-level branching structure and some language variants. You should mention structured programming (7.8).

- Chapter 8 on routines should in my view be included in its entirety; in particular it is useful to provide a simple proof of the undecidability of the Halting Problem.

- In chapter 9, sections up to 9.5 cover fundamental concepts. 9.6, discussing the difficulty of programming with references, with the example of list reversal, is important but more advanced. The last subsection, on dynamic aliasing, is optional material.

- How much to cover chapter 10 on computers depends on what students are learning elsewhere about computer architecture. The chapter is not deep but provides basic points of reference for programmers.

- Chapter 11 on syntax is important material but not absolutely required for the rest of the book. I suggest covering at least the sections up to 11.4 (if only because students need to understand the concept of abstract syntax). If most students will *not* take a course on language and compilers, they will benefit from the basic concepts in subsequent sections.

- Chapters 12 on programming languages and tools is background material; I do not cover it explicitly in my class but provide it as a resource.

- Chapter 13 introduces fundamental concepts on data structures, genericity, static typing and algorithm complexity. It is possible to skip 13.8 (list variants) and 13.13 (iteration).

- Chapter 14 discusses recursion in some depth — more depth than is customary in an introductory presentation, because I feel it is useful to remove the potential mystery of recursive algorithms and show the importance of recursion *beyond* algorithms: recursive definitions, recursive data structures, recursive syntax productions and recursive proofs. The core material is the beginning of the chapter: 14.1 to 14.4, including the discussion of binary trees. The other sections may be viewed as supplementary; backtracking and alpha-beta (14.5) are a useful illustration of the applications of recursion. If the course is strongly implementation-oriented, consider 14.9 (implementing recursion); if you think that contracts are important, direct the students to 14.8 (contracts and recursion).
- Chapter 15 is a detailed discussion of an important application, topological sort. It introduces no new programming construct and so you can skip it, or replace it with one of your own examples, without damage. I cover it in some depth because it describes the complete progression from mathematics to algorithms to choice of optimal data structures to proper engineering of the API.
- In chapter 16, on inheritance, the essential sections are 16.1 to 16.7, plus 16.9 on the role of contracts, which illuminates the whole concept of inheritance. It is also useful to explain the connection to genericity in 16.12. The end of the chapter, in particular 16.14 about the Visitor pattern, is more advanced material that most courses probably will not have the time to cover, but which can be given as a reading assignment or as preparation for later courses.
- Chapter 17 on agents (closures, delegates) again goes beyond the usual scope of introductory courses. This is so important to modern programming that in my opinion it should be covered at least up to 17.4 (including illustrations through numerical programming and iteration). I usually do not have the time to cover 17.6, a gentle introduction to lambda calculus, but it should interest the more mathematically-oriented students, if only as extra reading material.
- If you do cover agents, you should then reap the benefits by covering the application to event-driven programming and especially GUI design (of interest to many students) in chapter 18. This is a good opportunity to learn an important pattern, Observer. Our course covers this and the previous chapter together, in four 45-minute lectures.
- Chapter 19 (introduction to software engineering) is not critical to an introductory course and I have not had time so far to cover it (but we do have "software architecture" and "software engineering" courses later in the curriculum). It is appropriate for an audience that needs to be exposed to the issues of production-quality software development in industry.
- The appendices are background material and I do not cover them, although some instructors might want to devote some time to a language such as Java or C++ (we do this, as noted, in specialized courses focusing on these languages).

A final note: while the course and the book were developed together, I always make a point of devoting a couple of lectures in the course to a topic *not* covered in the book — to introduce some spontaneity and avoid limiting the course to pre-packaged material. I like for example to present the algorithm for *Levenshtein distance* (edit distance between two strings), as it provides an outstanding example of the usefulness of loop invariants: without the invariant the algorithm looks like magic, with the introduction of the invariant it becomes limpid. Some of the extra material is available from the book site, touch.ethz.ch. (In the same vein, I have found that the textbook is sufficiently detailed to allow me to use a "Socratic" style for a couple of lectures in the semester: I ask the students to read a chapter in advance; then I do not cover the material sequentially in class but just come and wait for questions. Maybe this can work for other instructors as well.)

Contents

PART I:

Basics

In this introductory part we start our journey through programming with the essentials of programming: objects, classes, interfaces and contracts, and supporting concepts including logic and properties of hardware that every programmer must know.

1

The industry of pure ideas

1.1 Their machines and ours

Engineers design and build machines. A car is a machine for traveling; an electronic circuit is a machine for transforming signals; a bridge is a machine for crossing a river. Programmers — "software engineers" — design and build machines too. We call our machines *programs* or *systems*.

There is a difference between our machines and theirs. If you drop one of their machines, it will hurt your feet. Ours won't.

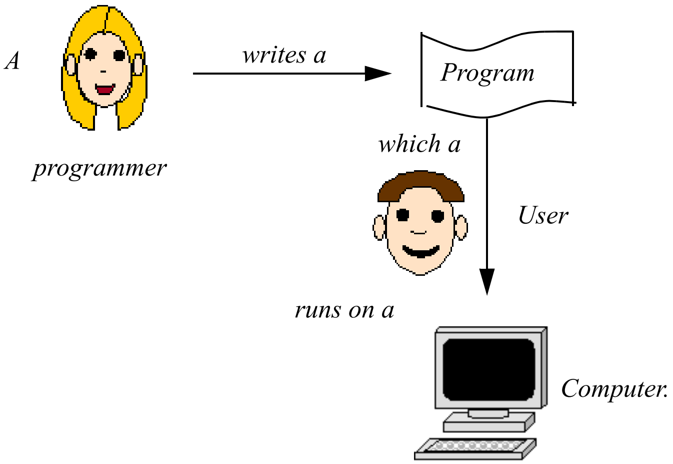
Programs are immaterial. This makes them closer, in some respects, to a mathematician's theorems or a philosopher's propositions than to an airplane or a vacuum cleaner. And yet, unlike theorems and propositions, they are engineering devices: you can operate a program, like you operate vacuum cleaners or planes, and get results.

Since one cannot operate a pure idea you will need some tangible, material support to operate programs or, using the more common terms, to *run* or *execute* them. That support is another machine: a **computer**. Computers and related devices are called **hardware**, indicating that — although they are getting ever lighter — computers are the kind of machine that will hurt your feet. Programs and all that relates to them are by contrast called **software**, a word made up in the 1950s when programs emerged as topic of interest.

Here is how things work. You dream up a machine, big or small, and describe your dream in the form of a program. The program can then be fed into a computer for execution. The computer by itself is a general-purpose machine, but when equipped with your program it becomes a specialized machine, a material realization of the immaterial machine that you defined through your program.

The person who writes the program — "you" in the previous paragraph — is predictably called a *programmer*. Others, known as *users*, can then run your program on your computer, or theirs.

From idea to results



If you have used a computer, you have already run some programs, for example to browse the Web or play a DVD, so you already are a user. This book should help you make it to the next step: programmer.

> Cynics in the software industry pronounce "user" as "*loser*". It's one of the goals of this book that users of your programs will pronounce themselves winners.

The immaterial nature of the machines we build is part of what makes programming so fascinating. Given a powerful enough computer you can define any machine you want, whose operation will require billion upon billion of individual steps; and the computer will run it for you. You do not need wood or clay or iron or a hammer or anything that could wear you out carrying it up the stairs, burn you, or damage your clothes. State what you want, and you will receive it. The only limit is your imagination.

All right, it is one of *two* limits; we avoid mentioning the other in genteel company, but you will likely encounter it before long; it is your own fallibility. Nothing personal: if you are like me and the rest of us, you make mistakes. Lots of mistakes. In ordinary life they are not all harmful, as most human activities are remarkably error-tolerant. You can press your fork a little too intensely, swallow water a little too fast, push the accelerator a little too hard, use the wrong word; this happens all the time and in most cases does not prevent you from achieving what you wanted: eat, drink, drive, communicate. But programming is different! At a dazzling speed — hundreds of millions of basic operations per second — the computer will run your machine description, your program, exactly as you prepared it. The computer does not "understand" your program, it just runs it; the slightest mistake will be faithfully carried out by the machinery. What you wrote is what you get.

As you learn about programming in the following chapters, this is perhaps the most important property of computers to keep in mind. You might still believe otherwise: because computer programs do things that seem so sophisticated — like finding, in less than a second, your ideal vacation deal from millions of offers available on the World-Wide Web — you may easily succumb to the impression that computers are smart. Wrong. Although some programs embody considerable human intelligence, the computer that runs them is like a devoted and insufferable servant: infinitely faithful, almost infinitely fast, and definitely stupid. It will carry out your instructions exactly as you give them, never taking any initiative to correct mistakes, even those a human being would find obvious and benign. The challenge for you, the programmer, is to feed this obedient brute with flawless instructions representing — in an execution of any significant program — billions of elementary operations.

In any experience you may have had with computers, you will have noticed that they do not always react the way you like. It does not take very long to experience a "crash", that state in which everything seems to disappear and execution stops. But except for the rare case of a hardware malfunction it wasn't the computer that crashed; it was a program that did not do the right thing, and behind the program it was a programmer who did not foresee all possible execution scenarios.

You cannot learn programming without going through this experience of programs — yours or someone else's — that do not work as they should; and you cannot become a professional programmer without learning the techniques that will let you build programs that *do* work as you want.

The good news is that it is possible to produce such programs, provided you use the proper tools and maintain discipline, attention to the big picture as well as the details, and dedication.

Helping you acquire this discipline is one of the main tasks of this book, which is an introduction not just to programming but to programming *well*. Note in particular, starting in the next chapter, the boxes labeled "Touch of Methodology" and "Touch of Style", where I have collected advice — learned over the years, sometimes the hard way — which will help you write software that works as you want it to.

1.2 THE OVERALL SETUP

In the next chapters we will jump straight into program development. We will not need much detailed knowledge about computers, but let us take a look at their fundamental properties, since they set the context for the construction of software.

→ Chapter 10, "Just enough hardware", has more about computers.

The tasks of computers

Computers — *automatic stored-program digital computers* to be precise — are machines that can store and retrieve information, perform operations on that information, and exchange information with other devices.

This definition highlights the major capabilities of computers:

What computers do

- **Storage and retrieval**
- **Operations**
- **Communication**

Storage and retrieval capabilities are a prerequisite for everything else: computers must be able to keep information somewhere before they can apply operations to it, or communicate it. Such a "somewhere" is called a **memory**.

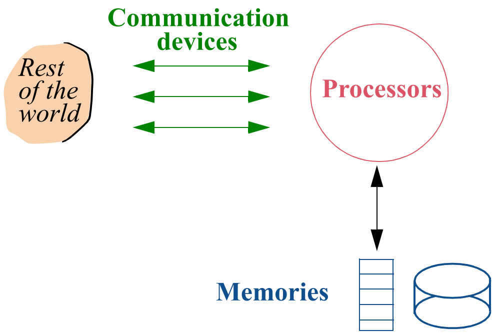
→ A more precise definition of "memory" appears below: page 10.

Operations include comparisons ("Are these two values the same?"), replacement ("Replace this value by that one"), arithmetic ("Compute the sum of these two values") and others. These operations are primitive; what makes computers able to perform amazing feats is not the intrinsic power of their basic mechanisms, but the *speed* at which they can carry them out and the ingenuity of the *humans* — you! — who write programs that will execute millions of them.

Communication allows us to enter information into computers, and retrieve information from them (the original information, or information that has been produced or modified by the computer's operations). It also enables computers to communicate with other computers and with devices such as sensors, phones, displays and many others.

General organization

The previous definition yields the basic schematic diagram for computers:



Components of a computer system

The **memories** hold the information. We talk of memories in the plural because most computers have more than one storage device, of more than one kind, differing by size, speed of access to information and *persistence* (affecting whether a memory retains information when power is switched off).

The **processors** perform the operations. Again there usually are several of them. Occasionally you will see a processor called a **CPU**, an acronym for the older term *Central Processing Unit*.

The **communication devices** provide means of interacting with the rest of the world. The figure shows the communication devices as interfacing with the processors rather than the memories; indeed, when exchanging information between a memory and the outside world, you will usually need to go through some operations of a processor. A communication device supports **input** (outside world to computer), **output** (the other way around), or sometimes both. Examples include:

- A keyboard, through which a person enters text (input).
- A video display or "terminal" (output).
- A mouse or joystick, enabling you to designate points on the terminal screen (input).
- A sensor, regularly sending measurements of temperature or humidity to a computer in a factory (input).
- A network connection to communicate with other computers and devices (input and output).

The abbreviation **I/O** covers both input and output. The words "input" and "output" are also used as verbs, as in "you must input this text".

Information and data

The key word in the above definition of computers is “information”: what you would like to store into memories and retrieve from them, process with the processors’ operations, and exchange through the communication devices.

This is the human view. Strictly speaking, computers do not directly manipulate information; they manipulate *data* representing that information:

Definitions: Data, information

Collections of symbols held in a computer are called *data*.

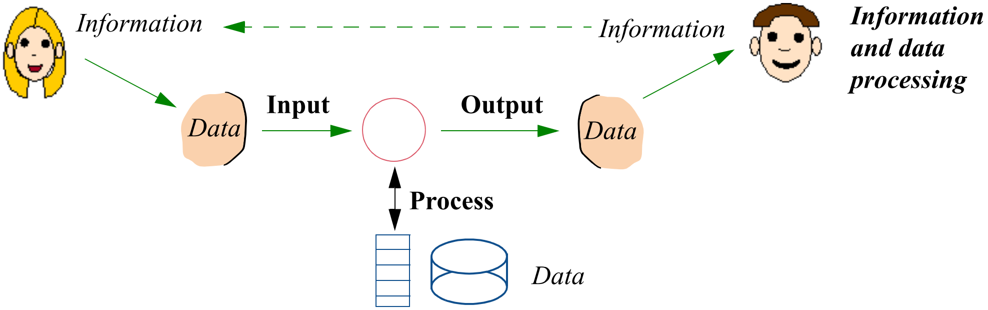
Any interpretation of data for human purposes is called *information*.

Some supercilious people will tell you that “data” should only be used in the plural, because it is originally the plural of “*datum*”. Thank them for the kindness of their advice and disregard it cheerfully. Unless they intend to continue the conversation in Latin, their grammatical data is obsolete.

Information is what you want: the day’s headlines, a friend’s picture, background on the speaker in today’s seminar. Data is how it is encoded for the computer.

As an example, the MP3 audio format, which you may have used to listen to music with the help of a computer, is a set of rules for encoding *information* about a piece of music into *data* that can be stored in a computer, exchanged across a network, and sent to an audio device so that it will replay the music.

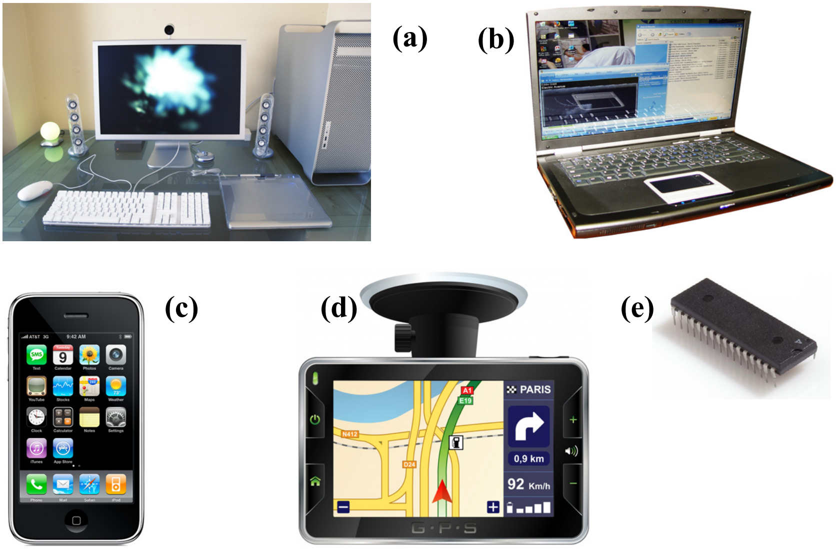
The data will be stored in memory. The task of the communication devices is to produce data from information coming from the world, store it in memory, and when the processors transform this data, or produce new data, send it out to the world so that it will understand it as information. Adapted to show the functions performed, the original picture becomes this:



The right-to-left arrow suggests that the process is not just one-way but repetitive, with information being fed back to yield new results.

Computers everywhere

The familiar picture of a computer is the “desktop” or “laptop” computer, whose processor and memory components are hosted in a box of a size somewhere between a textbook like this one and a big dictionary; the terminal is often the biggest part. All this is at human size. At *hand* size we find such devices as mobile phones, which today are essentially pocket computers with extended telecommunication capabilities. At the higher end, computers used for large scientific computations (physics, weather prediction...) can reach *room* size. This is of course nothing compared to computers of a generation ago, which took up *building* size for much more modest capabilities.



Computers: (a) desktop; (b) laptop; (c) iPhone (Apple); (d) GPS navigation system; (e) processor to be embedded.

Reduced to their central processor and memory components, computers can be much smaller than any of this. Increasingly, “the computer” is a device included — the technical term is **embedded** — in products or other devices. Today’s cars include dozens of small computers, controlling fuel delivery, braking, even windows. The printer connected to your desktop computer is not just a printing engine, it is itself a computer, able to produce fonts, smooth out images, restart with the first unprocessed page after a paper jam. Electric razors include computers; *manual* razors might include one some day. (The more expensive razor *blades* already contain electronic tracking tags to fight theft.) Washing machines contain computers, and in the future *clothes* may embed their own tiny computers, helping to tune the washing process.

The computers you will use for the exercises of this book are still of the keyboard-mouse-terminal kind, but keep in mind that software techniques have to cover a broader scope. Software for embedded systems must satisfy very high quality requirements: malfunctions in (for example) brake-control software can have terrible consequences, and you cannot fix them — as you would for a program running on your laptop — by stopping execution, correcting the error, and starting again.

The stored-program computer

A computer, as noted, is a universal machine: it can execute any program that you input into it.

> For this input process you will use communication devices, typically a keyboard and mouse. Text will appear on your screen as you type it, seemingly as a direct result, but this is an illusion. The keyboard is an input device, the terminal a distinct output device; echoing the input text on the screen requires a special program, such as a *text editor*, to obtain this input, process it and display it. Thanks to the speed of computers, this usually happens fast enough to give the illusion of a direct keyboard-screen connection; but if the computer responds more slowly, perhaps because it is running too many computation-intensive programs at the same time, you may notice a delay between typing characters and seeing them displayed.

When you input the program, where does it go? *Memories* are available to host it. This is why we talk of *stored-program* computers: to become a specific machine ready to carry out the specific tasks that you (as the programmer) have assigned to it, the computer will read its orders from its own memory.

The stored-program property of computers explains why we have not seen a proper definition of "memory" yet. It could have said that a memory is a device for storing and retrieving data; but this would require extending the notion of data to cover programs. It is clearer to keep the two notions separate:

Definition: Memory

A memory is a device for storing and retrieving data and programs.

The ability of computers to treat programs as data — *executable* data — explains their remarkable flexibility. At the dawn of the computer age, it led to visions of *self-modifying* programs (since a program can modify data, it can modify programs, including itself) and to some grand philosophizing about how programs would, through repeated self-modification, become ever more "intelligent" and take over the world. Closer to us but more prosaically and annoyingly, it is also the reason why email users are told to be careful about opening an email attachment, since the data it contains could be a maliciously written program whose execution might destroy other data.

For programmers, the stored-program property has a more immediate consequence: it makes programs amenable, like data of any other kind, to various transformations, performed by other programs. In particular, *the program you write is usually not the program you run*. The operations that a processor can execute are designed for machines, not humans; using them directly to construct your programs would be tedious and error-prone. Instead you will:

- Write programs in notations designed for human consumption, called *programming languages*. This form of a program is called its **source** text (or source form, or just source).
- Rely on special programs called *compilers* to transform such human-readable program texts into a format (their **target** form) appropriate for processor execution.

→ Or "machine code", or "object form".

We will often encounter the following terms reflecting this division of tasks:

Definitions: Static, Dynamic

Static properties of a program are properties of its source text, which can be analyzed by a compiler.

Dynamic properties are those characterizing its individual executions.

The details of all this — processor codes, programming languages, compilers, examples of static and dynamic properties — appear in later chapters. What matters for the moment is knowing that the programs you are going to write, starting with the next chapter, are meant for people as well as for computers.

This human aspect of programming is central to the engineering of software. When you program you are talking not just to your computer but also to fellow humans: whoever will be reading the program later, for example to add functions or correct a mistake. This is a good reason to worry about program readability; and it is not just a matter of being nice to others, since that "whoever" might be you, a few months older, trying to decipher what in the world you had in mind when writing the original version.

Throughout this book, the emphasis is not only on practices that make your programs good for the computer (such as *efficiency* techniques ensuring they run fast enough), but also on practices that make programs good for human readers. Program texts should be understandable; programs should be *extendible* (easy to change); program elements should be *reusable*, so that when later on you are faced with a similar problem you do not have to reinvent the solution; programs should be *robust*, protecting themselves against unexpected input; most importantly, they should be *correct*, producing the expected results.

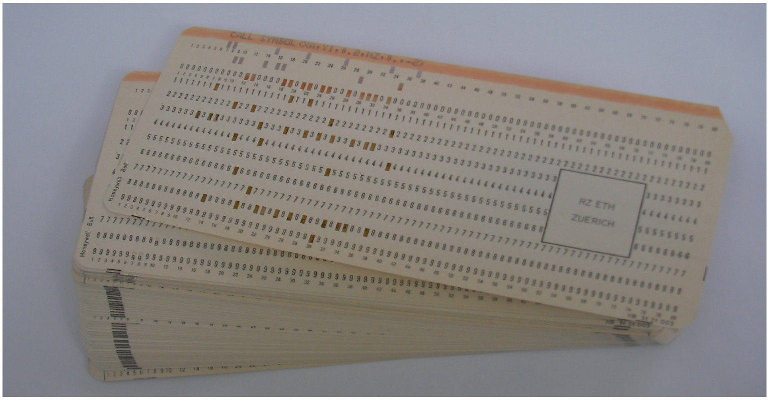
→ The final chapter contains a more detailed discussion of quality factors: "Components of quality", 19.3, page 705.

Touch of folk history: It’s all in the holes

Aerospace industry old-timers tell the story of the staff engineer who, in an early rocket project, was in charge of tracking the weight of everything that would get on board. He kept pestering the programmers about how much the control software would weigh. The reply, invariably, was that the software would weigh nothing at all; but he was not convinced.

One day he came into the head programmers’ office, waving a deck of punched cards (the input medium of the time, see the picture): “This is the software”, he said, “Didn’t I tell you it had a weight like everything else!”. This did not deter the programmer: “See the holes? They are the software.”

(Possibly apocryphal, but a good story still.)



A deck of punched cards

1.3 KEY CONCEPTS LEARNED IN THIS CHAPTER

- *Computers* are general-purpose machines. Providing a computer with a *program* turns it into a special-purpose machine.
- Computer programs process, store and communicate *data* representing *information* of interest to people.
- A computer consists of *processors*, *memories* and *communication devices*. Together these material devices make up *hardware*.
- Programs and associated intellectual value are called *software*. Software is an engineering product of a purely intellectual nature.
- Programs must be stored in memory prior to execution. They may have several forms, some readable and intended for human use, others directly processable for execution by computers.

- Computers appear in many different guises; many are *embedded* in products and devices.
- Programs must be written to facilitate understanding, extension and reuse. They must be correct and robust.

New vocabulary

At the end of every chapter you will find such a list. Check (this is the first exercise in the chapter) that you know the meaning of each term listed; if not, find its definition, as you will need the terms in subsequent chapters. To find a definition, look up the index, where definition pages appear in bold.

→ *Page 849.*

Communication device	**Compiler**	**Computer**
Correct	**CPU**	**Data**
Dynamic	**Embedded**	**Extendible**
Hardware	**Information**	**Input**
Memory	**Output**	**Persistence**
Processor	**Programmer**	**Programming language**
Reusable	**Robust**	**Software**
Source	**Static**	**Target**
Terminal	**User**	

1-E EXERCISES

1-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

1-E.2 Data and information

For each of the following statements, say whether it characterizes data, information or both (explain):

1 “You can find the flight details on the Web.”

2 “When typing into that field, use no more than 60 characters per line.”

3 “Your password must be at least 6 characters long.”

4 “We have no trace of your payment.”

5 “You can’t really appreciate her site without the Flash plug-in.”

6 “It was nice to point me to your Web page, but I can’t read Italian!”

7 “It was nice to point me to your Web page and I’d like to read the part in Russian, but my browser displays Cyrillic as garbage.”

1-E.3 Defining precisely something you have known all along

You know about alphabetical order: the order in which words are listed in a dictionary or other "*alphabetical*" list. Alphabetical order specifies, of two different words, which is "**before**" the other. For example the word *sofa* is before *soft*, which itself is before *software*.

The question you are asked in this exercise is simply:

Define under what exact conditions a word is alphabetically "before" another.

That is to say, define alphabetical order. This is a notion that you undoubtedly know how to apply in practice, for example to look up your name in a list; what the exercise requests is a *precise* definition of this intuitive knowledge, of the kind you might need for a mathematical notion — or for a concept to be implemented in a program.

To construct your definition you may assume that:

- A **word** is a sequence of one or more letters. (It is also OK to use "*zero* or more letters", accepting the possibility of empty words, if you find this more convenient. Say which convention you are using.)
- A **letter** is one among a finite number of possibilities.
- The exact set of letters does not matter, but for any two letters it is known which one is "**smaller**" than the other. For example, with letters of the Roman alphabet, *a* is smaller than *b*, *b* is smaller than *c* and so on.

  If you prefer a fully specified set of letters, just take it to include the twenty-six used in common English words, lower-case only, no accents or other diacritical marks: *a b c d e f g h i j k l m n o p q r s t u v w x y z*, each "smaller" than the next.

The problem calls for a definition, not a recipe. For example, an answer of the form "You first compare the first letters of the two words; if the first word's first letter is smaller than the second word's first letter then the first word is before the second, otherwise..." is **not** acceptable since it is the beginning of a recipe, not a definition. A proper answer may start: "*A word w1 is before a word w2 if and only if any of the following conditions holds: …*".

Make sure that your definition covers all possible cases, and respects the intuitive properties of alphabetical ordering; for example it is not possible to have both *w1* before *w2* and *w2* before *w1*.

About this exercise: The purpose is to apply the kind of precise, non-operational reasoning essential in good software construction. The idea is borrowed from a comment of Edsger Dijkstra, a famous Dutch computer scientist.

1-E.4 Anthropomorphism

Match the components and functions of a computer system to a human's body parts and their functions; discuss similarities and differences.

← *"The overall setup", 1.2, page 6.*

2

Dealing with objects

You are now going to write, execute and modify your first program.

Prerequisites: you must be able to use the basic functions of a computer and find your way through its directories and files; EiffelStudio must have been installed on your computer; and you must have downloaded the Traffic software. Everything else you will learn here and on the supporting Web site.

To download Traffic: traffic.origo.ethz.ch. Book site: touch.ethz.ch.

2.1 A CLASS TEXT

Each of the first few chapters relies on a different "system" (a collection of files making up a program) included in the Traffic delivery. The name of the system is reminiscent of the chapter's title: *objects*, *interfaces*, *creation*… In the *example* directory of the delivery, each system appears in a subdirectory, whose name also includes the chapter number so that they appear in order: *02_object*, *04_interfaces* and so on.

Start EiffelStudio and open the "system" called *objects*. The precise details of how to do this are given in the EiffelStudio appendix:

→ See "Setting up a project", E.2, page 844.

> ***Touch of practice*: Using EiffelStudio**
>
> Since this book focuses on principles of software construction, the details of how to use the EiffelStudio tools to run the examples appear separately in appendix E: "Using the EiffelStudio environment", page 843 and the associated Web page. To set up and run any example, read that appendix.

In case something goes wrong at any time, remember this:

> ***Touch of practice*: If you mess up**
>
> It is possible, especially if you are not too experienced with computers, to make a mistake that will take you off the track carefully charted below. Try to avoid getting into that situation (by following the instructions precisely) but if it happens don't panic; just check the EiffelStudio appendix.

You will be looking at program texts both throughout this book and on your screen. The book applies systematic typesetting conventions:

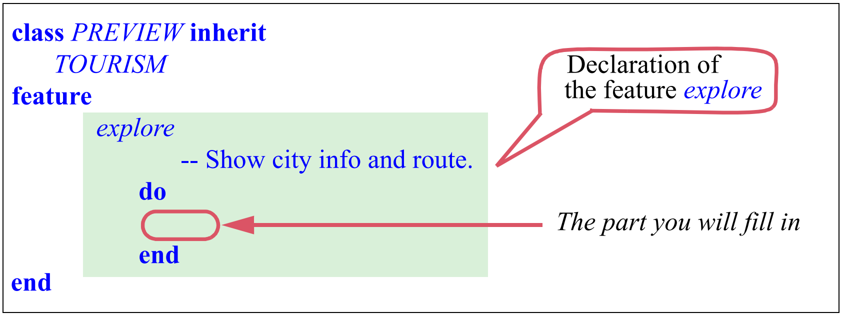
Touch of style: Program text and explanation text

In this book, anything that is part of a program text appears in this blue (sometimes **bold** or *italics* according to precise rules specified below). Everything else is the book's explanations. This way you will never confuse elements of the programs with observations about these programs.

These typesetting conventions are standardized. So are the conventions for displaying software texts in EiffelStudio; they are very similar, but you will notice a few differences since paper and screen have different constraints.

You are going to work on a program element, or "class", called *PREVIEW*, which will be the core of your first program. Bring up the text of class *PREVIEW*. The initial display will look like this:

→ See again : E.2, page 844 on how to bring up the class.



```
class PREVIEW inherit
    TOURISM
feature
        explore
                -- Show city info and route.
            do

            end
end
```

The first line says you are looking at a "class", one of those immaterial machines out of which we build programs; it calls it *PREVIEW*, as indeed the class describes a small preview of a city tour.

The first two lines also state that *PREVIEW* will **inherit** from an existing class called (second line) *TOURISM*; this means that *PREVIEW* extends *TOURISM*, which already has lots of useful facilities, so all you have to do is include your own programming ideas in the new class *PREVIEW*. The class names reflect this relationship: *TOURISM* describes a general notion of touring the city; *PREVIEW* covers a particular kind of tour, not a real visit but a preview from the comfort of your desk.

Touch of Magic?

Class *TOURISM* is part of supporting software prepared specifically for this book. By piggybacking on these predefined facilities, rather than building everything from scratch, you can immediately learn the most commonly useful programming concepts, and practice them right away by writing and running example programs.

So if it seems like magic that your first programs will work at all, it is not: the supporting software — the apparent "magic" — uses the same techniques that you will be learning throughout the book. Little by little we will be removing pieces of the magic, and at the end there will not be any left; you will be able to reconstruct everything by yourself if you wish.

Even now, nothing prevents you from looking at the supporting software, for example class *TOURISM*; it is all in the open. Just do not expect to understand everything yet.

The text of a class describes a set of operations, called *features*. Here there is only one, called *explore*. The part of the class that describes it is called the **declaration** of the feature. It consists of:

- The feature's name, here *explore*.
- "-- Show city info and route.", a *comment*.
- The actual content of the feature, enclosed in the *keywords* **do** and **end**, but empty for the moment: this is what you are going to fill in.

A **keyword** is a special word that has a reserved meaning; you may not use it for naming your own classes and features. To make keywords stand out we always show them in **bold** (blue, since they are part of program text). Here the keywords are **class**, **inherit**, **feature**, **do** and **end**. (With just these five you can already go quite a way.)

A **comment**, such as -- Show city info and route, is explanatory text that has no effect on the program execution but helps *people* understand the program text. Wherever you see "--" (two consecutive "minus" signs), it signals a comment, extending to the rest of the line. When you write a feature declaration you should **always**, as a matter of good style, include a comment after the first line as here, explaining what the feature is about.

2.2 OBJECTS AND CALLS

Your first program will let you prepare a trip through a city that looks remarkably like Paris, which may be the reason why the program text calls it *Paris*. As this is your first trip let's play it safe. All we want the program to do is display some information on the screen:

- First, display a map of Paris, including a map of the Metro (the underground train network).
- Next, spotlight, on the map, the position of the Louvre museum (you have heard about it, or maybe it's the only local name that you can pronounce at the moment).
- Next, highlight, on the Metro map, one of the metro lines — Line 8.
- Finally, since your ever thoughtful travel agent has prepared a route for your first trip through the city, animate that route by showing a small picture of a traveler hopping through the stops.

Editing the text

Programming time!
Your first program

In this section you are asked to fill in your first program text, then to run the program.

Here is what you should do. Edit the text of the class *PREVIEW* and modify the feature *explore* so that it reads like this:

```
explore
        -- Show some city info.
    do
        Paris.display
        Louvre.spotlight
        Line8.highlight
        Route1.animate
    end
```

The text you should type in

To make the process smoother and avoid any confusion, note the following about how to "input" (type in) the text:

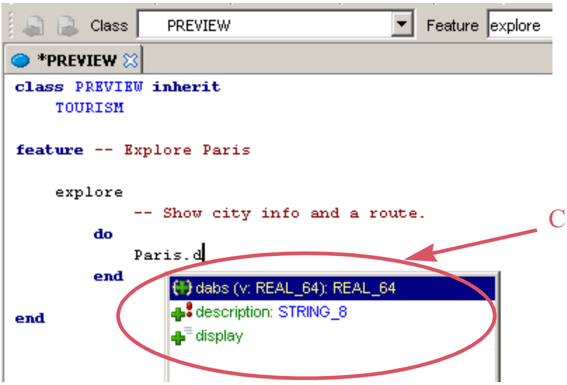
- The text of each line starts some distance away from the left margin; this is known as **indentation** and serves to show the structure of the text. As it has no effect on program execution, you could write everything left-aligned if you wanted to; but it does have an effect on program understandability (and probably on your grade when you submit programs), so please observe it carefully. We will see the indentation rules as we go.

- To achieve the indentation, do not use repeated spaces, which could make it messy to align text; use the character marked *Tab* on your keyboard. Tabs automatically align to equally spaced positions.

- In *Paris*.*display* and similar notations on subsequent lines, you see a period "**.**" between successive words. Unlike the period that terminates a sentence in written English, it is not followed by a space. Since it is an important element of program texts this book shows it as a big blue dot, "**.**", but on your keyboard it is just the plain period character.

- More generally, the typographical variations — boldface, italics, color … — do not affect how you type the text, only how you *read* it, in this book and on the screen as displayed by EiffelStudio.

Also note that you do not actually need to type everything; EiffelStudio has a "*completion*" mechanism which suggests possibilities for continuing any initial text that you have typed. For example, if you type *Paris*., EiffelStudio displays, immediately after you type the dot, a menu of possibilities, corresponding to the various features applicable to *Paris*. You could scroll down to find *display*, but this is not so convenient because the list is still too long, so type one more letter, the *d* of *display*; the menu gets updated to list those features whose name starts with a *d*:



```
class PREVIEW inherit
    TOURISM

feature -- Explore Paris

    explore
            -- Show city info and a route.
        do
            Paris.d
        end

end
```

Completion menu

Here you see *display* as one of the possibilities and can choose it either by clicking it or by moving through menu entries with the up and down arrow keys of the keyboard (here, press the down arrow twice) and pressing the Enter key. The completion menu automatically appears under certain contexts, such as when you typed a period after *Paris*; if at any point you are looking for help with possible completions and the menu does not show up, just type CTRL-Space (hold down the Control key while pressing the space bar) to get it.

If you are not interested in automatic completion, just continue typing and ignore the completion menu, except possibly as suggestions of what you may type. Typing the ESC key will dismiss the menu.

After typing the changes to the text you may *save* them (to make sure they are recorded for good); you can use the Save entry of the File menu, or just type Control-S (pressing S while holding the Control key down). You need not worry about forgetting to save; EiffelStudio will tell you if needed.

Running your first program

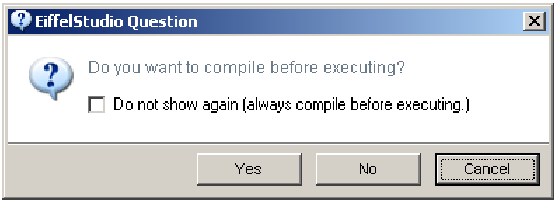
So much for the “cosmetics”, as programmers say — superficial aspects of a program’s textual appearance.

You will now run (execute) the program. The following describes the basic process; you will find more details of how to interact with EiffelStudio in the corresponding appendix.

→ *“Setting up a project”, E.2, page 844.*

Click the Run button (it is towards the right at the top of the window and looks like this ▶ Run). The first time you do this, you will actually get the following message (“dialog box”):

You must compile before executing


EiffelStudio Question

Do you want to compile before executing?

□ Do not show again (always compile before executing.)

“Compiling” a system means transforming it into a form that can be directly processed by the computer, as opposed to the original, or “source” form in which you wrote it.

This is a completely automatic process. Some people prefer to forget about it, and pretend that they just run the program directly after a change; hence the checkbox (“Do not show again…”), which will avoid being bothered in the future. Personally I do not check the box as I prefer to start my compilations explicitly (by clicking the “compile” button Compile ▾ , or just hitting the F7 function key). You can decide your own preference later; for the moment just click **Yes**.

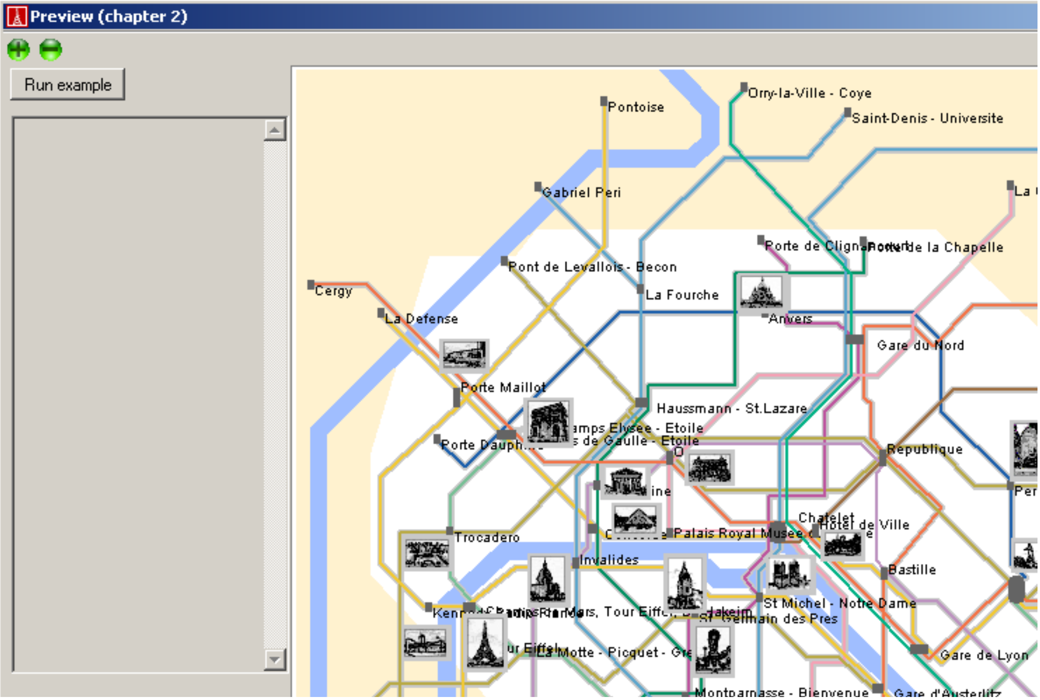
> If you have forgotten to save your changes EiffelStudio will detect it and bring up a similar message. Again there is a "Do not show again" checkbox; here I prefer not to save my files explicitly, letting EiffelStudio do it. In any case, save now if you need to.

The compilation starts. EiffelStudio must compile not just your class *PREVIEW* with its single feature *explore* but everything else that it needs — the entire Traffic software, and the supporting libraries. Depending on your initial setup — whether or not you have a *precompiled* version of this software — this might take a while, but only the first time; later compilations will only process your latest changes, so they will be almost instantaneous, even if the overall program is very large.

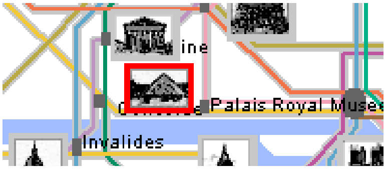
Unless you mistyped something — in which case you will get an error message so that you can correct the mistake and restart the process — compilation will proceed to the end and execution will start.

The first screen of the system appears. In this screen, click Run example ; you will see the following sequence of events:

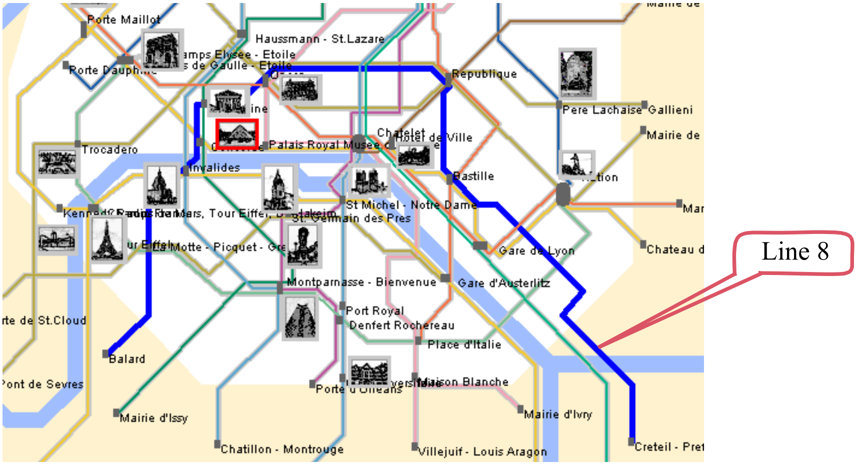
1 As a result of executing the first line, *Paris*.*display*, of our *explore* feature, the city map including the metro network appears in the window:



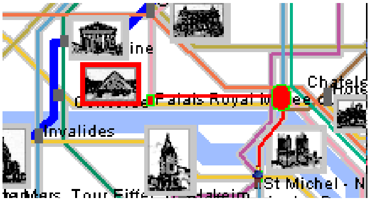
2 Nothing happens for five seconds, then as a result of the second line *Louvre*.*spotlight* the position of the Louvre museum (next to the Palais Royal metro station) shows up spotlighted on the map:



3 After another five seconds, Line 8 of the metro network comes up highlighted as a result of the third line *Line8*.*highlight*:



4 After another short delay, the fourth line *Route1*.*animate* causes the map to show a figurine representing a person and move it through the successive stops along the chosen route.



Once ready, a program can of course be executed as many times as you like, so you can repeat the above execution by pressing again the ▶ Run button of EiffelStudio. If you have not changed the program this will simply run it again. If you changed it, EiffelStudio will recompile it (after asking for your confirmation unless you changed the settings to start a compilation automatically) before running it again.

Dissecting the program

The execution just described is the effect of the four lines that you inserted into the text of the feature *explore*. Let us look at what they mean. The techniques used in this simple program are fundamental; make sure that you understand everything in the following explanation.

The first line,

```
Paris.display
```

uses an **object**, known to the program as *Paris*, and a **feature**, known as *display*. An object is a unit of data (the next section explains this notion in more detail); a feature is an operation applicable to such data. *Paris.display* is an example of a fundamental program construct known as a **feature call**:

```
x.f
```

where *x* denotes an object and *f* a feature (an operation). This has a well-defined effect:

Touch of Semantics: Feature call

The execution-time effect of a feature call *x.f* is to apply the feature of name *f*, from the corresponding class, to the object that *x* denotes at that moment in execution.

Previous rules addressed the form, or *syntax*, of programs. This the first rule defining *semantics* — the execution-time behavior of programs. We will study these concepts in detail in later chapters.

Feature call is the basis of computation: over and again, that is what our programs do during their execution.

In our example the target object is called *Paris*. As the name suggests, it represents a city. How much of the real city "Paris" does it really describe? You need not worry since *Paris* has been predefined for us. Pretty soon you will learn to define your own objects, but for the moment you have to rely on those prepared for this exercise. A standard convention facilitates recognizing them:

Touch of style: Names of predefined objects

Names of predefined objects always start with an upper-case letter, as in *Paris*, *Louvre*, *Line8* and *Route1*.

New names, corresponding to the objects that you define, will by default start with a lower-case letter.

Where are these "predefined" objects defined? You guessed it: in the class *TOURISM*, which your class *PREVIEW* **inherit**s. This is where we put the "magic" through which your program, simple as it is, can produce significant results.

One of the features applicable to an object representing a city, such as *Paris*, is *display*, which shows the current state of the city on the screen.

After applying *display* to the object *Paris*, the program performs another feature call:

```
Louvre.spotlight
```

The target object here is *Louvre*, another predefined object (name starting with a capital letter) denoting the Louvre museum. The feature is *spotlight* which will spotlight the corresponding place on the map.

Then to highlight Line 8 we execute

```
Line8.highlight
```

using a feature *highlight* that highlights the target object, here *Line8* denoting an object that represents line number 8 of the underground system.

The final step, again a feature call, is

```
Route1.animate
```

where the target object is *Route1*, representing a predefined route — we assume, as noted, that it was all prepared by your travel agent — and the feature is *animate* which will showcase the route by moving a figurine along it.

For the program to work as expected, the features used in this program — *display*, *spotlight*, *highlight*, *animate* — must all do a little more than just displaying something on the screen. The reason is that computers are fast, *very* fast. So if the only effect of the first operation, *Paris*.*display*, were to display the map of Paris, the next operation, *Louvre*.*spotlight*, would follow a fraction of a second later; when you run the program you would never see the first display, the one that shows the map without the Louvre. To avoid this, the features all make sure, after displaying what they need to display, to pause execution for five seconds.

This is all taken care of in the text of these features, which we are not showing you yet (although you can look at them if you want to).

Congratulations! You have now written and run your first program, and you even understand what it does.

2.3 WHAT IS AN OBJECT?

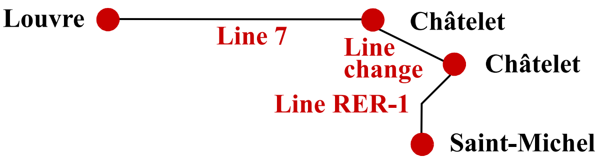
Our example program works with *objects* — four of them, called *Paris*, *Louvre*, *Line8* and *Route1*. Working with objects is what all our programs will do; this notion of object is so fundamental that it gives its name to a whole style of programming, used in this book and widely applied in the software industry today: *Object-Oriented*, often abbreviated as "O-O".

Objects you can and cannot kick

What exactly should we understand from the word "object"? Here we are using for technical purposes a term from ordinary language — very ordinary language, since it is hard to think of a more general notion than objects. Anyone can immediately relate to this word; this is both appealing and potentially confusing:

- It is appealing because using "objects" for your programs lets you organize them as models of real systems using real objects. If you do go to Paris you will see that the Louvre is a real object; if its sight is not enough to convince you of its reality, try kicking it with your fist. (Buying this book does not entitle you to a refund of medical expenses.) Our second software object so far, *Paris*, also corresponds to a real object, an even bigger one, the whole city.
- But this convenience of using software "objects" to represent physical ones should not lead you to confuse the two kinds. The reality of a software object does not extend beyond an immaterial collection of data stored in your computer; your program may have set it up so that operations on it represent operations on a physical object — like *Bus48*.*start*, representing the operation of making a bus move — but the connection is all in your mind. Even though our program uses an object called *Paris*, it is not the real Paris. ("One cannot put Paris into a bottle", says, more or less, an old French proverb, and you cannot put Paris into a program either.)

Never forget that the word "object" as used in this book denotes a software notion. Some software objects represent things from the world out there, like the Louvre, but as we move to more sophisticated programming techniques that will not always be the case. For example, the last object we used, called *Route1* represents a route — a travel plan. The particular plan represented by *Route1* enables you to go by metro (underground) from the Louvre station (also known as Palais Royal) to Saint-Michel. As the black line shows in the figure, this route has three parts, or "*legs*":



A metro route

- Go from the "Louvre" station to "Châtelet" on line 7 (3 stops).
- Change lines.
- Go from Châtelet to "Saint-Michel" on line RER-1 (1 stop).

The "route" is this sequence of legs. It is not a physical object that you can kick, like the Louvre or your little brother; but it is an object all the same.

Features, commands and queries

What makes an object is not that it has a physical counterpart, but that we can manipulate it with our program through a set of well-defined operations, which we call **features**.

Some of the features applicable to a "route" object include *questions* that we may ask; for example:

- What is the starting point? What is the ending point? (For our example *Route1*, as described above: Louvre and Saint-Michel.)
- What kind of route is it: walking, by bus, by car, by metro, or some combination of these? (For *Route1* the answer is: a metro route.)
- How many legs does it use? (For *Route1*, three: metro from Louvre to Châtelet, line change at Châtelet, metro from Châtelet to Saint-Michel.)
- What metro lines, if any, does it use? (For *Route1*: lines 7 and RER-1.)
- How many metro stations does it go through? (Here: three altogether.)

Such features, allowing us to obtain properties of an object, are called **queries**.

There is a second kind of feature, called a **command**; a command enables the program to change the properties of some objects. We already used commands: in our first program, *Paris*.*display* changes the image shown on the screen, so *display* is a command. In fact all four operations of our first program were commands. As another set of examples, we may want to define the following commands on routes:

- Remove the first leg of the route, or the last leg, or any other.
- "Append" (add at the end) a new leg; it must start at the current destination. Here we can append to *Route1* a new leg provided it starts at Saint-Michel, for example a metro leg from Saint-Michel to Port Royal (1 station on the simplified map); the route will be changed to involve 4 legs, 3 metro lines, and 4 stations; the result now starts at Louvre and ends at Port Royal.

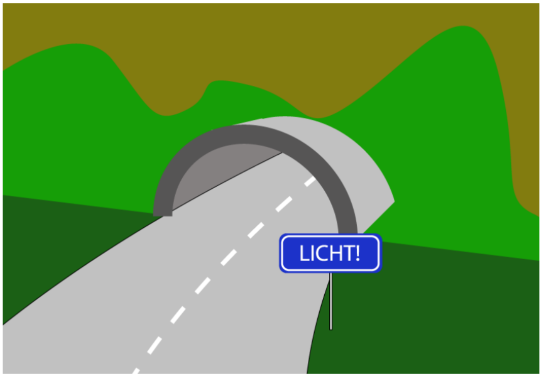
← *The map was on page 22.*

- "Prepend" (add at the beginning) a new leg; it must end at the current origin. For example we can make *Route1* start with a leg going from *Opéra* to Louvre; this changes the number of stations but not the set of metro lines since Opéra is already on line 7.

All these operations change the route, and hence are commands.

We can, by the way, define precisely what it means for a command to "*change*" an object: it changes the visible properties of the object — visible, that is, through the queries. For example if you ask for the number of legs in a route (a query), then append a leg (a command), then ask again, the new answer will be one more than the original. If the command is to *remove* a leg, the query's result afterwards will be one *less* than it was before the command.

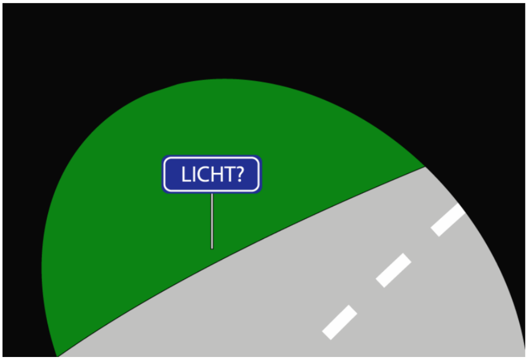
The tunnel signs that one encounters on German *Autobahnen* (freeways) are a good illustration of the command-query distinction. The sign at the entrance to a tunnel looks like this:



Command upon entering a tunnel

"*Licht*!", you are told in no uncertain terms. Switch on your lights! Unmistakably a command.

When you exit the tone is more gentle:



Query upon leaving a tunnel

"*Licht*?": did you remember to switch off your lights? Just a query.

This query is a nice example of "user interface design", resulting from careful research and intended to avoid common mistakes — as should also be the case for the user interface of software systems. Apparently, before it went into effect, the exit sign was a command, "Switch off your lights!", which disciplined drivers would mechanically obey — including at night. Nothing like a good query to keep the audience awake.

Objects as machines

The first thing we learned about programs is that they are machines. Like any complex machine, a program during its execution is made of many smaller machines. Our objects are those machines.

← *"The industry of pure ideas", 1, page 3.*

Perhaps you find this hard to visualize: how can we see a travel route across the metro as a *machine*? But in fact we just saw the answer: what characterizes a machine is the set of operations — commands and queries — that it provides to its users. Think of a DVD player, with commands to start playing, move to the next track and stop playing, and queries to show the number of the track being played, the time elapsed etc. To our programs, the *Route1* object is exactly like the DVD player: a machine with commands and queries.

The figure evokes this correspondence: blue rectangular buttons on the left represent commands; yellow elliptical buttons on the right represent queries.

animate
append
prepend
first
last
count
stations

A "route" object pictured as a machine

When thinking about objects — such as the one denoted by *Route1* — we now have two perspectives:

1 The object covers a certain collection of data in memory, describing, in this case, all the information associated with a certain route — it has three legs, it starts at the station "Louvre" etc.

2 The object is a machine, providing certain commands and queries.

These two views are not contradictory, but easy to reconcile: the operations that the machine provides (view 2) access and modify the data collected in the object (view 1).

Objects: a definition

Summarizing this discussion of objects, here is the precise definition that serves throughout this book:

> ### *Definition*: Object
>
> An object is a software machine allowing programs to access and modify a collection of data.

In this definition and the rest of the discussion, to "access" data is to obtain the answer to a question about the data, without modifying it. (We could also say "consult" the data.)

The words "access" and "modify" reflect the already encountered distinction between two fundamental kinds of operation:

> ### *Definitions:* Feature, Query, Command
>
> An operation that programs may apply to an object is called a **feature**, and:
>
> - A feature that *accesses* an object is called a **query**.
> - A feature that may *change* an object is called a **command**.

Examples of commands were *display* for a city such as *Paris* and *spotlight* for a location such as *Louvre*. Queries have been mentioned, for example the starting point of a route, but we have not yet used one in a software text yet.

Queries and commands work on existing objects. This means we need a third kind of operation: *creation* operations, to give us objects in the first place. You do not need to worry about this for the moment because all the objects you need in this chapter — *Paris*, *Louvre*, *Route1* … — are created for you as part of the "magic" of class *TOURISM*, and at execution time they will be created when your program needs to use them. Soon you will learn how to create your own objects as you please. → *Chapter 6*.

This will also explain why the notion of "machine" characterizes not only objects but also classes.

2.4 FEATURES WITH ARGUMENTS

Queries are just as important as commands. We will now look at some examples of how to use them. We may for example want to know the starting point — the origin — of a route. It is given by a query *origin*, applicable to routes; its value for our example route *Route1* is written

```
Route1.origin
```

This is a feature call, like the calls to commands such as *Route1. animate* and the others we have seen. In this case, since the feature is a query, the call does not "do" anything; it simply yields a value, the origin of *Route1*. We could use this value in various ways, like printing it on a piece of paper; let us instead display it on the screen.

You will have noticed at the bottom of the display a little window (rectangular area) marked "Console"; it is used to show information about the state of our city-modeling system. In our program it is — guess what — an object. You can manipulate it through the feature *Console*; it is one of those predefined features, like *Paris* and *Route1*, that our example class *PREVIEW inherits* from *TOURISM*.

← See class PREVIEW, page 16.

One of the commands applicable to *Console* is called *show*; its effect is to output (display) a certain text in the console. Here we may use it to show the name of the starting point of the route.

Programming time!

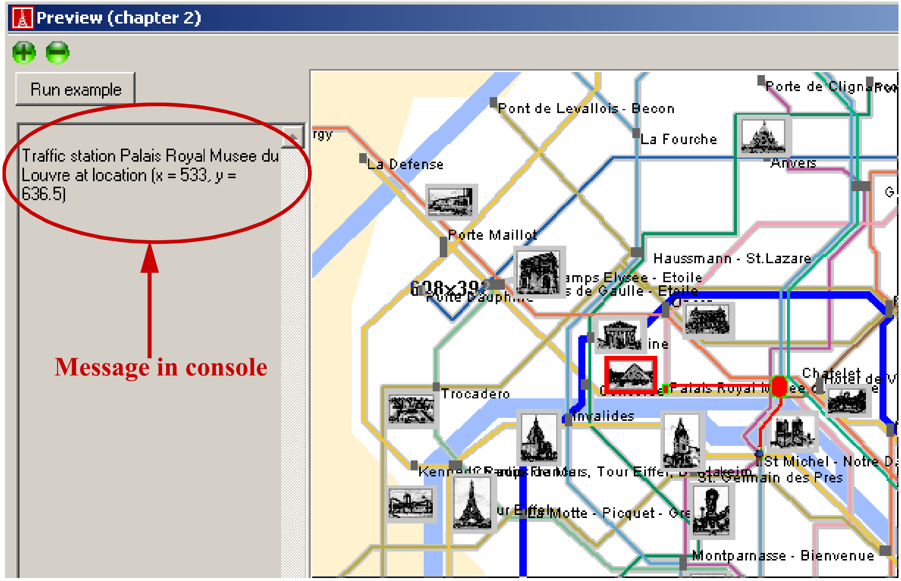
Displaying specific information

You will now modify the previous program to make it display new information.

There are only two changes, as highlighted below: an update to the comment — for explanation purposes only — and a new operation at the end:

```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Show city info, a route, and the route's origin.
        do
            Paris.display
            Louvre.spotlight
            Line8.highlight
            Route1.animate
            Console.show (Route1.origin)
        end
end
```

Execute the resulting program; the origin of the route, Louvre (formally: *Palais Royal Musée du Louvre*), shows up in the console window:



Traffic station Palais Royal Musee du Louvre at location (x = 533, y = 636.5)

This is the effect of our new feature call, *Console*.*show* (*Route1*.*origin*). Previous feature calls were all of the form *some_object*.*some_feature*, but the form of this one is new:

some_object.*some_feature* (*some_argument*)

where *some_argument* is a value that we pass to the feature because it needs that value to do its job. Feature *show*, for example, needs to know what to “show”, so we give it the corresponding value.

Such a value is known as an **argument** to the feature; the concept is the same as for arguments to functions in mathematics, where *cos* (*x*) denotes the cosine of *x* — the function *cos* applied to the argument *x*.

Some features will have more than one argument (separated by commas), although in well-designed software the majority of features typically have zero or one argument.

The notion of argument completes our panoply of basic program elements, which serves as the basis for all the discussions in subsequent chapters: classes, features, commands, queries, objects and arguments.

2.5 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A software system is a set of mechanisms to create, access and change collections of information called *objects*.
- An object is a machine controlling a certain collection of data, providing the program, at run time, with a set of operations, called *features*, applicable to this data.
- Features are of two kinds: *queries*, which return information about an object; and *commands*, which can change the object. Applying a command to an object may change the results of applying queries to that object later on.
- Some objects are software models of things from the physical world, like a building; others are software models of concepts from the physical world, like a travel route; yet others collect information meaningful to the software only.
- The basic operations performed by programs are *feature calls*, each of which applies a certain feature to a certain target object.
- A feature may have *arguments*, representing information it needs.

New vocabulary

Argument	**Class**	**Command**
Declaration	**Feature**	**Feature call**
Indentation	**Object**	**Query**

"Class" awaits a more complete definition in the chapter on interfaces.

Chapter 4.

2-E EXERCISES

2-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

The definition of "class" may be less precise than the others as we have not yet seen the concept in full.

2-E.2 Concept map

This exercise will run through subsequent chapters where you will be invited to extend the results first developed here. The goal is to create a conceptual map of the terms defined in vocabulary lists. To this effect, put the terms of the above list in boxes on a piece of paper, distributed over the page. Then connect them with arrows labeled by relation names to reflect the relation between the corresponding notions.

If you prefer an electronic medium, use a computer diagramming tool, but a single page of plain paper suffices for the terms of this chapter.

You may use a variety of relation names, but in any case consider the following fundamental ones:

- "Is a more specific kind of". For example, in a domain of knowledge having nothing to do with programming: *journal* "is a more specific kind of" *publication*; so is *book*.
- "Is an instance of". For example: *Australia* "is an instance of" *country*. Do not confuse with the previous relation: Australia is not a kind of country, it is a country.
- "Relies on" (as applied to concepts). For example, in mathematics, *division* "relies on" *multiplication*, since you cannot really understand division without first understanding multiplication. If either of the previous two relations holds between two terms, "relies on" also holds; so you should reserve it for cases in which the dependency is weaker (the second concept is neither a more specific kind nor an instance of the first).
- "Contains" (applied to instances of two concepts). For example, every *country* "contains" a *city*. You may instead consider the inverse relation, "is a part of".

You may add other kinds of relations as long as you provide a precise definition for each.

2-E.3 Commands and queries

In software for creating, modifying and accessing documents, assume a class *WORD* that describes a notion of word, and a class *PARAGRAPH* that describes a notion of paragraph. For each of the following possible features of class *PARAGRAPH*, state whether it should be a command or a query:

1 A feature *word_count*, used under the form *my_paragraph*.*word_count*, which gives the number of words in a paragraph.

2 A feature *remove_last_word*, used as *my_paragraph*.*remove_last_word*, which removes the last word of a paragraph.

3 A feature *justify*, used as *my_paragraph*.*justify*, which "justifies" a paragraph (makes sure it is aligned to both the left and right margins, like the present paragraph and most others in this book, but not the margin notes such as the one adjacent to exercise 2-E.1).

4 A feature *extend*, used as *my_paragraph*.*extend* (*my_word*), which takes an argument representing a word and adds it at the end of the paragraph.

5 A feature *word_length*, used as *my_paragraph*.*word_length* (*i*), which takes an integer argument representing the index of a word in a paragraph (*i* = 1 for the first word, *i* = 2 for the second word etc.) and gives the number of characters in the corresponding word (the word of index *i*) in the paragraph.

2-E.4 Designing an interface

Assume that you are building the software for an MP3 player.

1 List the main classes that you would use.

2 For each such class, list applicable features, indicating for each whether it is a command or a query and what arguments, if any, it should have.

3

Program structure basics

The previous chapter brought us our first brush with programs. We are ready to move on to new concepts of software design; to make this experience more productive, we pause briefly to take a closer look at some of the program parts we have been using, so far without having names for them.

3.1 INSTRUCTIONS AND EXPRESSIONS

The basic operations that we instruct our computer to execute, like the five we had in the latest version

```
Paris.display
Louvre.spotlight
Line8.highlight
Route1.animate
Console.show (Route1.origin)
```

are, naturally enough, called **instructions**. It is customary to write just one instruction per line, as here, for program readability.

All of the instructions seen so far are feature calls. In subsequent chapters we will encounter other kinds.

To do its work, an instruction will need some values, in the same way that the mathematical function "cosine", as in *cos* (*x*), can only give you a result if it knows the value of *x*. For a feature call the needed values are:

- The *target*, an object, expressed as *Paris*, *Louvre* etc.

- The *arguments*, if any, such as *Route1.origin* in the last example.

Such program elements denoting values are called **expressions**. Apart from the forms illustrated here we will also encounter expressions of the standard mathematical forms, such as *a + b*.

Definitions: Instruction, Expression

In program texts:

- An **instruction** denotes a basic operation to be performed during the program's execution.
- An **expression** denotes a value used by an instruction for its execution.

3.2 SYNTAX AND SEMANTICS

In the above definitions of "instruction" and "expression" the word "denotes" is important. An expression such as *Route1*.*origin* or *a + b* is not a value: it is a sequence of words in the program text. It *denotes* a value that may exist during the program's execution.

Similarly, an instruction such as *Paris*.*display* is a certain sequence of words, combined according to certain structural rules; it *denotes* a certain operation that will happen during execution.

This term *denotes* reflects the distinction between two complementary aspects of programs:

- The way you write a program, with certain words themselves made of certain characters typed on a keyboard: for example the instruction *Paris*.*display* consists of three parts, a word made of five characters *P*, *a*, *r*, *i*, *s*, then a period, then a word made of seven characters.
- The effect you expect the elements of these programs to have at execution: the feature call *Paris*.*display* will display a map on the screen.

The first kind of property characterizes the *syntax* of programs, the second their *semantics*. Here are the precise definitions:

Definitions: Syntax, Semantics

The **syntax** of a program is the structure and form of its text.

The **semantics** of a program is the set of properties of its potential executions.

It is fine to use "semantics" as a singular, like other similar words: "*Economics was a big part of the minister's speech, but while the politics was obvious the semantics was tortuous*".

Since we write programs to execute them and obtain certain effects, it is the semantics that counts in the end; but without syntax there would be no program texts, hence no program execution and no semantics. So we must devote our attention to both aspects.

Earlier on, we had another distinction: *commands* versus *queries*. Commands are *prescriptive*: they instruct the computer, when executing the program, to perform some actions, which may change objects. Queries are *descriptive*: they tell the computer to give the program some information about its objects, without changing these objects. Combining this distinction with the syntax-semantics division yields four cases:

→ *A synonym for "prescriptive" is "imperative". For more on this matter, see "Functional programming and functional languages", page 324.*

	Syntax	**Semantics**
Prescriptive	Instruction	Command
Descriptive	Expression	Query Value

In the bottom-right entry we have two semantic concepts: a *query* is a program mechanism to obtain some information; that information itself, obtained by the program by executing queries, is made of *values*.

3.3 PROGRAMMING LANGUAGES, NATURAL LANGUAGES

The notation that defines the syntax and semantics of programs is a **programming language**. Many programming languages exist, serving different purposes; the one we use in this book is Eiffel.

→ *For a brief review of major programming languages see "programming language styles", 12.1, page 322.*

Programming languages are artificial notations. Calling them "languages" suggests a comparison with the *natural* languages, like English, that we use for ordinary communication. Programming languages do share some characteristics with their natural cousins:

- The overall organization of a text as a sequence of *words* and *symbols*: a period "**.**" is a symbol in both Eiffel and English; *PREVIEW* in Eiffel and "*the*" in English are words.
- The distinction between *syntax*, defining the structure of texts, and *semantics*, defining their meaning.
- The availability both of words with a predefined meaning, such as "*the*" in English and **do** in Eiffel, and of ways to define your own words — as Lewis Carroll did in *Alice in Wonderland*: "*Twas brillig, and the slithy toves…*", and also as we just did by calling our first class *PREVIEW*, a name that means nothing special in Eiffel.

Word creation is far more common and open-ended with programming languages than with human ones. In natural languages you do not invent new words all the time, unless you are a poet, a little child or an Amazonian botanist. In programming, people who have never seen a flower, even less one from South America, and outwardly appear adult, might on a good day make up several dozen new names.

- Eiffel reinforces the human language flavor by drawing its keywords from English; every keyword of Eiffel is in fact a single and commonly used English word.

  Some other programming languages use abbreviations, such as *int* for *INTEGER*, but we prefer full words for clarity.

There is one exception: the keyword **elseif** *combines* two *words. See page 179.*

- It is also recommended that, whenever possible, you use words from English or your own language for the names you define, as we did in the examples so far: *PREVIEW*, *display*, or *Route1* (with a digit).

All these similarities between programming languages and human languages are good, because they help people understand programs. But they should not fool us: programming languages are different from human languages. They are *artificial* notations, designed for a specific purpose. This is both a loss and a gain:

- The power of expression of a programming language is ridiculously poor compared to the realm of possibilities available in any human language, even to a four-year old child. Programming languages cannot express feelings or even thoughts: they define objects to be represented on a computer and tasks to be performed on these objects.
- What they miss in expressiveness, programming languages make up for in *precision*. Human texts are notorious for their ambiguity and their openness to many interpretations, which are even part of their charm; when telling computers what to do, we cannot afford approximation. The syntax and semantics of a programming language must indeed be defined very precisely.

Touch of Style: Putting some English into your programs

For "English", substitute your own natural language if different.

Natural language has a place in programs: in *comments*. We saw that any program text element that starts with two dashes "--" is, up to the end of the line, a comment. Unlike the rest of program texts, comments do not follow any precise rules of syntax, but that is because they have no effect on execution — no semantics. They are just explanations, helping people understand your programs.

In addition, natural language serves as the basis for identifiers, in particular class and feature names. The methodological advice is to use full, clear names, such as *METRO_LINE* for a class (rather than abbreviations, except if they are already accepted in natural language).

In the end, to call our notations "languages" is to confer on them an honor they do not entirely deserve. Rather than scaled-down versions of the languages that people use to address each other, they are scaled-up versions of the *mathematical notations* that scientists and engineers use to express formulas.

> The term **code**, meaning "text in some programming language", reflects this. It is used in the expression "Line of code", as in "*Windows Vista contains over 50 million lines of code*". It is also used as a verb: "to code" means to program, often with an emphasis on the lower-level aspects rather than the design effort, as in "*they think all the ideas are there and all there remains to do is coding*". "Coder" is a somewhat derogatory term for "programmer".

Still, programming languages have a beauty of their own, which I hope you will learn to appreciate. When you start thinking of your love life as *relationship.is_durable*, or sending your mother an SMS that reads *Me.account.wire* (*month.allowance* + (*month*+1)*.allowance* + 1500, *Immediately*), it will be a sign that either or both: (1) the concepts are starting to seep in; (2) it is time to put this book aside and take the week-end off.

3.4 GRAMMAR, CONSTRUCTS AND SPECIMENS

To describe the syntax of a human language — meaning, as we have just seen, the structure of the corresponding texts — a linguist will propose a *grammar* for that language. For the simple English sentence

Isabelle calls friends

a typical grammar would tell us that this is a case (we say a *specimen*) of a certain "construct", maybe called "Simple_verbal_sentence" in the grammar, with three components, each a specimen of some construct:

- The subject of the action: *Isabelle*, a specimen of the construct "Noun“.
- The action described by the sentence: *calls*, a specimen of "Verb".
- The object of the action: *friends*, another specimen of Noun.

Exactly the same concepts will apply to the syntax description of programming languages. For example:

- A *construct* of the Eiffel grammar is Class, describing all the class texts that anyone can possibly write.
- A particular class text, such as the text of class *PREVIEW* or class *TOURISM*, is a *specimen* of the construct Class.

A future chapter discusses in detail how to describe syntax. For the moment we only need the basic definitions:

→ *Chapter 11.*

Definitions: Grammar, Construct, Specimen

A **grammar** for a programming language is a description of its syntax.

A **construct** is an element of a grammar describing a certain category of possible syntax elements in the corresponding language.

A **specimen** of a construct is a syntax element.

→ *"Grammar" will have a more detailed definition on page 296. A justification for using the term "specimen" appears on page 300.*

Be sure to note the relationship between constructs and specimens. A construct is a type of syntax element; a specimen is an instance — a specific example — of that type. So:

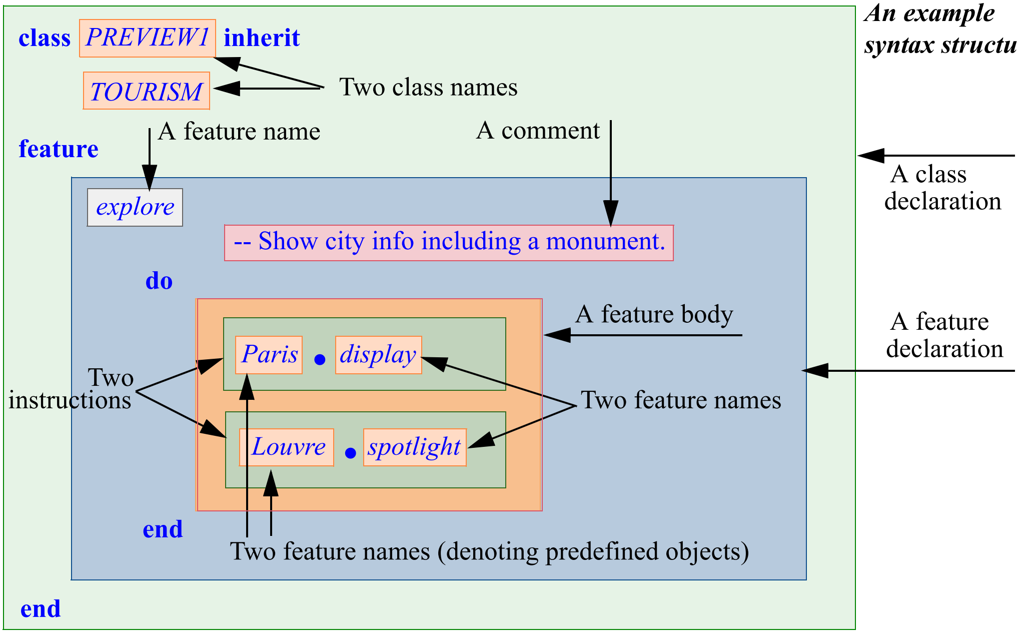
- In a grammar for English, we may have the constructs Noun and Verb; then *Isabelle* is a specimen of Noun and *calls* is a specimen of Verb.
- The standard grammar of Eiffel has the constructs Class and Feature; a particular class text is a specimen of Class, and any particular feature text is a specimen of Feature.

As these examples illustrate, construct names will always appear in This_green, with an upper-case first letter. They are not program elements, but ways to *describe* certain categories of program elements, for example classes and features. Specimens are program elements, and so will appear, like all program text, in *this_blue*.

3.5 NESTING AND THE SYNTAX STRUCTURE

The syntax structure of a software text can involve several levels of specimens (syntax elements). A class is a specimen; so is an instruction, or a feature name like *display*.

Interesting languages support embedding specimens within other specimens; the technical term is **nesting**. For example a class may contain features, features may contain instructions, and instructions may contain expressions and other instructions. Here is the nesting structure of specimens in our example class (retaining only two instructions for simplicity, and with a new name *PREVIEW1* to distinguish it from the full version):



An example syntax structure

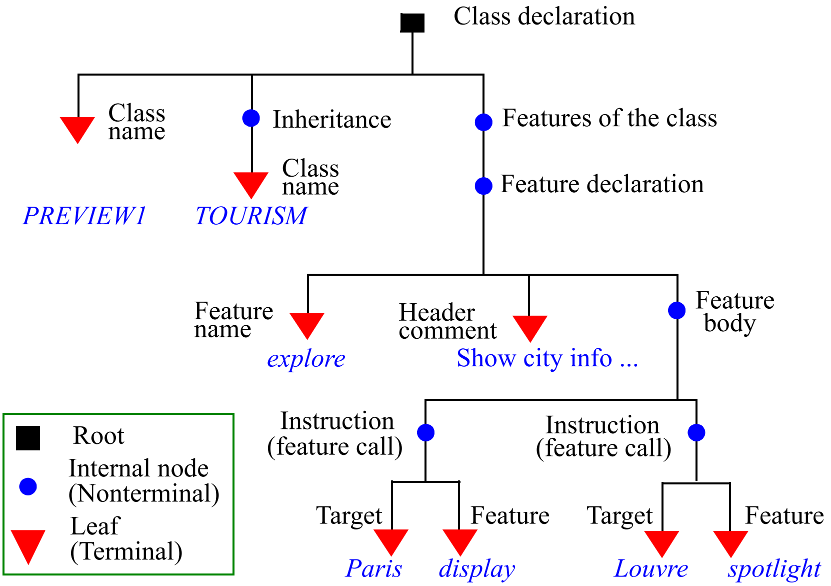
The embedding of the colored rectangles highlights the nesting of the specimens: the outermost rectangle covers the class declaration; the class declaration contains, among other specimens, a feature declaration; the feature declaration contains a "feature body" (the part that appears between the keywords **do** and **end**); the feature body contains two instructions; and so on.

Some elements of the syntax — keywords like **class**, **do**, **end**, and the period in feature calls — serve purely as delimiters and do not carry any semantic value of their own. We do not consider them specimens.

Make sure you understand the syntactic structure as illustrated above.

3.6 ABSTRACT SYNTAX TREES

For larger program texts, another representation of such a structure is more convenient. It relies on the notion of **tree**, as used for example to represent the organizational chart of a company — and inspired from nature's own trees with their branches and leaves, although *our* trees tend to grow top-down or left-to-right. A tree has a "root" which branches out to other "nodes" that may branch further. Trees serve to represent hierarchical structures as here:



An abstract syntax tree

This is known as an **abstract syntax tree**; it is "abstract" because it does not include the elements playing a delimiting role only, like the keywords **do** and **end**. We could also draw a "*concrete* syntax tree" that retains them.

Often abbreviated as "AST".

→ Chapter 11 discusses concrete syntax.

A tree includes nodes and branches. Each branch connects a node to another. From a given node, they may be any number of outgoing branches; but at most one branch may lead into the node. A node with no incoming branch is a **root**; a node with no outgoing branch is a **leaf**; a node that is neither a root nor a leaf is an **internal node**.

A non-empty tree has exactly one root. (A structure made of zero, one or more disjoint trees, having any number of roots, is called a *forest*.)

Trees are important structures of computer science and you will encounter them in many contexts. Here we are looking at a tree representing the syntax structure of a program element, a class. It represents the nesting of specimens, with the three kinds of node:

→ See in particular "Binary trees", 14.4, page 447..

- The root represents the overall structure — the outermost rectangle in the earlier figure.
- Internal nodes represent substructures that contain further specimens; for example a feature call contains a target and a feature name. In the earlier figure, these were the internal rectangles containing other rectangles.
- Leaves represent specimens with no further nesting, such as the name of a feature or class.

← Page 41.

For an abstract syntax tree, the leaves are also called **terminals**; a root or internal node is called a **nonterminal**.

Every specimen is of a specific kind: the topmost node represents a class; others represent a class name, an "inheritance" clause, a set of feature declarations etc. Each such kind of specimen is a **construct**. The above syntax tree shows, for each node, the corresponding construct name. Depending on the specimens it represents, a construct is either a "terminal construct" or a "nonterminal construct": the figure shows "Feature declaration" as a nonterminal and "Feature name" as a terminal.

A construct defines a general syntax notion, for example the notion of class text; a particular instance of that notion — for example the text of a particular class, such as the text of class *PREVIEW1* as given — is a specimen of that construct. As another example, the particular feature call *Paris***.***display* is a specimen of the construct "feature call".

The syntax of a programming language is defined by a set of constructs and the structure of these constructs.

3.7 TOKENS AND THE LEXICAL STRUCTURE

The basic constituents of the syntax structure include terminals, keywords, and special symbols such as the period "**.**" of feature calls. These basic elements are called **tokens**.

> Tokens are similar to the words and symbols of ordinary languages. For example the sentence in the margin has nine words ("This", "is" etc.) and three symbols (two hyphens and the final period)

This is a nine-word and three-symbol sentence.

Token categories

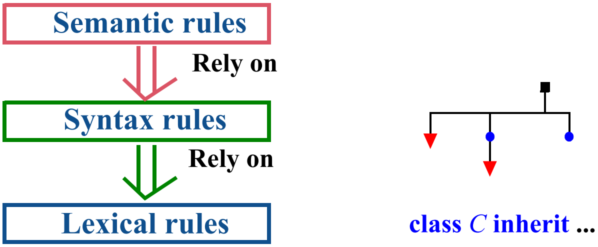
Tokens are of two kinds:

- **Terminals** correspond, as we have seen, to leaves of the abstract syntax tree; each carries some semantic information. They include names such as *Paris* or *display*, called **identifiers**, and chosen by each programmer to represent semantic elements such as objects (*Paris*) and features (*display*). Other examples are **operators** such as + and <= which will appear in expressions such as $a + b$, and **manifest constants** denoting self-explanatory values, such as the integer 34.
- **Delimiters** serve a purely syntactic role and do not carry any semantics. They include the 65 or so **keywords** of the language, such as **class**, **inherit**, **feature**, and **special symbols** such as the period "**.**" of feature calls and the colon "**:**". They do not appear in abstract syntax trees (but would figure, as leaves, in a concrete syntax tree).

Levels of language description

The form of tokens defines the **lexical** structure of the language. The syntax level comes above the lexical level, and semantics above syntax:

- Lexical rules define how to make up tokens out of characters.
- Syntax rules define how to make up specimens out of tokens satisfying the lexical rules.
- Semantic rules define the effect of programs satisfying the syntax rules.



Levels of language description

An important property of this hierarchy is that properties at any level are only defined if the constraints on earlier levels hold: syntax is only defined for lexically legal texts, and semantics for syntactically legal texts.

We will encounter an extra level between syntax and semantics: *validity*, defining non-syntax rules such as type constraints.

→ *"Definitions: Validity, correctness", page 368.*

Identifiers

For the moment we need only one lexical rule, governing identifiers:

Syntax: Identifiers

An identifier starts with a letter, followed by zero or more characters, each of which may be:

- A letter.
- A digit (0 to 9).
- An underscore character "_".

Route1 was an example of an identifier including a digit.

You may define your own identifiers as you please based on this rule, except that you may not pick a keyword since it is already reserved for a specific purpose. (Of course you only know a few keywords so far. But if you mistakenly reuse a keyword that you did not know existed you will get a clear error message.)

Touch of Style: Choosing your identifiers

For program readability, always choose identifiers that clearly identify the intended role; except in special cases (which we will see), use full names, not abbreviations: *Route1*, not *R1* or *Rte1*.

There is no tax on keystrokes; the few seconds that you might save by omitting a few letters will be more than offset by the time that you or someone else trying to understand your program will waste, later on, figuring out what you meant.

For identifiers denoting complex notions, use underscores to separate successive words, as in *My_route* or *bus_station*. This also works for class names, always in upper case: *PUBLIC_TRANSPORT*. Don't overdo it: for most identifiers, a single word, or two words separated by an underscore, are enough. Clear does not mean verbose.

In some programs, although usually not Eiffel ones, you may encounter for multi-word identifiers the use of an upper-case letter in the middle of a name: *myRoute*, *PublicTransport*. This convention is called "camel case" because of the humps. Better stay away from it, asItIsFarLessReadableThan underscores, which retain_perfect_clarity_even_with_very_long_identifiers.

Breaks and indentation

The lexical structure consists of successive tokens. To separate adjacent tokens you may use a **break**, which is a sequence of one or more of the following:

- Space.
- *Tab* character (which shows up as a sequence of spaces to reach aligned positions, but internally is just one character).
- Return to the next line.

Breaks only serve to separate tokens. It makes no difference to the syntax and semantics whether you go to the next line or use one space, one or more tabs (typically at the start of a line, for indentation), or several spaces (seldom useful), or any combination of these. Such a flexible structure, known as "*free format*", lets you devise the text layout that best reflects the program's structure — especially by highlighting its syntax nesting — to help readability, as in this book's examples.

Your program is stored in a *file*, which contains a sequence of characters such as letters, digits, tabs and special symbols. In such files a return to the next line is — in most file formats used today — simply represented by a particular character, known as "*New Line*". You will also encounter mentions of the "*Carriage Return*" character. The "carriages" in question are not the horse-and-buggy kind, although today they seems almost as old — a delightful and nostalgic reminder of the time when we typed our programs on typewriters. The print head was lodged in a little mechanical "carriage", which at the end of a line we would "return" to the leftmost position to start typing the next line.

On the Windows operating system, a line return is actually encoded by two characters, a Carriage Return followed by a New Line.

A break is usually not required between an identifier and a symbol: you may write $a+b$ without spaces, since this is not ambiguous. The style rules suggest including the break anyway for clarity: $a + b$.

3.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Programs are expressed in a *programming language*.
- A program has a *lexical structure*, defining the form of a program's basic elements (*tokens* separated by *breaks* such as spaces, tabs and line returns); a *syntax structure*, defining its hierarchical decomposition into elements (specimens) built out of tokens; and a *semantics* defining the execution-time effect of each specimen and of the whole program.
- The syntax structure usually involves nesting and may be described as a tree, known as an *abstract syntax tree*.

3-E.9 New vocabulary

Abstract syntax tree (AST)
Break
Carriage return
Code
Construct
Delimiter
Expression
Free format language
Grammar
Identifier
Instruction
Internal node
Leaf
Lexical
Natural language
Nesting
New line
Node
Nonterminal
Operator
Root
Semantics
Special symbol
Specimen
Syntax
Terminal
Token
Tree
Value

3-E EXERCISES

3-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

3-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapter.

← *Exercise "Concept map", 2-E.2, page 32.*

3-E.3 Syntax and semantics

For each of the following statements, say whether it characterizes syntax, semantics, both, or neither (explain):

1 "In a feature call, you must separate the target object from the feature name by a period."

2 "In a feature call *x*.*f*, there is no need to put spaces before or after the period, although they wouldn't hurt."

3 "Every feature call applies a feature to a certain object, the 'target' of the call."

4 "If there is an argument, it must be in parentheses."

5 "Separate two or more arguments of a given type by commas."

6 "Instructions separated by a semicolon will be executed one after the other".

7 "Eiffel and Smalltalk are object-oriented programming languages".

4

The interface of a class

In the previous chapters we have started to build some software relying on existing elements. We are going to do more of this now by seeing how we can use previously written *classes*. This will also be an opportunity to gain new insights into this notion of class, fundamental to everything we do in programming, and to discover the concepts of *interface* and *contract*.

Per the convention defined earlier, the system containing the examples of this chapter is in the subdirectory *04_interface* of the Traffic *examples* directory. ← *Page 15*.

4.1 INTERFACES

Many of the key decisions about building and using software systems involve the notion of interface. We may define it in relation to the notions, of client and supplier, also useful on their own:

> ***Definitions*: Client, Supplier, Interface**
>
> A **client** of a software mechanism is a system of any kind — such as a software element, a non-software system, or a human user — that uses it. For its clients, the mechanism is a **supplier**.
>
> An **interface** of a set of software mechanisms is the description of techniques enabling clients to use these mechanisms.

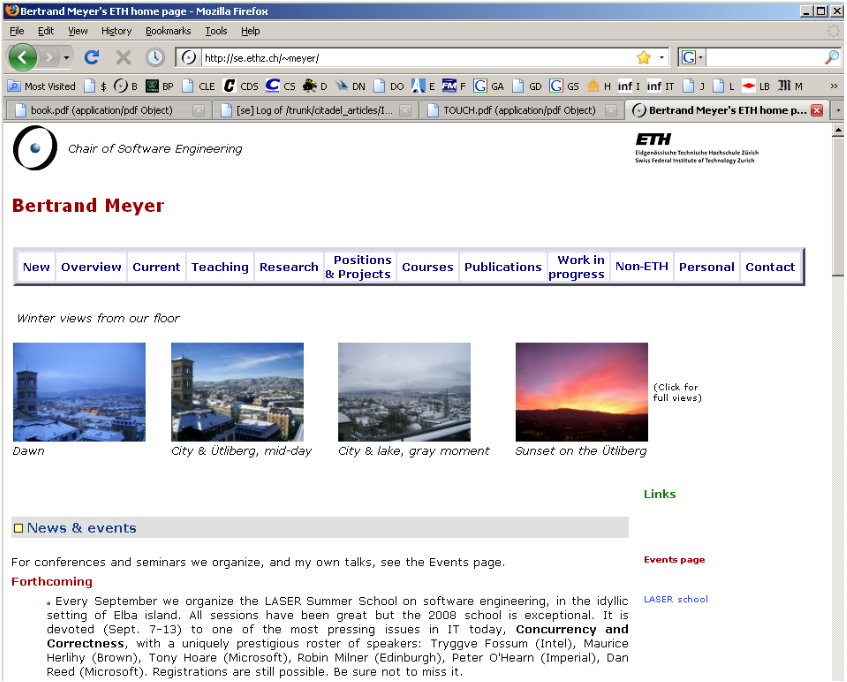
Informally, then, an interface for a piece of software is the description of how the rest of the world may "talk to" the software.

The definition speaks of "an interface", not "*the* interface". There are indeed several kinds of interface, and a software element may offer more than one interface, of the same or different kinds. The two principal kinds are:

- A **user interface**, whose intended clients are people using a software system.
- A **program interface**, for clients that are themselves software elements.

As an example of a *user interface*, consider a Web browser as shown (top part only) below. Its user interface is the description of what people can do with the browser; it includes:

- The specifications of the fields into which users may type their own texts, such as the address field at the top.
- The properties of buttons ("Back" etc.) that users may click to obtain certain effects.
- The conventions for hyperlinks (left-click will lead to a new page, right-click brings up a menu etc.).
- More generally the set of rules that govern the interaction between the browser and its users.


Bertrand Meyer's ETH home page - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://se.ethz.ch/~meyer/
Most Visited
$
B
BP
CLE
CDS
CS
D
DN
DO
E
F
GA
GD
GS
H
I
IT
J
L
LB
M
book.pdf (application/pdf Object)
[se] Log of /trunk/citadel_articles/I...
TOUCH.pdf (application/pdf Object)
Bertrand Meyer's ETH home p...
Chair of Software Engineering
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Bertrand Meyer
New
Overview
Current
Teaching
Research
Positions & Projects
Courses
Publications
Work in progress
Non-ETH
Personal
Contact
Winter views from our floor
(Click for full views)
Dawn
City & Ütliberg, mid-day
City & lake, gray moment
Sunset on the Ütliberg
Links
News & events
Events page
LASER school

For conferences and seminars we organize, and my own talks, see the Events page.

Forthcoming

- Every September we organize the LASER Summer School on software engineering, in the idyllic setting of Elba island. All sessions have been great but the 2008 school is exceptional. It is devoted (Sept. 7-13) to one of the most pressing issues in IT today, **Concurrency and Correctness**, with a uniquely prestigious roster of speakers: Tryggve Fossum (Intel), Maurice Herlihy (Brown), Tony Hoare (Microsoft), Robin Milner (Edinburgh), Peter O'Hearn (Imperial), Dan Reed (Microsoft). Registrations are still possible. Be sure not to miss it.

A user interface

Such a user interface is *graphical*, meaning that it involves pictures and other visual dialog elements such as buttons and menus. The computing profession, with its crush on acronyms, calls this a **GUI**, for Graphical User Interface, pronounced "*Gooey*" (or, increasingly, just **UI** — "*You-I*" — as we take the graphical aspect for granted).

Other user interfaces involve no graphics but only text, as on older mobile phones; they are called "text interfaces" or "command-line interfaces".

If in dealings with computer systems you have encountered less-than-perfectly-friendly user interfaces, you will probably agree that GUI design is an important part of software design. More fundamental for the present discussion is the second kind of interface cited, *program interfaces*. Here too you have to get used to a three-letter acronym: **API**, for "Abstract Program Interface" (an older expansion of the A is "*Application*").

Three-Letter Acronyms are affectionately called "TLAs".

In the rest of this chapter we will learn how APIs look for a particularly important kind of software element: the class. Since for the moment we are not concerned any more with user interfaces, we will indifferently say "API", "program interface" or even just "interface" to mean the same thing.

4.2 CLASSES

A previous discussion defined an object as "a software machine allowing programs to access and modify a collection of data". Such collections of data might represent, to stick to the examples we have seen:

← Page 29.

- A city, where the "access and modify" operations may include finding out about current traffic conditions and adding some vehicles to the traffic. We have used *Paris* as an example, although we may obtain objects representing any other city provided we have the relevant data.
- A travel route. Again we may have many such routes, not just *Route1* as used in the original example.
- A list of cars waiting at a red light. Many possible objects again.
- Closer to the computer, an element of the GUI such as a button or a window on the screen. Of these too we will have many.

Within any such category of objects, for example all city objects, a strong similarity exists: the operations applicable to a city object such as *Paris* would also apply to other city objects, say *New_York* or *Tokyo*. They do not apply to objects of another category, such as the route object *Route1*; but operations applicable to *Route1*, such as adding a new leg to a route, would also apply to other routes.

What this tells us is that the objects manipulated by our programs classify themselves naturally into certain **classes**: the class of objects representing cities, the class of objects representing travel routes, the class of objects representing buttons on the screen, and so on.

"Class" is indeed the technical term. What characterizes objects of a given class is a common set of applicable operations — or *features* in the terminology introduced in the discussion of objects. Hence the definition:

Definition: Class

A class is the description of a set of possible run-time objects to which the same features are applicable.

In program texts, classes will stand out by always having names all in upper case, such as *CITY*, *ROUTE*, *CAR_LIST*, *WINDOW*. The names representing objects are in lower case, or, for predefined objects such as *Paris*, with only the first letter in upper case.

A class represents a category of things; an object represents one of these things. The following terms express precisely this relationship between classes and objects:

Definitions: Instance, Generating class

If an object *O* is one of the objects described by a class *C*, then *O* is an **instance** of *C*, and *C* is the **generating class** of *O*.

CITY is a class, representing all possible cities (as we have decided to model them in our program); *Paris* denotes an object, an instance of that class.

This relationship between classes and objects is the usual one between a category and members of that category: “Human” is a category, “Socrates” is one of its members. If these were software notions we would say there’s a class *HUMAN* and one of its instances is the object called *Socrates*.

In software the difference goes further. Classes are static and objects are dynamic:

← The terminology was introduced in “Definitions: Static, Dynamic”, page 11.

- Classes exist only in the software text. As the definition of “class” says, a class is a *description*; it will be given by a **class text**, a software element describing the properties of the associated objects (instances of the class). A program is a collection of class texts.
- Objects — “collections of data” — exist only during the software’s execution; you do not see them in the program text, although you will see there names such as *Paris* and *Route1* denoting objects that will appear during execution.

  As a consequence, the term “run-time object” appearing in the definition of “class” above includes a redundancy, since objects by definition can only exist during program execution (“run time”). From now on we will just say “object”.

Finding appropriate classes is a central part of the task of software *design*, devoted to organizing the essential structure, or *architecture*, of a program — as opposed to writing down the details, or *implementation*.

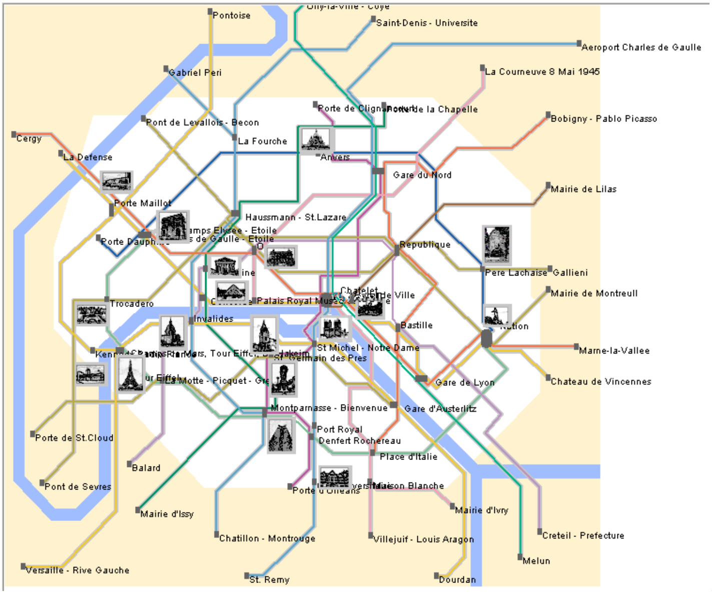
4.3 USING A CLASS

You are now going to learn what a class looks like and how you can use it to build new classes — *client* classes — for your own programs.

The classes that we will examine have been written to cover properties and operations relative to a transportation network like the Paris metro. We use Paris, as you know, because it is one of the world's top tourist destinations, but of course it is only an example; the software knows nothing about a particular location, it simply reads in a file describing a city and transportation network, so you can tailor it to any ones you choose. (At ETH we use Zurich and its tram network.) Below for reference is the metro plan (the simplified version taken from Traffic, omitting some stations).

The description in the file is in XML, a standard format for structured data.

Metro plan



Defining what makes a good class

Assume we are asked to devise a software model for the metro as seen by actual and prospective passengers. As in any software design problem, the key question will be: *What are the classes*? To find good classes answering this question, we search the problem domain for concepts that:

- Describe sets of objects (their future instances).
- Can be explained clearly.
- Can be characterized in terms of clearly defined **features**, including both *queries* and *commands*, applicable to the corresponding objects.

← "Features, commands and queries", page 26.

A mini-requirements document

Can we find classes and their features to build a software model of the metro? Often a first step to design is simply to express in clear, simple language what the problem domain is about. Let's try:

Touch of Paris: Welcome to the Metro

The Metro is a train network, mostly underground, enabling people to travel across the city quickly and conveniently.

The network is made of "lines"; each line connects a set of "stations", two of which are its "end stations". Trains on a line travel from one of the end stations to the other, stopping at each station along the way, and then back in the same manner.

Some stations belong to two or more lines; they are called "exchanges" and allow passengers to connect from one line to another.

To go to a certain destination using the Metro you should first identify the stations closest to where you are and to where you want to go, then construct a Metro route between them. The route is made of a number of legs, each consisting of successive stations on a single line; successive legs connect through exchange stations. It is a property of the Metro network that such a route always exists between any two stations (in mathematical terms, the graph is *connected*).

This description is far more formal, pedantic even, than what you would probably tell a visitor who has not used the metro before — but still far less precise and complete than what we expect to find in the "requirements document" of a software project in industry (a topic discussed in the chapter on software engineering). The text is good enough for our current purpose of discovering a few classes.

→ *"Requirements analysis", 19.6, page 718.*

Initial ideas for classes

As usual in requirements documents, some details are irrelevant for our immediate need, for example that the network runs "mostly underground". The word "network" itself is not that useful. But without much hesitation we can spot four concepts likely to yield classes:

- *STATION*. The metro is made of stations; people travel from a station to a station, going through other stations. This seems like an inevitable notion for our software.
- *LINE*: the metro consists of a set of lines, each connecting a number of stations that the line traverses in a set order.
- *ROUTE*: a description of how to go from a given station to another.
- *LEG*: a set of contiguous stations on a line.

Close relations exist between these notions: a line is made of stations; a leg, also made of stations, is part of a line; a route is made of legs from different lines.

← We have already extended the notion of leg to cover a change of lines within a station; see the figure "A metro route", page 25.

You indeed have at your disposal, in the Traffic software, a set of classes covering these notions, and we now take a look at some of their properties. More precisely, Traffic is a **library**: a collection of software elements that do not by themselves constitute a program, but provide important functionalities useful to many programs (each of which will add its own specific elements). Even though the Traffic library is available as part of the material for this book, we will work in this chapter as if *we* had to design the corresponding classes, starting from the basic concepts of line, leg, station and route. You are as usual welcome to look up the classes in the software. If you do so note the following convention:

Convention: Traffic library class names

For clarity, and to avoid clashes with other libraries, all classes in the Traffic library have names starting with the *TRAFFIC_* prefix, for example *TRAFFIC_STATION*, *TRAFFIC_LINE*, *TRAFFIC_ROUTE*, *TRAFFIC_LEG*. For brevity, the discussions in this book ignore the prefix, talking instead of classes such as *STATION* and *LINE*. You must use the prefix if looking for the library classes in EiffelStudio.

This only applies to library classes; names of example classes such as *TOURISM* and *PREVIEW* in the previous chapter do not use the prefix.

What characterizes a metro line

Let us start by understanding the interface (in the sense of program interface, or API) of a class representing metro lines. You can use the EiffelStudio environment to see the interface of any available class.

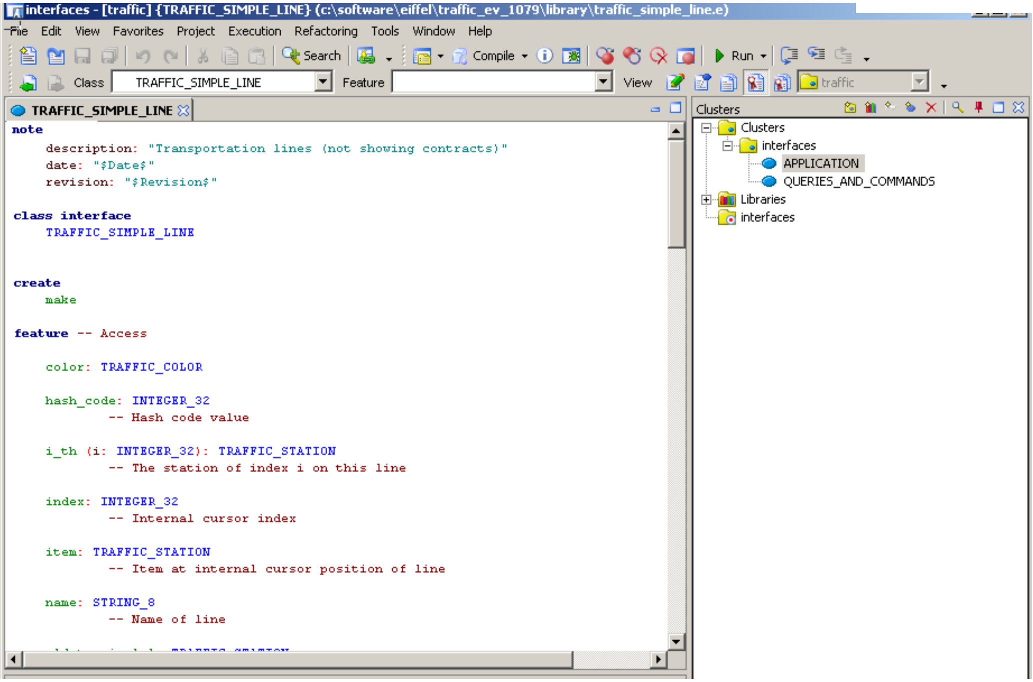
← "API" was defined on page 49.

We first look at a simplified form of class *LINE*, called *SIMPLE_LINE*. With EiffelStudio you can bring up the class text, but this is not what I would like you to do right now. Other than the class text, EiffelStudio can display various other *views* of a class, each highlighting different properties. The view you should bring up now is known — for reasons that will become clear as we go — as its **contract view**.

Actually TRAFFIC_SIMPLE_LINE per the convention above.

To get it, enter the class name in the corresponding field, then click the contract view button towards the right of the top row of buttons — you can find it easily by moving the cursor over the buttons and watching for the tooltip that reads "Contract view". (See the EiffelStudio appendix for more detailed instructions.) The result looks as follows (overleaf):

→ "Bringing up classes and views", E.3, page 845.


interfaces - [traffic] {TRAFFIC_SIMPLE_LINE} (c:\software\eiffel\traffic_ev_1079\library\traffic_simple_line.e)
File Edit View Favorites Project Execution Refactoring Tools Window Help
Search
Compile
Run
Class
TRAFFIC_SIMPLE_LINE
Feature
View
traffic
TRAFFIC_SIMPLE_LINE
Clusters
Clusters
interfaces
APPLICATION
QUERIES_AND_COMMANDS
Libraries
interfaces

```
note
    description: "Transportation lines (not showing contracts)"
    date: "$Date$"
    revision: "$Revision$"

class interface
    TRAFFIC_SIMPLE_LINE


create
    make

feature -- Access

    color: TRAFFIC_COLOR

    hash_code: INTEGER_32
            -- Hash code value

    i_th (i: INTEGER_32): TRAFFIC_STATION
            -- The station of index i on this line

    index: INTEGER_32
            -- Internal cursor index

    item: TRAFFIC_STATION
            -- Item at internal cursor position of line

    name: STRING_8
            -- Name of line
```

The contract view shows the features — commands and queries — applicable to an instance of a class *SIMPLE_LINE*, representing a line of the metro. The next sections study these features.

> Once you have read the discussion of these features, go back to the preceding picture (or display again the corresponding contract view on your computer) and make sure you understand all their details as given in the contract view.

To follow the discussion of queries and commands you need to remember that the class describes a set of possible objects, its instances (each representing a metro line). The features are declared in the class; each defines an operation applicable to any such object. For example the class will have a query *south_end* giving one of its two end stations (the one to the south of the other); this means that we may apply this query to any of its instance. If *Line8* denotes an instance of *SIMPLE_LINE*, then *Line8*.*south_end* denotes its end station. We commonly say things like “a *SIMPLE_LINE*” to talk about a typical instance of the class, representing a typical line.

← *“Definitions: Instance, Generating class”, page 50.*

SIMPLE_LINE represents a slightly simplified version of the final class *LINE*; the discussion applies to both classes. We look at the queries first, then the commands.

4.4 QUERIES

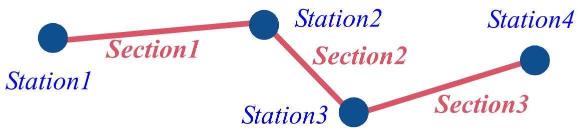
Class *SIMPLE_LINE* offers a number of queries about metro lines.

How long is this line?

One of the first things we may need to know about a line is the number of its stations. This is provided by a query *count* (taken, as an English word, to refer not to the verb as in "Count up to 10!", but to the noun as in "head count"). The specification of that feature appears in the contract view as

```
count: INTEGER
        -- Number of stations in this line.
```

The second line is, as you know, a comment, more precisely a **header comment** that should come with every feature. It is always useful to give a plain language explanation of what a feature is about. Among other things this avoids misunderstandings. Here for example, we could have chosen, as measure of size, the number of segments (elementary legs, those from a station to the next) rather than stations; the result would always be one fewer, as illustrated on this little line with four stations and three segments:



Four stations, three segments

The comment clarifies our convention: for class *SIMPLE_LINE* the *count* of a line is the number of its stations.

In the comment, note the expression "in *this* line". The class describes the general notion of line, but when a feature like *count* is applied to a particular line, as in the feature call *Line8*.*count*, it will give us the station count of that line, *Line8* in this example. So in the end the class always talks about a particular line, even though in the class we do not know what it is. That's what "this line" means: whatever line object to which we will apply the feature *count*.

The query declaration starts with *count*: *INTEGER*. This introduces the name of the query, *count*, and the type of the result it will return, *INTEGER*. A query provides information on an object (here an instance of *SIMPLE_LINE*), so the class interface must say what type of information that is.

INTEGER is such a type, denoting integer values, zero: positive or negative. The names of types, like classes, will always be written all in upper case. Other types encountered later in this chapter include:

- *STRING*, for values that are sequences of characters, such as "*ABCDE*".
- *BOOLEAN*, for "truth values" that can only be either **True** or **False**.
- Classes themselves, such as *STATION* or *LEG*.

More on types soon. For the moment *INTEGER* suffices.

Experimenting with queries

As you encounter features in this chapter, you can try them out.

> ***Programming time!***
> **Length of a line**
>
> The first programming exercise of this chapter, detailed below, asks you to find the length of Line 8 of the metro.

A system called *interface* has been set up (in the subdirectory *04_interface*) as part of the Traffic software. Start EiffelStudio now on that system and bring up the text of its class *QUERIES_AND_COMMANDS* (see instructions in the EiffelStudio appendix):

The actual name (without TRAFFIC_) as it is just an example class, not part of the library.

```
class QUERIES_AND_COMMANDS inherit
    TOURISM
feature
        tryout
                -- Try out queries and commands on lines.
            do

            end
end
```

The part you will fill in

This class is just a playground for trying out the concepts of this chapter; to achieve this you can, as you go along, fill in with various feature calls the part highlighted above. You will be able to execute the resulting system and see the effects in each case.

The metro Line 8 is defined, in the context set up for you by class *TOURISM*, by a feature called *Line8*. Enter, into the "fill-in" part, the instruction

```
Console.show (Line8.count)
```

This calls the just described feature *count* on *Line8*, and then uses the command *show* on *Console* to display the result in the console window. Now you know how many stations Line 8 has.

> As in the chapter on objects some "magic" remains involved since you are relying on *Console* and on *Line8* (denoting an instance of *SIMPLE_LINE*), both prebuilt in *TOURISM*. There will be a little more such magic in this chapter; we need it to let you concentrate on the new concepts you are learning. Pretty soon the magic will go away and you will be able to define everything you need.

From the terminology of the previous chapter, you will remember that *Line8*.*count*, denoting the result of applying a query to an object, is an **expression**. Every expression has a type; here, because the query *count* has been declared to return an *INTEGER* result, the type of the expression is also *INTEGER*, as appropriate since it denotes a number of stations.

← *"Definitions: Instruction, Expression", page 36.*

The stations of a line

Our next queries tell us exactly what stations are on a line. Remember the explanation in our little requirements statement:

> Each line connects a set of "stations", two of which are its "end stations" ...

Although we need to complement this imperfect specification with our intuitive understanding of a transportation network, it clearly implies that a line contains a *sequence* of stations: one of the end stations; a station; another station; and so on up to the other end station. A simple way to represent this is the following query of class *SIMPLE_LINE*:

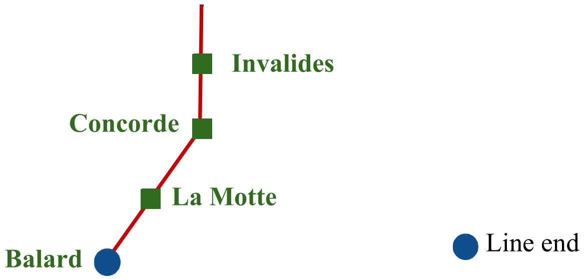
```
i_th (i: INTEGER): STATION
        -- The station of index i on this line.
```

> The name *i_th* comes from the common way of referring to an item by its position in a series: "the *i*-th element", as in "the 25-th element". We cannot call the query *i-th* because hyphens "-" are not permitted in identifiers, but underscores "_" are.

← *"Syntax: Identifiers", page 44.*

The query *i_th*, like *show* for *Console*, takes an argument, representing the number, or "index" of the stop we want, starting at 1 for the first end station, then 2 for the first stop after it and so on. So if we again take Line 8 as an example, and refer to this map extract:

← *"Features with arguments", 2.4, page 30.*



Start of line 8 (main stations only, some names shortened)

To know all about Line 8 of the Metro: *jefx.chez.com/lignes/ligne8.htm*.

we may use, in our program text, the expression

```
Line8. i_th (1)
```

representing the station called "**Balard**" on the above map); *Line8*.*i_th* (2) is "**La Motte**" and so on. *Line8* has been predefined for us as part of the "magic". To make our life easier we will follow two conventions:

Conventions: Line numbering

- A line always has at least one station, even if it is empty. (In that case it has zero segments.)
- The numbering of stations on a line always starts from the south end. Query *south_end* will denote that station, *north_end* the other end. For an empty line (and a circular one) they denote the same station. In the rare case that the two ends are different but at the same latitude, *south_end* denotes the one to the west of the other.

The reference to empty lines sounds strange if you think of the actual lines of the metro. But we are playing with the abstract notion of line, for which an empty line is possible. Later in this chapter we will build a (virtual) metro line, starting with an empty line and adding stations.

Class *SIMPLE_LINE* has the following two queries denoting the ends of a line:

```
south_end: STATION
        -- End station on south side.

north_end: STATION
        -- End station on north side.
```

← Query south_end was mentioned on page 54.

Quiz time: The other end

Line8.i_th (1) is an expression of type *STATION* denoting the station at the south end of Line 8. Without looking up the number of stations on that line or the names of individual stations (or the answer to this quiz, which appears below), write another expression that denotes in the same style the object representing the station at the *other* end of the line. *Hint*: use another query already introduced.

Properties of start and end lines

To express more precisely our decision to start numbering at the south end, we note that any line *l* will satisfy the following properties:

```
l.south_end = l. i_th (1)
l.north_end = l. i_th (l.count)
```

Do not even *think* of reading any further unless you understand these two program lines perfectly. Each states a property of *l*, an equality, similar to equalities you have seen in mathematics, such as $cos^2(x) + sin^2(x) = 1$ for any number x:

- The first equality says that the query *south_end* will always return the same result as the query *i_th* applied to the same metro line with the argument 1; in other words, it states our convention that station numbering on a line starts at the south end.
- The second equality gives the corresponding equality at the other end. Since *l.count* denotes the number of stations on the line, the expression *l.i_th* (*l.count*) denotes the last station.

The convention that a line always has a station, even if it is empty, is also essential here: otherwise *l. i_th* (1) would not always be meaningful.

This also gives us the answer to the little quiz above: to obtain the second end station of Line 8 you may use the expression *Line8.i_th* (*Line8.count*) — or the simpler one *Line8.north_end*.

4.5 COMMANDS

So far we have accessed properties of existing lines, using queries. It is time now to look at the other category of features: commands, which enable us to change an object.

Building a line

What can we do to a line to change it? The most obvious operation is to add a station to it, for example at one of its ends.

If you are thinking: "*This is nonsense*: *a program cannot create a metro station*, *and the Metro lines already exist anyway*!", you should probably read again the section that explained that our objects are **software** artifacts, not the real thing. We will need the ability to change lines, if only to set up our object structure at the beginning of an execution, once we get rid of the magic of class *TOURISM* which at the moment creates the structure for us.

← *"Objects you can and cannot kick", page 25.*

To set up the object structure ourselves we might get the information from some external description of the metro structure (in a file or database), then use it to create all the objects we need, such as stations and lines.

Let us indeed rebuild line 8. From class *TOUR* we may assume the following: predefined features such as *Station_Balard*, *Station_La_Motte* etc. are available for every station; the name of the feature for station "xxx" is *Station_Xxx*; multiple words, as with the station "La Motte" of Line 8, are separated by underscores in the identifier (here giving *Station_La_Motte*).

Of course *Line8* is itself predefined from *TOUR*, so the first thing we need to do is to empty it out of its stations. In the contract view of class *SIMPLE_LINE* you may note the following command, which will do the job:

```
remove_all_segments
          -- Remove all stations except the south end.
```

Our program will use it under the form

```
Line8.remove_all_segments
```

Remember that by convention our lines always have at least one station; when it is the only one, for example after a call to *remove_all_segments* (which, as indicated by the header comment, retains the south end), it will be the value of both *south_end* and *north_end*.

Now we are ready to add stations. Again you can see the relevant command in the contract view:

```
extend (s: STATION)
          -- Add s at end of line.
```

This means that if *li* denotes a line you may add a station *st* at its end through

```
li.extend (st)
```

Indeed, you may now start filling in the text of this chapter's example class:

← *From page 56.*

```
class QUERIES_AND_COMMANDS inherit
    TOURISM
feature
        tryout
                -- Recreate a partial version of Line 8.
            do
                Line8.remove_all_segments
                        -- No need to add Station_Balard, since
                        -- remove_all_segments retains the south end.
                Line8.extend (Station_La_Motte)
                Line8.extend (Station_Concorde)
                Line8.extend (Station_Invalides)
                    -- We stop adding stations, to display some results:
                Console.show (Line8.count)
                Console.show (Line8.north_end.name)
            end
end
```

Quiz time: The last name shown

As you may guess from the last instruction, class *STATION* (the type of *north_end*) has a query *name*, which gives the name of a station. What name should this last instruction display in the console window?

To check that your reasoning is correct, run this example now.

4.6 CONTRACTS

One of the reasons that the "line" class *SIMPLE_LINE* used so far is not the final *LINE* class is that it misses a fundamental property which we cannot ignore if we are to write serious software: that not all features are applicable to every possible argument and instance. Interfaces will need to be more precise about what is permitted.

Preconditions

The interface for the query *i_th* in class *SIMPLE_LINE*, as shown earlier

```
i_th (i: INTEGER): STATION
        -- The i-th station on this line.
```

does not mention that only certain values for *i* makes sense: the value must be between 1 and the number of stations on the line, *count*. If Line 8 has 20 stations then it would be wrong to use *Line8*.*i_th* (300), or *Line8*.*i_th* (0), or *Line8*.*i_th* (–1).

> You may try such an out-of-bounds value on the computer if you wish: edit the class, execute the program under EiffelStudio, and see what happens.

→ *Exercise "Violating a contract", 4-E.3, page 69.*

A programmer who is trying to understand what the class is about — a potential "*client programmer*" — needs this kind of information. This is precisely what interfaces are about: telling client programmers what a given class can do for them.

We could of course add the information to the header comment, as in

```
i_th (i: INTEGER): STATION
        -- The i-th station on this line
        -- (Warning: use only with i between 1 and count, inclusive.)
```

Warning: *not recommended style, see next.*

which is better than nothing, but not good enough. Such usage properties are so common, and so critical for the proper use of classes and their features, that they must be treated as an integral part of the program, at the same level as the instructions and expressions. They will be called **contracts**. For *i_th* we have our first contract element, a **precondition**. A precondition is a property that a feature imposes on all its clients; here, the property that the argument must be within a certain range.

The interface of a feature will show the contract using the keyword **require**. So the contract view of class *LINE* actually describes *i_th* in this way:

```
i_th (i: INTEGER): STATION
        -- The i-th station on this line
    require
        not_too_small: i >= 1
        not_too_big: i <= count
```

The precondition clause is made of two separate elements called **assertions**. Each expresses a property: $i \geq 1$ in the first assertion and $i \leq count$ in the second one. Note that because of the limitations of computer keyboards we cannot use the mathematician's symbols ≥ and ≤; programming languages let us instead use the 2-character symbols >= and <=. Also, the names not_too_small and not_too_big, called **assertion tags**, serve to clarify the purpose of the assertions, but the actual meaning (the semantics) is in the expressions that follow, *i* >= 1 and *i* <= *count*. We may omit the assertion tags and colons, as in

```
require
    i >= 1
    i <= count
```

without changing the meaning of the precondition, but tags make things clearer, so you should include them as a matter of good style. When present, the tags appear in roman to stand out from the program elements in *italics*.

Expressions like *i* >= 1 and *i* <= *count* denote *conditions* which, at any time during program execution, may be either true or false. Earlier examples involved equality, as *l*.*south_end* = *l*.*i_th* (1) for a line *l*. An expression that can take the values true and false — written in Eiffel as **True** and **False**, with a capital first letter since they are predefined values — is known as **boolean**:

← *Page 59.*

Definition: Boolean value

A boolean value is one of: **True** and **False**.

The corresponding type is called *BOOLEAN*; it is one of the types we have at our disposal, along with *INTEGER*, *STRING* and names of classes you define. Most other types have many values — we will see for example that typically the representation for an integer value on a computer supports some billions of possibilities — but *BOOLEAN* provides only two. The purpose is to represent conditions; as in non-programming uses of this concepts ("there is enough snow", as in a condition to decide whether you can go skiing) a condition is either true or false. As usual, our boolean expressions in software must be more precisely defined, like in mathematics: *i* >= 1 is unambiguously true or false once we know the value of the integer *i*, whereas how much snow is "enough snow" is in the eyes of the would-be skier.

← *The notion of type was introduced in "How long is this line?", page 55.*

Boolean values and boolean expressions lie at the heart of *logic*, the art of precise reasoning; the next chapter is devoted to this topic.

Preconditions and the other forms of contract will use boolean expressions to state conditions that clients and suppliers must satisfy. Here the precondition of *i_th*, as it appears in the interface

```
require
    not_too_small: i >= 1
    not_too_big: i <= count
```

is essential information for the client.

A client that does *not* satisfy that property, for example if it has a call

```
Line8.i_th (1000)
```

is faulty software, or *buggy* according to habitual terminology, where a "bug" is simply a mistake.

→ We will see a more precise terminology in "Varieties of quality assurance", page 728.

We may express this observation as a general principle:

Touch of Methodology: Precondition Principle

A client calling a feature must make sure that the precondition holds before the call.

Whenever you consider using a feature, you will see its specification in the contract view of the corresponding class, including its precondition if any, as in the example of *i_th* above. It is then your responsibility, as the client programmer, to make sure that any call to the feature satisfies the precondition.

Some features are always applicable; they do not have a **require** clause. By convention this means the same as if they had one of the form

```
require
    always_OK: True
```

defining a precondition that is always satisfied.

Contracts for debugging

One way that preconditions and other contracts will help you during software development is that the tools will check them when you execute your program. So if one of the contracts does not hold, revealing a bug, you will get a precise message telling you what happened; the message lists the tag (such as not_too_small) of the violated assertion, so that you know what exactly went wrong.

→ Try this with the exercise "Violating a contract", 4-E.3, page 69.

When a program is ready for distribution, you should have corrected all the bugs, and can change the options to stop checking contracts at run time.

A section of the EiffelStudio appendix tells you how to tune the environment parameters defining whether (and which) contracts should be monitored during execution.

→ "Contract monitoring", E.5, page 846.

Contracts for interface documentation

The better approach to software correctness is, of course, to avoid bugs in the first place (rather than make mistakes and then correct them); systematic use of contracts helps. In particular, the documentation of a software mechanism, as given by its interface, should always list the complete *precondition* that defines under what circumstances it is legitimate to use the mechanism.

This style — illustrated by the interface for *i_th* as shown above — will be the standard form of interface description for the rest of this book. It is also what you get in EiffelStudio's standard view for presenting the interface of the class — correspondingly called, as we have seen, the *contract view*.

Postconditions

In describing the interface that a feature presents to its potential clients, preconditions address only one side: what a feature expects from the clients before a call. For the clients, a precondition is an *obligation*. As in any good relationship, the clients will want to know what *benefits* they will get after a call. The feature's interface can express this through a **postcondition**.

Unlike with preconditions, we will not always be able in postconditions to express all relevant properties, but often we can say something interesting anyway. Here for example is the interface for *remove_all_segments* in class *LINE*:

```
remove_all_segments
        -- Remove all stations except south end.
    ensure
        only_one_left: count = 1
        both_ends_same: south_end = north_end
```

There is no precondition here since *remove_all_segments* is always applicable to a route. The keyword **ensure** introduces a postcondition. Here the feature guarantees two things to its client when it has done its job:

- The number of stations, *count*, is equal to 1.

- The two end stations, *south_end* and *north_end*, are now the same station. Remember that this is the convention we take: an empty line has no segments but one station, serving as both south and north ends.

← *"Conventions: Line numbering", page 58.*

Similarly, here is the interface (precondition omitted) for *extend*, the command that adds a station at the end of a line:

```
extend (s: STATION)
        -- Add s at end of line.
    ensure
        new_station_added: i_th (count) = s
        added_at_north: north_end = s
        one_more: count = old count + 1
```

The first postcondition clause uses the query *i_th*: it states that after a call to *extend*, if we ask what is the station at position *count*, that is to say the last station, the answer will be *s*, the station that we have asked the command *extend* to add. This expresses precisely the intent of *extend*: if the command does its job properly — that is, if its program text does not have any bugs — this property will always hold as a result of an execution of the command.

The second clause expresses that *north_end* will also be equal to *s*. From the invariant, to be seen in the next section, we will learn that *north_end* must be equal to *i_th* (*count*); so this clause is in fact redundant, but it does not hurt.

The third clause tells us that the routine increases *count* by one. It uses a keyword that we have not yet encountered, **old**. A postcondition clause states a property that will hold when a routine call terminates; it often needs to relate the value that an expression will have at that time to the value it had on *entry* to the procedure. Hence the usefulness of an "Old expression", of the form

```
old some_expression
```

which means: "The value of *some_expression*, captured at the beginning of the routine's execution". Here the postcondition clause

```
count = old count + 1
```

tells us that the routine must increase the number of stations, *count*, by one.

Old expressions, and the **old** keyword, may only appear in postconditions.

When you write a feature in a class, you may assume that the precondition holds at the beginning — as we have seen, this is the client's job — but it is your responsibility to ensure that the postcondition holds when the feature terminates its execution:

Touch of Methodology: Postcondition Principle

A feature must make sure that, if its precondition held at the beginning of its execution, its postcondition will hold at the end.

Class invariants

Preconditions and postconditions are logical properties of your software, each associated with a particular *feature*, such as *i_th*, *remove_all_segments* and *extend* in the examples so far.

We also use logical properties to characterize an entire *class*, above the level of its individual features. Such a property, known as a **class invariant**, expresses relationships between the different queries of a class.

We have encountered such properties in the example of metro lines:

← *"Conventions: Line numbering", page 58; "Properties of start and end lines", page 59.*

- The convention that a line always has at least one station.
- The observation that if *l* is a line then *l*.*south_end* = *l*.*i_th* (1) and *l*.*north_end* = *l*.*i_th* (*l*.*count*).

These are properties not of any particular *l* but of **all** lines. In other words they characterize the class *LINE* as a whole. This is what the invariant of a class is about. It appears as a clause at the end of the class text:

```
invariant
    at_least_one_station: count >= 1
    south_is_first: south_end = i_th (1)
    north_is_last: north_end = i_th (count)
    identical_ends_if_empty: (count = 1) implies (south_end = north_end)
```

The last assertion uses the **implies** operator of logic, studied in the next chapter: *a* **implies** *b* states that *b* has value **True** whenever *a* has value **True**.

→ *"Implication", 5.2, page 84.*

This example is typical of the role of class invariants: expressing consistency requirements between the queries of a class. Here these requirements reflect that some redundancy exists between the queries of class *LINE*: *south_end* and *north_end* provide information also available through *i_th*, applied to arguments 1 and *count*.

Another example would be a class *CAR_TRIP* providing queries such as *initial_odometer_reading*, *trip_time*, *average_speed* and *final_odometer_reading*, with roles implied by their names ("odometer reading" is the total number of kilometers or miles traveled). There is again a certain redundancy between them, which you may capture through a class invariant (where the symbol "∗"denotes multiplication):

```
invariant
    consistent: final_odometer_reading = initial_odometer_reading +
                            trip_time ∗ average_speed
```

There is nothing wrong in principle with including such redundant queries when you design a class: they may all be relevant to the clients, even if they are derived from some of the same internal information about the corresponding objects. But without the invariant, the redundancy might cause confusion or errors. The invariant expresses clearly and precisely how the different queries may relate to each other.

We saw earlier that a precondition must hold at the beginning of a feature call, and a postcondition at the end. An invariant — which applies to all the features of a class, not just a specific one — must hold at both points:

> **_Touch of Methodology_: Class Invariant Principle**
>
> A class invariant must hold as soon as an object is created, then before and after the execution of any of the class features available to clients.

Contracts: a definition

We have seen various kinds of contract — preconditions, postconditions, class invariants — from which a general definition now emerges:

> **_Definition_: Contract**
>
> A contract is a specification of properties of a software element that affect its use by potential clients.

We will use contracts throughout the software to make it clear what each element — class or feature — is about. As noted, they serve for documenting software, especially libraries of components meant (like Traffic) for reuse by many different applications; they help in debugging; and they help us avoid bugs in the first place by writing correct software.

4.7 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A software element presents one or more interface to the rest of the world.
- Classes exist only in the software text; objects exist only during the execution of the software.
- A class describes a category of possible objects.
- Every query returns a result of a type specified in the query's declaration.
- We may specify the interface of a class through a "contract view" which lists all the features of the class — commands and queries — and, for each of them, the properties relevant to *clients* (other classes that use it).

- A feature may have a precondition, specifying initial properties under which it is legitimate to call the feature, and a postcondition, specifying final properties that it guarantees when it terminates.
- A class may have a class invariant, specifying consistency conditions that connect the values of its queries.
- Preconditions, postconditions and class invariants are examples of contracts.
- Among other applications, contracts help for software design, documentation, and debugging.

4-E.8 New vocabulary

API
Assertion
Assertion tag
Boolean
Bug
Class invariant
Client
Client Programmer
Contract
Generating class
GUI
Implementation
Instance
Interface
Library
Postcondition
Precondition
Program interface
Software design
Supplier
Type
User interface

4-E EXERCISES

4-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

4-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise “Concept map”, 3-E.2, page 46.*

4-E.3 Violating a contract

1 Write a simple program (starting for example from the system of this chapter) that uses the query *i_th* of class *LINE*. Run it, using a known *LINE* object, for example *Line8*.

2 Change the argument passed to *i_th* so that it is out of bounds (less than one, or larger than the number of stations). Run the program again. What happens? Explain the messages that you get.

4-E.4 Breaking the invariant

An invariant must be satisfied on object creation, then before and after execution of features by clients. This does not require the invariant to be satisfied *during* the execution of such a feature. Can you think of examples in which it is appropriate for a feature to perform operations that might invalidate the invariant, then other operations that restore it?

← *"Touch of Methodology: Class Invariant Principle", page 68.*

4-E.5 Postcondition vs invariant

You are not sure whether to include a certain property in a routine's postcondition or in the class invariant. What criteria would help you decide?

4-E.6 When to write contracts

The example contracts of this chapter were added to program elements — features and classes — after a first version of these elements was already available. Can you think of circumstances in which it is preferable to write the contracts *before* the implementation of the corresponding program elements?

5

Just Enough Logic

Programming is, for a large part, reasoning. We use computers to perform certain combinations of basic tasks, executed at rates beyond direct human comprehension; to get the results that we need, we must be able to understand the program's possible run-time behaviors, which are nothing but consequences and ramifications of the effects prescribed by our programs, if often very indirect consequences and myriad ramifications. All can, in principle, be deduced from the program text through mere reasoning. It would help us considerably if there were a science of reasoning.

We are in luck, because there *is* such a science: Logic. Logic is the machinery behind the human aptitude to reason. The laws of logic enable us, when told that Socrates is human, and that all humans are mortal, to deduce without blinking that Socrates, then, must be mortal. When someone announces that whenever the temperature in the city rises above 30 degrees a pollution alert will result, so because the temperature today is only 28 degrees there will not be a pollution alert, you will say that his logic is flawed.

Logic is the basis of mathematics; mathematicians will believe a five-line or sixty-page proof only because they accept that each step proceeds according to the rules of logic.

Logic is also at the basis of software development. Already in the last chapter we encountered *conditions* in the contracts associated with our classes and features, for example the precondition "*i* must be between 1 and *count*". We will also use conditions in expressing the actions of a program, for example "If *i* is positive, then execute this instruction".

We have seen in the study of contracts how such conditions appear in our programs in the form of "boolean expressions". A boolean expression may be complex, involving operators such as "**not**", "**and**", "**or**", "**implies**"; this mirrors modes of reasoning familiar in ordinary language: "If it's already 20 minutes past the time for our date *and* she did *not* call *or* send an SMS, it *implies* she will not show up at all". We all intuitively understand what this means, and so far this informal understanding has been good enough for our software conditions too.

No longer. Software development requires precise reasoning, and precise reasoning requires the laws of logic. So before we rush back to the delights of objects and classes we must familiarize ourselves with these laws.

Logic — *mathematical* logic as it is more precisely called —is a discipline of its own, and even just "Logic for Computer Science" is the topic of many textbooks and courses; I hope that you will take such a course or have already taken it. This chapter introduces some essential elements of logic needed to understand programming. More precisely, even though logic in its full glory is the science of reasoning, we need it, just now, for a more limited goal: understanding the part of reasoning having to do with *conditions*. Logic will give us a solid basis for expressing and understanding conditions as they appear in contracts and elsewhere in programs.

The first part of the chapter introduces *boolean algebra*, a form of "propositional calculus" dealing with basic propositions involving specified variables. The second part extends the discussion to *predicate calculus*, which expresses properties of arbitrary sets of values.

5.1 BOOLEAN OPERATIONS

A condition in boolean algebra as well as in programming languages is expressed as a boolean expression, built out of boolean variables and operators, and representing possible boolean values.

Boolean values, variables, operators and expressions

There are two **boolean constants**, also called "boolean values" and "truth values"; we write them **True** and **False** for compatibility with our programming language, although logicians use just **T** and **F**. Electrical engineers, who rely on logic for circuit design, often call them 1 and 0.

→ **True** *and* **False** *are "reserved words" of the programming language; see page 234.*

A **boolean variable** is an identifier denoting a boolean value. Typically we use a boolean variable to express a property that might be either true or false: to talk about the weather we might have the boolean variable *rain_today* to stand for the property that we think rain will fall today.

Starting from boolean constants and boolean variables we may use **boolean operators** to make up a **boolean expression**. For example, if *rain_today* and *cuckoo_sang_last_night* are boolean variables, then the following will be boolean expressions according to the rules studied next:

- *rain_today*
  -- A boolean variable, without operators: already a boolean
  -- expression (the simplest form, along with boolean constants).
- **not** *rain_today*
  -- Using the boolean operator **not**.
- (**not** *cuckoo_sang_last_night*) **implies** *rain_today*
  -- Using the operators **not** and **implies**, and parentheses
  -- to delimit a subexpression.

Each boolean operator — such as **not**, **or**, **and**, =, **implies** as defined below — comes with rules defining the value of the resulting expression from the values of the variables making it up.

We express the boolean operators, like the two boolean constants, through programming language keywords. In mathematical textbooks you will see them expressed as symbols, most of which you could not directly type on your keyboard. Here is the correspondence:

Eiffel keyword	**Common mathematical symbol**
not	¬ or ~
or	∨
and	∧
=	⇔ or =
implies	⇒

In Eiffel, boolean constants, variables and expressions have the type *BOOLEAN*, defined from a class like all types. *BOOLEAN* is a library class, which you may look up in EiffelStudio; you will see all the boolean operators discussed in this chapter.

Negation

The first operator is **not**. To form a boolean expression with **not**, write the operator followed by another expression. That expression can be a single boolean variable, as in **not** *your_variable*; or it can itself be a composite expression (enclosed in parentheses to dispel any ambiguity), as in the following examples where *a* and *b* are boolean variables:

- **not** (*a* **or** *b*).
- **not** (**not** *a*)

For an arbitrary boolean variable *a*, the value of **not** *a* is **False** if the value of *a* is **True**, and **True** if the value of *a* is **False**. We may also express this, the defining property of **not**, through the following table:

a	**not** *a*
True	**False**
False	**True**

This is called a **truth table** and is a standard way of specifying the meaning of a boolean operator: in the first columns (here just one), list all the possible values for the variables involved in an expression that uses the operator; in the last column, list the corresponding value of the expression in each case.

The operator **not** represents **negation**: replacing every boolean value by its opposite, where **True** is the opposite of **False** and conversely.

From the truth table we note interesting properties of this operator:

***Theorems:* Negation properties**

For any boolean expression *e* and any values of its variables:

1 Exactly one of *e* and **not** *e* has value **True**.

2 Exactly one of *e* and **not** *e* has value **False**.

3 One of *e* and **not** *e* has value **True**. (**Principle of the Excluded Middle**.)

4 Not both of *e* and **not** *e* have value **True**. (**Principle of Non-Contradiction**.)

Proof: by definition of a boolean expression, *e* can only have value **True** or **False**. The truth table shows that if *e* has value **True**, then **not** *e* has value **False**; all four properties are consequences of this (and the last two directly of the first).

Disjunction

The operator **or** uses two operands (instead of one for **not**). If *a* and *b* are boolean expressions, the boolean expression *a* **or** *b* has value **True** if and only if at least one of *a* and *b* has that value. Equivalently, it has value **False** if and only if both of the operands have that value. The truth table expresses this:

a	*b*	*a* **or** *b*
True	**True**	**True**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

The first two columns list all four possible combinations of values for *a* and *b*.

The word "or" is taken here from ordinary language in its **non-exclusive** sense, as in "*Whoever made up this regulation must have been stupid or asleep*", which does not rule out that he might have been both.

Ordinary language frequently uses "or" in an *exclusive* sense, meaning that the result will hold if one of the conditions holds but not both: "*Shall we order red or white*?". This corresponds to a different operator, "exclusive or" — **xor** in Eiffel — whose properties you are invited to study by yourself.

→ *See exercise 5-E.12, page 104.*

The **or** operator, non-exclusive, is called **disjunction**. That is not such a good name, because it may suggest an exclusive operator; but it has the benefit of symmetry with "conjunction", the name for our next operator, **and**.

A disjunction has value **False** in only one case out of the four possible value combinations: the last row in the table. This provides an alternative, often useful form of the definition:

Theorem: Disjunction Principle

An **or** disjunction has value **True** except if both operands have value **False**.

The truth table shows that the operator **or** is *commutative*: for any *a* and *b*, the value of *a* **or** *b* is the same as that of *b* **or** *a*. This is also a consequence of the Disjunction Principle.

Conjunction

Like **or**, the operator **and** takes two operands. If *a* and *b* are boolean expressions, then the boolean expression *a* **and** *b* has value **True** if and only if both *a* and *b* have that value. Equivalently, it has value **False** if and only if at least one of the operands has that value. In truth table form:

a	*b*	*a* **and** *b*
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**False**

The application of **and** to two values is known as their **conjunction**, as in the conjunction of two events: "*Only the conjunction of a full moon and Saturn's low orbit can bring true romance to a Sagittarius*" (perhaps not the kind of example that directly influences mathematical logicians).

Studying **and** and **or** reveals a close correspondence, or **duality**, between the two operators: many interesting properties of either operator yield a property of the other if we swap **True** and **False**. For example the Disjunction Principle has a dual that applies to conjunction:

← *Page 75.*

Theorem: Conjunction Principle

An **and** conjunction has value **False** except if both operands have value **True**.

Like **or**, the operator **and** is commutative: for any *a* and *b*, *a* **and** *b* has the same value as *b* **and** *a*. This property can be seen on the truth table; it is also a consequence of the Conjunction Principle.

Complex expressions

You may use boolean operators — the three already introduced, **not**, **or** and **and**, and the other two described next — to build a more complex boolean expression, and deduce the truth table of the expression from the truth tables defining the operators. Here for example is the truth table for the boolean expression *a* **and** (**not** *b*):

a	*b*	**not** *b*	*a* **and** (**not** *b*)
True	**True**	**False**	**False**
True	**False**	**True**	**True**
False	**True**	**False**	**False**
False	**False**	**True**	**False**

To derive this truth table, it suffices to replace, in the truth table for **and**, each value of *b* by the value of **not** *b* as obtained from the truth table for **not**; a third column has been added to show **not** *b*.

Truth assignment

A boolean variable represents a value that may be either **True** or **False**. The value of a boolean expression depends on the value of its variables. For example by building the truth table for a **and** (b **and** (**not** c)) you would see that this expression has:

- Value **True** if a has value **True**, b also, and c has value **False**.

- Value **False** in all other cases.

The following notion helps express such properties:

Definition: Truth assignment

A **truth assignment** for a set of variables is a particular choice of values, **True** or **False**, for each one of the variables.

So we can say that a **and** (b **and** (**not** c)) has value **True** for exactly one truth assignment of its variables (the one that chooses **True** for a, **True** for b, and **False** for c) and **False** for all other truth assignments.

Each row of the truth table for an expression corresponds, one to one, to a truth assignment of its variables.

It is easy to see that for an expression involving n variables there are 2^n possible truth assignments and hence 2^n rows in the truth table. For example the table for **not**, with one operand, had $2^1 = 2$ rows; the table for **or**, with two operands, had $2^2 = 4$ rows. The number of columns is $n + 1$:

- The first n columns of each row list the values of each of the variables for the corresponding truth assignment.

- The last column gives the expression's value for that truth assignment.

(For explanatory purposes the last example added a column for **not** b.)

If an expression has value **True** for a certain truth assignment, as reflected in the last column for the corresponding row, we say that the truth assignment **satisfies** the expression. For example the truth assignment cited — **True** for a, **True** for b, **False** for c — satisfies a **and** (b **and** (**not** c)); all others don't.

Tautologies

We are often interested in expressions that have value **True** for every truth assignment of their variables. Consider

a **or** (**not** *a*)

This states that for a variable *a* either (or both, although that is not possible):

- *a* has value **True**
- **not** *a* has value **True**.

This is only an informal interpretation; to study the value of this expression we may build its truth table, deduced from those for **or** and for **not**:

a	**not** *a*	*a* **or** (**not** *a*)
True	**False**	**True**
False	**True**	**True**

The second column is not strictly part of the truth table but gives the value of **not** *a*, coming from the table for **not**. Combining this with the truth table for **or** (which tells us that both **True or False** and **False or True** have value **True**) yields the third column.

From that column we see that any truth assignment — meaning here, since there's only one variable, any value of *a*, **True** or **False** — satisfies the expression. Such expressions have a name:

Definition: Tautology

A **tautology** is a boolean expression that has value **True** for every possible truth assignment of its variables.

The property that *a* **or** (**not** *a*) is a tautology was expressed earlier as the **Principle of the Excluded Middle**. ← *Page 74.*

Other simple tautologies, which you should now prove by writing their truth tables, are:

- **not** (*a* **and** (**not** *a*)), expressing the Principle of Non-Contradiction. ← *Also page 74.*
- (*a* **and** *b*) **or** ((**not** *a*) **or** (**not** *b*))

Sometimes it is also interesting to exhibit a property that is *never* true:

> ***Definition:* Contradiction**
>
> A **contradiction** is a boolean expression that has value **False** for every possible truth assignment of its variables.

For example (check the truth table again), a **and** (**not** a) is a contradiction; this restates, more simply, the Principle of Non-Contradiction.

From these definitions and the truth table for **not** it follows that a is a tautology if and only if **not** a is a contradiction, and conversely.

An expression that has value **True** for at least one truth assignment of its variables is said to be **satisfiable**. Obviously:

- Any tautology is satisfiable.
- No contradiction is satisfiable.

There exist, however, satisfiable expressions that are neither tautologies nor contradictions: they have value **True** for at least one truth assignment, and value **False** for at least one other truth assignment. This is the case, for example, with a **and** b and with a **or** b.

> "a is not a tautology" is not the same as "**not** a is a tautology". The second property states that no truth assignment satisfies a or, as just seen, that a is a contradiction. The first property states that at least one truth assignment does not satisfy a; but then some other truth assignments might still satisfy a, in which case a is satisfiable but neither a tautology nor a contradiction.

Equivalence

To prove or disprove tautologies, contradictions and satisfiability, we are soon going to get fed up with writing truth tables. With 2^n rows for n variables, this is tedious; to find that a **and** (b **and** (**not** c)) is satisfiable but neither a tautology nor a contradiction we would have to consider eight cases. We need a better way. For example, you may have resented being asked to use a truth table to show that a **and** (**not** a) is a contradiction if previously you had proved that **not** (a **and** (**not** a)) is a tautology. It's time for more general rules.

The *equivalence* operator helps define such rules. It uses the equals symbol, $=$, and has the following table (the truth table to end all truth tables!) stating that $a = b$ has value **True** if and only if a and b either have both the value **True** or both the value **False**:

a	b	$a = b$
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

This operator is commutative ($a = b$ always has the same value as $b = a$). It is also *reflexive*, that is to say, $a = a$ is a tautology for any a.

→ *See exercise 5-E.3, page 102.*

Although logicians generally use the symbol $\Leftrightarrow$ for equivalence, the equality symbol $=$ is also appropriate because $a = b$ expresses equality in the usual sense, denoting an expression that has value **True** if and only if a and b have the same value. The following property extends this observation:

Theorem: Substitution

For any boolean expressions u, v and e, if $u = v$ is a tautology and e' is the expression obtained from e by replacing every occurence of u by v, then $e = e'$ is a tautology.

Proof sketch: if u does not occur in e, then e' is the same expression as e, and we have seen (reflexivity of $=$) that $e = e$ is a tautology. If u does occur in e, we note that the value of a boolean expression under any particular truth assignment is entirely determined by the value of its subexpressions under that assignment. Here e' differs from e only by having occurrences of the subexpression u replaced by v. Under any particular truth assignment, since $u = v$ is a tautology, these subexpressions will have the same value in e and e'; because the rest of the expression is the same, the value of the entire expression will be the same, implying that the truth assignment satisfies $e = e'$. Since this is the case for any truth assignment, $e = e'$ is a tautology.

This rule is the key to proofs of non-trivial boolean properties. We do proofs by truth tables for the basic expressions only; then we use equivalences to replace expressions by simpler ones. For example, to prove that

$$(a \textbf{ and } (\textbf{not } (\textbf{not } b))) = (a \textbf{ and } b) \qquad \text{-- GOAL}$$

is a tautology, you do not need to write its truth table; first you prove that for any expression x the following general properties are both tautologies:

```
not (not x) = x                  -- T1
x = x                            -- T2
```

T2 is the reflexivity of =, proved from the truth table; T1 is easily proved in the same way. You may then use T1, applied to the expression b, and the Substitution theorem, to replace **not** (**not** b) by just b on the left-hand side of the property GOAL; then applying T2 to a **and** b yields the desired result.

To express that two boolean values are *not* equal, we use /= (the best approximation, with two characters available on all keyboards, of the mathematical symbol ≠). Its definition is that a /= b has the same value as **not** ($a = b$).

De Morgan's laws

Two tautologies are of particular interest in using **and**, **or** and **not**:

Theorems: De Morgan's Laws

The following two properties are tautologies:

- (**not** (a **or** b)) = ((**not** a) **and** (**not** b))
- (**not** (a **and** b)) = ((**not** a) **or** (**not** b))

Proof: either write the truth tables, or better combine the Non-Contradiction, Excluded Middle, Disjunction and Conjunction principles.

These properties make the **and**-**or** duality even more remarkable, by expressing that if you negate either of the two operators you get the other by negating the operands.

Informally interpreting — for example — the first one: "if we say that it's not true that a or b holds, that is exactly the same as if we were saying that neither a nor b holds". Of course we are already at a stage where formal notations such as those of logic, with their precision and concision, are vastly superior to such natural-language statements.

Another aspect of the close association between the **or** and **and** operators is that each is **distributive** with respect to the other, meaning that the following two properties are tautologies:

Theorems: Distributivity of boolean operators

The following two properties are tautologies:

- $(a$ **and** $(b$ **or** $c))$ $=$ $((a$ **and** $b)$ **or** $(a$ **and** $c))$
- $(a$ **or** $(b$ **and** $c))$ $=$ $((a$ **or** $b)$ **and** $(a$ **or** $c))$

Compare to the distributivity of multiplication with respect to addition in arithmetic: if $+$ is addition and $*$ is multiplication, then $m * (p + q)$ is the same as $(m * p) + (m * q)$ for any numbers m, p, q.

Distributivity is easy to prove, for example from truth tables. It helps simplify complex boolean expressions.

Simplifying the notation

To avoid the accumulation of parentheses, it is customary to accept some *precedence rules* that give a standard understanding for boolean expressions, removing ambiguity even if some parentheses are missing. This is the same idea that enables us to understand $m + p * q$, in arithmetic and in programming languages, as meaning $m + (p * q)$ rather than the other possible grouping. We say that the operator $*$ **binds tighter**, or has **higher precedence**, than the operator $+$: it "attracts" the neighboring operands before $+$ gets its chance.

For boolean operators we may use the same precedence as used by the syntax of Eiffel; the order from highest precedence to lowest is:

- **not** binds tightest.
- Then comes equivalence: $=$.
- Then comes **and**.
- Then **or**.
- Then **implies** (studied below).

Under these rules, the expression $a = b$ **or** c **and not** $d = e$, with no parentheses, is legal and means

$(a = b)$ **or** $(c$ **and** $(($**not** $d) = e))$

It is desirable, however, to retain some parentheses to protect readers of your programs from misunderstandings which might lead to errors.

In the recommended style you should *not* drop the parentheses that separate **or** and **and** expressions since the precedence rule making **and** bind tighter than **or** is arbitrary. It is also better to keep the parentheses around a **not** subexpression used as operand of an equivalence, to avoid confusing $(\textbf{not}\ a) = b$ with $\textbf{not}\ (a = b)$. You may, however, drop the parentheses around a subexpression of the form $x = y$ where x and y are single variables. So for the last example you would just write

$$a = b\ \textbf{or}\ (c\ \textbf{and}\ (\textbf{not}\ d) = e)$$

Another property that simplifies the notation is the **associativity** of certain operators. In arithmetic we commonly write $m + p + q$ even though it could mean $m + (p + q)$ or $(m + p) + q$, because the choice does not matter: these two expressions have equal values, reflecting that addition is an *associative* operation. Multiplication is also associative: $m * (p * q)$ always has the same value as $(m * p) * q$. In boolean logic both operators **and** and **or** are associative, as expressed by the following tautologies:

$$(a\ \textbf{and}\ (b\ \textbf{and}\ c)) \quad = \quad ((a\ \textbf{and}\ b)\ \textbf{and}\ c)$$
$$(a\ \textbf{or}\ (b\ \textbf{or}\ c) \quad = \quad ((a\ \textbf{or}\ b)\ \textbf{or}\ c)$$

For the proofs: you may write truth tables but it is easier to use previous rules. In the first example, the left side is true, from the Conjunction Principle, if and only if both a and b **and** c are true, that is to say — applying that Principle again — if and only if all three of a, b and c are true; but from the same reasoning this is also the case with the right-hand side, so the two sides are equivalent (satisfied under exactly the same truth assignments). For the second line the reasoning is the same, using the Disjunction Principle.

This enables us to write expressions of the form a **and** b **and** c, or a **or** b **or** c, without risk of confusion. To summarize:

Touch of Style: Parentheses for boolean expressions

In writing subexpressions of a boolean expression, drop the parentheses:

- Around "$a = b$" if a and b are single variables.
- Around successive terms if they each involve a single boolean variable and are separated by the same associative operators.

For clarity and to help avoid errors, retain other parentheses, without taking advantage of precedence rules.

5.2 IMPLICATION

One more basic operator — along with **not**, **or**, **and** and equivalence — belongs to the basic repertoire: implication. Although similar to the others, and in fact close to **or**, it requires some attention because its precise properties initially seem, for some people, to contradict intuitive views of the concept of implication in ordinary language.

Definition

The simplest way to define the **implies** operator is in terms of **or** and **not**:

***Definition*: Implication**

The value of a **implies** b, for any boolean values a and b, is the value of (**not** a) **or** b

→ *This temporary definition will be slightly revised on page 94.*

This gives the truth table (which could serve as an alternative definition):

a	b	a **implies** b
True	**True**	**True**
True	**False**	**False**
False	**True**	**True**
False	**False**	**True**

It is the same as the table for **or** if we switch **True** and **False** values for a. The result of a **implies** b is true for all truth assignments except in one case, the highlighted entry: when a is true and b false.

← *Page 75.*

In a **implies** b the first operand a is called the **antecedent** of the implication, and the second operand b is called its **consequent**.

The principles we saw for conjunction and especially disjunction have a direct counterpart for implication:

***Theorem:* Implication Principle**

An implication has value **True** except if its antecedent has value **True** and its consequent has value **False**.

As a consequence, it has value **True** whenever one of the following holds:

I1 The antecedent has value **False**.

I2 The consequent has value **True**.

Relating to inference

The name "**implies**" suggests that we can use the implication operator to infer properties from others. This is indeed permitted by the following theorem:

Theorem: Implication And Inference

I3 If a truth assignment satisfies both *a* and *a* **implies** *b*, it satisfies *b*.

I4 If both *a* and *a* **implies** *b* are tautologies, *b* is a tautology.

Proof: To prove I3, consider a truth assignment *TA* that satisfies *a*. If *TA* also satisfies *a* **implies** *b*, then it must satisfy *b*, since otherwise under row 2 of the truth table for **implies** the value of *a* **implies** *b* would be **False**. To prove I4, note that if *a* and *a* **implies** *b* are tautologies this reasoning is valid for any truth assignment *TA*.

← The highlighted entry on page 84. This property is also a consequence of the Implication Principle.

This property legitimates the usual practice, when we want to prove a property *b*, to identify a possibly "stronger" property *a*, and prove separately that

- *a* holds.
- *a* **implies** *b* holds.

Then we may deduce that *b* holds.

The term "stronger" used here is useful in the practice of reasoning with contracts of programs, and deserves a precise definition:

Definitions: Stronger, weaker

For two non-equivalent expressions *a* and *b*, we say that:

- "*a* is **stronger** than *b*" if and only if *a* **implies** *b* is a tautology.
- "*a* is **weaker** than *b*" if and only if *b* is stronger than *a*.

The definitions assume *a* and *b* to be non-equivalent because it could be confusing to say that *a* is stronger than *b* if they might be the same. In such cases we will use "stronger than or equal to" and "weaker than or equal to" (as with relations between numbers: "*greater than*", "*greater than or equal to*").

Getting a practical feeling for implication

How does the definition of **implies** relate to the usual notion of implication, expressed in ordinary language by such locutions as "If … then …"?

In such everyday use, implication often indicates causality: "*If* we get any more sun, *then* this will be a vintage year for Bourgogne" suggests that one event causes another. The **implies** of logic does not connote causality, it simply states that whenever a certain property is true another one must be too. The example just given can also be interpreted this way if we ignore any hint of causality.

Another typical example is (at the Los Angeles airport, trying to check in for Santa Barbara): "*If* your ticket says Flight 3035, *then* you are not flying tonight", perhaps because the plane is grounded for mechanical problems and this was the last flight. There is no causality here: what is printed on the ticket did not cause the plane to malfunction. It's simply that for anyone to whom the property "Reserved flight is 3035" applies, the property "can fly today" does not hold. Logic's **implies** operator covers such scenarios.

What — surprisingly — surprises many people is property I1 of the Implication Principle, resulting from the last two rows of the truth table: that whenever a is false the implication a **implies** b is true, regardless of the value of b. In fact this *does* correspond to the usual idea of implication:

← *Page 84.*

1 *"If I am the governor of California, two plus two equals five"*

2 *"If two plus two equals five, then I am the governor"*

3 *"If two plus two equals five, then I am not the governor"*

4 *"If I am the governor, two plus two equals four"*

5 "*If I am the governor, it will rain today"*

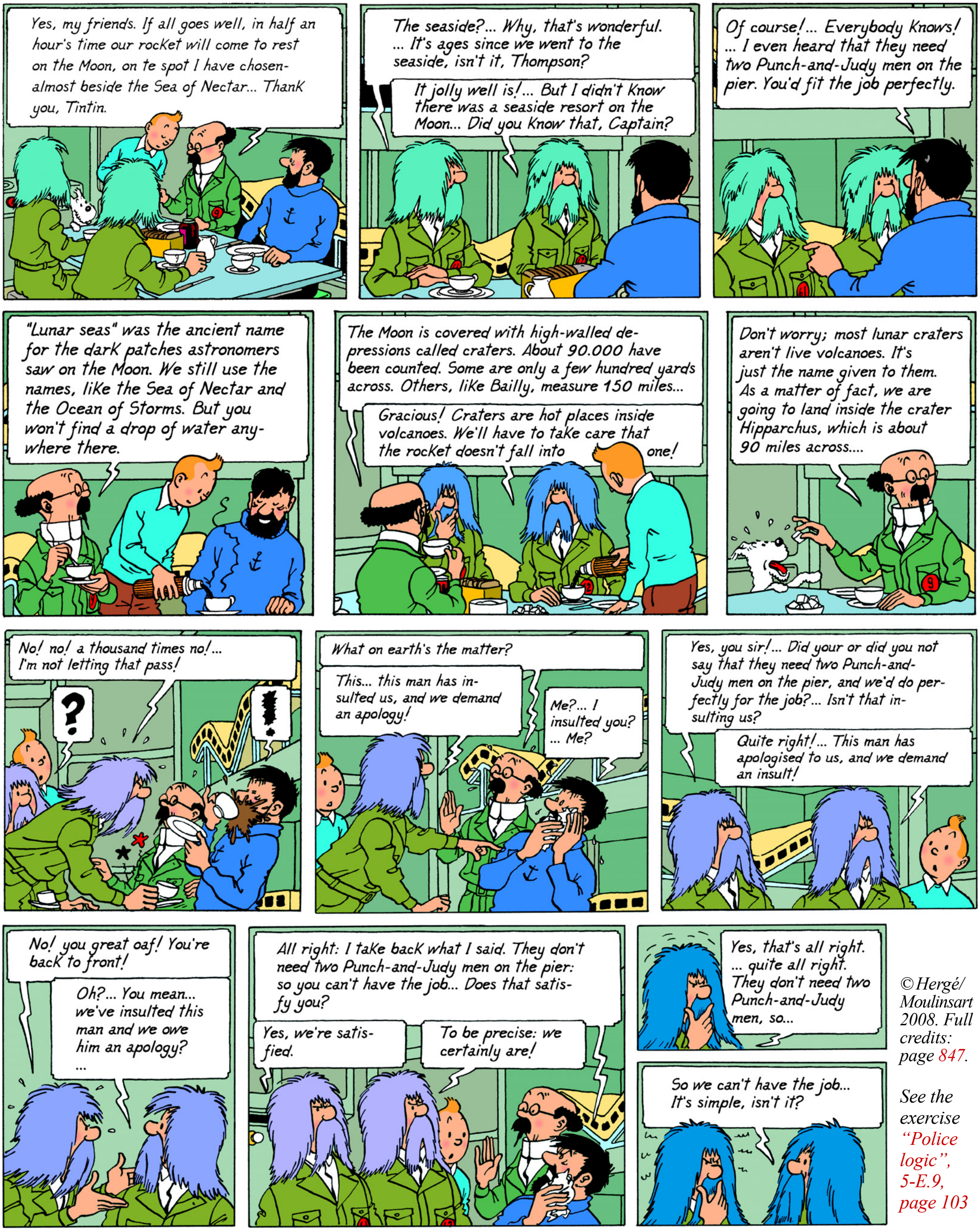
6 "*If it rains today, I will not be elected governor before the end of the year"*

Given that I am not the governor and do not expect to run for the job, all these implications are true — regardless, for the last two, of today's weather.

The little argument between Captain Haddock and the detectives Thomson and Thompson in the Tintin extract on the facing page provides a good opportunity to examine implication arguments (which ones are correct and which ones are flawed?).

← *Exercise "Police logic", 5-E.9, page 103.*

All that "If a, then b" tells us is that whenever the antecedent a holds, the consequent b must hold too. So the only possibility for this implication to be false is (second row, with highlighted entry, in the truth table) for a to be true and b false. Cases in which the antecedent does not hold (I1), and cases in which the consequent holds (I2), do not suffice to determine the truth of the implication as a whole.


Yes, my friends. If all goes well, in half an hour's time our rocket will come to rest on the Moon, on te spot I have chosen- almost beside the Sea of Nectar... Thank you, Tintin.
The seaside?... Why, that's wonderful. ... It's ages since we went to the seaside, isn't it, Thompson?
It jolly well is!... But I didn't know there was a seaside resort on the Moon... Did you know that, Captain?
Of course!... Everybody knows! ... I even heard that they need two Punch-and-Judy men on the pier. You'd fit the job perfectly.
"Lunar seas" was the ancient name for the dark patches astronomers saw on the Moon. We still use the names, like the Sea of Nectar and the Ocean of Storms. But you won't find a drop of water anywhere there.
The Moon is covered with high-walled depressions called craters. About 90.000 have been counted. Some are only a few hundred yards across. Others, like Bailly, measure 150 miles...
Gracious! Craters are hot places inside volcanoes. We'll have to take care that the rocket doesn't fall into one!
Don't worry; most lunar craters aren't live volcanoes. It's just the name given to them. As a matter of fact, we are going to land inside the crater Hipparchus, which is about 90 miles across....
No! no! a thousand times no!... I'm not letting that pass!
?
!
What on earth's the matter?
This... this man has insulted us, and we demand an apology!
Me?... I insulted you? ... Me?
Yes, you sir!... Did your or did you not say that they need two Punch-and-Judy men on the pier, and we'd do perfectly for the job?... Isn't that insulting us?
Quite right!... This man has apologised to us, and we demand an insult!
No! you great oaf! You're back to front!
Oh?... You mean... we've insulted this man and we owe him an apology? ...
All right: I take back what I said. They don't need two Punch-and-Judy men on the pier: so you can't have the job... Does that satisfy you?
Yes, we're satisfied.
To be precise: we certainly are!
Yes, that's all right. ... quite all right. They don't need two Punch-and-Judy men, so...
So we can't have the job... It's simple, isn't it?

© Hergé/ Moulinsart 2008. Full credits: page 847.

See the exercise "Police logic", 5-E.9, page 103

Beginners sometimes have trouble with accepting that "*a* **implies** *b*" can be true if *a* is false; most of that trouble, I guess, comes from the case (I1) in which *a* is false and *b* is true — such as 1, 2 and possibly 5 and 6 above — although there is nothing wrong with it. In fact, the misunderstanding may be due to a common distortion of reasoning which leads some people, equipped with the knowledge that *a* **implies** *b*, to infer happily that if *a* does not hold then *b* must not hold either! Typical examples:

1 "All professional politicians are corrupt. I am not a professional politician, so I am not corrupt and you must vote for me". If the premise is true, it tells us something about professional politicians, but nothing at all about anyone else!

2 "Whenever I take my umbrella it doesn't rain, so I will leave my umbrella at home as we badly need some rain right now." Joke of course, but suggesting the same flawed reasoning.

3 "All recent buildings in this area have bad thermal isolation. This is an older building, so it must be more comfortable in hot summers".

Each of these cases involves a property *a* that implies another *b*, and an erroneous deduction that the negation of *a* implies the negation of *b*. We cannot deduce any such thing. All we know is that if *a* holds then *b* will hold; if *a* does not hold, knowledge of the implication tells us nothing interesting. Couched in the language of logic, the flaw is to believe that

$$\boxed{(a \textbf{ implies } b) \;=\; ((\textbf{not } a) \textbf{ implies } (\textbf{not } b))}$$

Warning: *not a tautology (see exercise "Implication and negation", 5-E.10, page 103).*

is a tautology. Or perhaps it's just to imagine the slightly less powerful (*a* **implies** *b*) **implies** ((**not** *a*) **implies** (**not** *b*)). Neither is a tautology, as they both have value **False** when *a* has value **False** and *b* has value **True**.

Reversing an implication

Although the last two properties are not tautologies, there is an interesting tautology of the same general style:

$$\boxed{(a \textbf{ implies } b) \;=\; ((\textbf{not } b) \textbf{ implies } (\textbf{not } a)) \quad \text{-- REVERSE}}$$

Proof: we just expand the definition of **implies**. For the left side, it gives (**not** *a*) **or** *b*; for the right side, (**not** (**not** *b*)) **or** (**not** *a*). From a previous tautology, we know that (**not** (**not** *b*)) is *b*; from the commutativity of **or**, the right side has the same value as the left side for any truth assignment.

← *T1, page 81.*

Alternatively, we may note in the truth table for **implies** that swapping *a* and *b* then negating both yields back the original table.

This property, REVERSE, states that if b holds whenever a does, then from the knowledge that b does not hold we may infer that a also doesn't. (The informal justification is clear, using reasoning by contradiction: if a were true, then the implication tells us that b would be true; but we are precisely assuming that b does not hold.)

Using this rule, we may replace the earlier flawed examples by logically sound deductions:

1 “All professional politicians are corrupt. She is not corrupt, so she cannot be a professional politician.”

2 “Whenever I take my umbrella it doesn't rain: since weather.com says it is going to rain, I might just as well leave my umbrella at home.”

3 “Since all recent buildings in this area have bad isolation and this room remains cool in spite of the heat outside, the house must be older.”

5.3 SEMISTRICT BOOLEAN OPERATORS

Computer programming fundamentally relies on mathematical logic, to the point that some people consider programming to be just an extension of logic. This is all the more remarkable given that modern logic was established in the first few decades of the twentieth century, before there was any hint of computers in today's sense.

Touch of history: The road to modern logic

Logic goes back to the Ancients, Aristotle in particular, who defined the rules of “Rhetorics”, fixing some forms of deduction. In the eighteenth century Leibniz stated that reasoning was just a form of mathematics. In the nineteenth century, the English mathematician George Boole defined the calculus of truth values (hence “boolean”). The big push for logic in the following century was the realization that mathematics as practiced until then was shaky and could lead to contradictions; the goal pursued by the creators of modern mathematical logic was to correct this situation by giving mathematics a solid, rigorous foundation.

Applying logic to programming brings up some issues often overlooked in purely mathematical uses of logic. An example, important in programming practice, is the need for non-commutative variants of **and** and **or**.

Consider the following question, given a metro line l and an integer n:

> *“*Is the n-th station of line l an exchange?*”*

We might express it as the boolean-valued expression

```
l.i_th (n).is_exchange                                                    [S1]
```

Not the correct form (see [S3] on page 93 below).

where *is_exchange* is a boolean-valued query of class *STATION*, indicating whether a station is an exchange; the query *i_th*, seen in the previous chapter, delivers the stations of a line, each identified by an index, here *n*.

← “The stations of a line”, page 57.

The expression above, [S1], appears to do the job: *l* denotes a line; *l.i_th* (*n*), denotes its *n*-th station, an instance of class *STATION*; so *l.i_th* (*n*)*.is_exchange*, applying the query *is_exchange* to this station, tells us, through a boolean value, whether it is an exchange station.

But we have not said anything about the value of *n*. So *l.i_th* (*n*) may not be defined, since the query *i_th* had a precondition:

← Page 62.

```
i_th (i: INTEGER): STATION
        -- The i-th station on this line
    require
        not_too_small: i >= 1
        not_too_big: i <= count
```

In the absence of further information on *n*, it is incorrect to use the expression [S1] since its result is not defined for $n < 1$ or $n > l.count$.

How can we write a correct expression with the intended meaning? If $n < 1$ or $n > l.count$, it is reasonable to consider that the answer to our informal question, “Is the *n*-th station of line *l* an exchange?”, cannot be “Yes”, as this would imply that we certify that a certain station is an exchange, and we cannot do this if no such station exists. Since in the boolean world there are only two possibilities, the answer has to be “**No!**”, meaning formally that the boolean expression should have value **False**. To get this behavior we might try to express the desired property not as [S1] but as

```
(n >= 1) and (n <= count) and l.i_th (n).is_exchange                      [S2]
```

Still not right (see [S3]).

But this is still not good enough. The problem is that if n is out of bounds, for example $n = 0$, the last term *l*.*i_th* (*n*).*is_exchange* is not defined. If we are only interested in the value of [S2], we might not care, because the Conjunction Principle that tells us this value can only be **False** since the first term, $n >= 1$, has value **False**; the second and third terms do not affect the result.

Assume, however, that the expression appears in a program and gets evaluated during the program's execution. The operator **and**, as we have seen, is commutative; it is legitimate for the execution, when it needs to compute *a* **and** *b*, to compute both operands *a* and *b* and then combine their values using the truth table for **and**. But then the computation of [S2] will fail when it tries to evaluate the last term.

If that evaluation were conceptually required, we could do nothing: a computation that tries to evaluate an expression with undefined value should fail. It's like trying to evaluate the numerical expression $1 / 0$. But in this case we may prefer that when the first term has value **False** the evaluation will, instead of failing, return the value **False**, consistent with the definition of **and**.

This is not achievable with the usual commutative boolean operators: we cannot prevent their computer versions from evaluating both operands and thereby risking failure.

The case illustrated by this example — evaluating a condition that only makes sense if another condition is also satisfied — occurs so frequently in practice that we need a solution. There are three possible ways to go.

The first would be to try to recover from the failure. If an operand to a boolean expression is undefined, so that its evaluation leads to failure, we could have a mechanism that "catches" the failure and tries to see if other terms suffice to determine a value for the expression as a whole. Such a mechanism means that failure is not like real death but more like death in video games, where you can get new lives (as long as you continue paying). The mechanism exists: it is called **exception handling** and enables you to plan for accidents of execution and try to recover. In the present case, however, it would be (if one dares use the term) overkill. It requires far too much special programming for what is, after all, a simple and common situation.

→ *Exceptions are discussed in a later chapter: "An introduction to exception handling", 7.10, page 200.*

The second way would be to decide that **and**, as we understand it in programming, is not commutative any more (the same would, for duality, hold of **or**). In computing *a* **and** *b*, we would have the guarantee that *b* will not be evaluated if *a* has been evaluated to **False**, the result in that case being **False**. The problem with this approach is that it is unpleasant to make the software version of a well-accepted mathematical concept depart from its mathematical meaning. More pragmatically, the commutativity of **and** and **or** when both operands are defined can help make the computation faster, as it may be advantageous to evaluate the second expression first, or even, if the hardware includes several processors, to evaluate them in parallel.

Such improvement of execution speed, known as **optimization**, is generally carried out not by programmers but by compilers (the tools that translate your programs to machine code).

→ *"Compiler tasks", page 336.*

The third way — the one we retain — is to accept the usefulness of non-commutative boolean operators but give them different names to avoid any semantic confusion. The new variant of **and** will be written **and then**; by duality we also have a variant of **or**, called **or else**. In each case it is a double keyword, written with a space between the two constituent words. The semantics follows from the previous discussion:

Touch of Semantics: Semistrict boolean operators

Consider two expressions *a* and *b* which may be *defined* or not, and if defined have boolean values. Then:

- *a* **and then** *b* has the same value as *a* **and** *b* if both *a* and *b* are defined, and in addition has value **False** whenever *a* is defined and has value **False**.
- *a* **or else** *b* has the same value as *a* **or** *b* if both *a* and *b* are defined, and in addition has value **True** whenever *a* is defined and has value **True**.

If you are wondering about the name: we say that an operator is *strict* (as in "My mom is *strict* about having everyone at the table before any of us starts eating") if it insists, to produce its result, on having all operand values available, even those that the evaluation may turn out not to need. An operator is "*non-strict*" on an operand if it may in some cases yield a meaningful result even when that operand does not have a defined value. We call **and then** and **or else** *semistrict* because they are strict on their first operand and non-strict on the second.

Saying "non-commutative" would be acceptable for the operators seen so far, but we will need semistrict variants of operators such as **implies**, which is not commutative in the first place.

Another way to define the semantics of the semistrict operators is to introduce a variant of truth tables where every operand and result may have three values rather than just two: **True**, **False** and *Undefined*.

→ *Exercise 5-E.15, page 106.*

Whenever *a* **and** *b* is defined, *a* **and then** *b* is defined and has the same value, but the converse is not true. The same holds for **or else** relative to **or**.

With this notation the correct way to express our example condition is

```
((n >= 1) and (n <= count)) and then l.i_th (n).is_exchange        [S3]
```

The previous version [S2] had two **and** operators, but only the second one needs to be turned into an **and then**; between the first two terms, grouped here in parentheses for clarity, a plain **and** is good enough since both will always be defined. This is a general advice:

Touch of Methodology: Choosing between ordinary and semistrict boolean operators

In expressing contracts and other conditions:

- Use the ordinary boolean operators, **or** and **and**, when you can guarantee that both operands are defined whenever the execution needs to evaluate the condition.
- If a condition only makes sense when another is false, use **or else**.
- If a condition only makes sense when another is true, use **and then**.

Our example, [S3], corresponds to the last case.

In the first case it would not be *wrong* to use the semistrict version, but this would needlessly prescribe a particular evaluation order; it is preferable to avoid such "overspecification" and stick instead to the operators with standard mathematical properties. This also leaves compilers free to optimize the order of operand evaluation.

The notion of semistrict operator is applicable to more than mathematical logic and software:

Touch of Practice: Semistrict operators and you

The semistrict operators reflect modes of reasoning that are common in daily life. Wherever you see the phrase "*if any*" you may suspect that semistrictness is involved. A credit application form might stipulate that the spouse, *if any*, must be a co-signer; we may understand this as *is_single* **or else** *spouse_must_sign* or, in more explicit programming terms:

```
(spouse = Void) or else spouse.must_sign
```

where **Void** denotes the absence of an object. In either form the second operand of the **or else** would not make sense with a strict **or**, since when the first operand has value **True** the notion of spouse is not defined.

→ *"Void references", 6.3, page 111.*

Semistrict implication

Implication also has a semistrict variant. A routine with arguments *l*: *LINE* and *i*: *INTEGER* might use the precondition

```
((i >= 1) and (i <= count)) implies l.i_th (i).is_exchange
```

meaning: apply the routine only if the *i*-th station of line *l*, assuming it exists, is an exchange.

This makes sense only with a semistrict interpretation of **implies**. Such a scheme — an expression of the form *a* **implies** *b* where *b* is defined only when *a* is true — occurs so frequently that for this operator, which is not commutative, the semistrict version seems appropriate in all cases. Such a convention is also consistent with the Implication Principle and its insistence (clause I1) that *a* **implies** *b* has value **True**, regardless of *b*, whenever *a* has value **False**.

← *"Theorem: Implication Principle", page 84.*

So we take the semistrict version as the definition of **implies**:

***Definition*: Implication with possible undefinedness**

The value of *a* **implies** *b*, for any *a* and *b* where *b* may not be defined, is the value of

(**not** *a*) **or else** *b*

← *This supersedes the temporary definition of page 84.*

The ordinary life example of semistrictness cited above falls in this category; we may now write it with semistrict **implies** as

```
(spouse /= Void) implies spouse.must_sign
```

Many uses of "if any", for example in legal documents, follow this pattern.

5.4 PREDICATE CALCULUS

The concepts discussed so far belong to a part of logic called **propositional calculus**, meaning that it deals with basic *propositions*, each stating a single property *p* that might be true or false: *n* has a positive value, I am the governor of California, it is full moon tonight. "Single property", in these examples, means that *p* characterizes a single object — the number *n*, me, the current night — or a finite set of explicitly listed objects, as in "I am not the governor and it is not a full moon tonight".

Another theory is directly useful in programs and discussions of programs: **predicate calculus**, which considers whether a property holds for the elements, not individually specified, of a *set* of objects.

Generalizing "or" and "and"

Given a set of objects *E* and a property *p* of objects, predicate calculus deals with two basic questions, generalizing "or" and "and":

1 Does *at least one* of the objects in *E* satisfy *p*?

2 Does *every one* of the objects in *E* satisfy *p*?

For example, we have seen that any metro line contains stations, and that stations may be exchanges. We may ask, about a particular line:

A1 Is at least one of the stations of Line 8 an exchange?

A2 Are all of the stations of Line 8 exchanges?

If you know all the stations by name you can express these questions as boolean expressions. A1 is an **or** expression and A2 is an **and** expression:

```
L1  Station_Balard.is_exchange or Station_La_Motte.is_exchange or
    Station_Concorde.is_exchange or … [Include all stations on line] …

L2  Station_Balard.is_exchange and Station_La_Motte.is_exchange and
    Station_Concorde.is_exchange and … [Include all stations on line] …
```

← *List of stations from figure on page 58.*

Both use the boolean-valued query *is_exchange* of class *STATION* to tell us if a station is an exchange. You would have to complete the expressions by including a term for each station of the line.

You can avoid naming individual stations by using the query *i_th* of class *LINE* which, as seen in the preceding chapter, gives us the *i*-th station of a line for any applicable *i*:

← *"The stations of a line", page 57.*

```
M1 Line8.i_th (1).is_exchange or Line8.i_th (2).is_exchange or
   … [Include all numbers from 1 to Line8.count]

M2 Line8.i_th (1).is_exchange and Line8.i_th (2).is_exchange and
   … [Include all numbers from 1 to Line8.count]
```

but that is still inconvenient as you must explicitly list all stations. In particular you cannot write, for either question, an expression that would make sense for *any* line, since different lines have different numbers of stations.

Predicate calculus addresses such cases by introducing **quantifier** expressions that describe the application of a property to a set of objects, letting you mention only that set, for example a metro Line, rather than individually listing every object — every station. There are two quantifiers:

- The **existential quantifier**, *exists*, or ∃ in mathematical notation, to state that *at least one* member of the set satisfies the property.
- The **universal quantifier**, *for_all*, or ∀ in mathematical notation, to state that *every* member of the set satisfies the property.

When you would need boolean operations on an arbitrary number of operands, *exists* generalizes **or**, and *for_all* generalizes **and**. If *Line8_stations* denotes a list of stations, the mathematical notations are:

Q1 ∃ *s*: *Line8_stations* | *s***.***is_exchange*

Q2 ∀ *s*: *Line8_stations* | *s***.***is_exchange*

which you may read aloud respectively as:

- **There exists** an *s* in *Line8_stations* such that *s***.***is_exchange* is true.
- **For all** *s* in *Line8_stations*, then *s***.***is_exchange* is true.

Rather than using a bar "|" as above to separate the property, here *s***.***is_exchange*, from the specification of the set of objects across which it will range, mathematicians often use a period "**.**" or a comma "**,**"; but for us this would be ambiguous since, as you know, we need these symbols for other purposes.

Q1 and Q2 are mathematical notations, not programming notations. We will shortly see how to express such properties in a program.

Precise definition: existentially quantified expression

The notations using existential and universal quantifiers, as just illustrated, are new forms of (mathematical) boolean expression, complementing the expressions of propositional calculus seen earlier in this chapter.

The definition of the existential quantifier is straightforward:

Definition: Existentially quantified expression

The value of the expression

∃ *s*: *SOME_SET* | *s***.***some_property*

is **True** if and only if at least one member of the given set *SOME_SET* satisfies the given property *some_property*.

For example let X be the set of integers $\{3, 7, 9, 11, 13, 15\}$ (that is to say, the set consisting of the integers listed between braces) and for any integer n let $n.is\_odd$ be the property that n is odd, $n.is\_even$ the property that it is even, and $n.is\_prime$ the property that it is a prime number. Then:

- $\exists\ n: X \mid n.is\_odd$ means that at least one member of X is odd; the expression has value **True** since we can take, for example, 3 as evidence that there is one such member. In this case we may take any other member of the set as evidence since they are all odd.

- $\exists\ n: X \mid n.is\_prime$ means that at least one member of X is prime; this expression also has value **True** since we may again take 3, for example, as evidence. It does not matter that some other member or members, such as 9, do not satisfy the property, since the truth of an existentially quantified expression only requires one example.

- $\exists\ n: X \mid n.is\_even$ means that at least one member of X is even; this expression has value **False** since no element of X is even.

These examples illustrate how you may prove or disprove an existentially quantified expression $\exists\ s: SOME\_SET \mid s.some\_property$:

E1 To prove that it is true, it suffices to exhibit *one* element of $SOME\_SET$ that satisfies the property. Once you have found such an element, others have no influence on the result. This means in particular that you may not need to investigate all elements of the set.

E2 To prove that it is false, you must prove that *no* element of $SOME\_SET$ satisfies the property. That *some* do not satisfy it is not enough to determine the result. This means in particular that you **must** consider all the elements.

Precise definition: universally quantified expression

For an expression using a universal quantifier

$$\boxed{\forall\ s: SOME\_SET \mid s.some\_property}$$

the informal definition of its value is that it is **True** if and only if every element of *SOME_SET* satisfies *some_property*. This is not quite precise enough, however, because of the case of an empty set (discussed next). A better approach is to base the definition on what has just been specified for *existentially* quantified expressions:

→ *"The case of empty sets", page 99.*

Definition: Universally quantified expression

The value of the expression

$\forall\ s\colon SOME\_SET \mid s.some\_property$

is the value of

$\textbf{not}\ (\exists\ s\colon SOME\_SET \mid \textbf{not}\ s.some\_property)$

This says that the $\forall$ expression has value **True** if and only if there is *no* member of the given set that does *not* satisfy the given property. It sounds like a contorted way of expressing what we want: that every element satisfies the property. In your writing classes you were probably told to avoid double negation, replacing "There's no course I don't like in this great university!" by "I like all courses here". The reason for the double negation is that we must be careful about the case of empty sets. Before examining this case, let us consider again our example set of integers X defined as $\{3, 7, 9. 11, 13, 15\}$:

- $\forall\ n\colon X \mid n.is\_odd$ means that all members of X are odd; the expression has value **True** since 3, 7, 9, 11, 13 and 15 are all odd numbers.
- $\forall\ n\colon X \mid n.is\_prime$ means that all members of x are prime numbers; this expression has value **False** since we can take 9, for example, as evidence that at least one member is not prime. We could also use another non-prime member as evidence — the other possibility is 15 — but one is enough to prove that the universally quantified expression is false.
- $\forall\ n\colon X \mid n.is\_even$ means that all members of X are even; this expression has value **False** since, for example, 3 is not even. Here *any* other member of the set could serve as evidence since none is even, but again one is enough.

These examples illustrate how you may prove or disprove a universally quantified expression $\forall\ s\colon SOME\_SET \mid s.some\_property$ (compare with those for existential quantification, E2 and E1 on the previous page):

U1 To prove that it is true, you must prove that *every* element of *SOME_SET*, if any, *satisfies* the property. That *some* satisfy it is not enough to determine the result. This means in particular that you **must** consider all the elements.

U2 To prove that it is false, it suffices to exhibit *one* element of *SOME_SET* that *does not satisfy* the property. Once you have found such an element, others have no influence on the result. This means in particular that you may not need to investigate all elements of the set.

The relationship between the existential and universal quantifiers generalizes the duality between **or** and **and**. In particular the following two properties generalize De Morgan’s Laws: ← *Page 81.*

$$\begin{aligned} \textbf{not}\ (\exists\, s\colon E \mid P) &= \forall\, s\colon E \mid \textbf{not}\ P \\ \textbf{not}\ (\forall\, s\colon E \mid P) &= \exists\, s\colon E \mid \textbf{not}\ P \end{aligned}$$

> The first property follows from the definition of $\forall$; the second property follows from applying the first to **not** P and negating both sides.

The case of empty sets

The set $\mathit{SOME\_SET}$ of possible values considered in a quantified expression might be empty. The effect on the two quantifiers reflects their duality:

- $\exists\, s\colon \mathit{SOME\_SET} \mid s.\mathit{some\_property}$ is true, according to its definition, if and only if some member of $\mathit{SOME\_SET}$ satisfies $\mathit{some\_property}$. If $\mathit{SOME\_SET}$ is empty, it has no member, and hence no member satisfying the property. So the value of the expression in this case, regardless of $\mathit{some\_property}$, is always **False**.
- $\forall\, s\colon \mathit{SOME\_SET} \mid s.\mathit{some\_property}$ is false if and only if some member of $\mathit{SOME\_SET}$ does *not* satisfy $\mathit{some\_property}$. If $\mathit{SOME\_SET}$ is empty, there will not be any such “counter-example” member since there is no member at all. So the value of the expression in this case is **True**.

We may also view the second case as a consequence of the definition of the universally quantified expression $\forall\, s\colon \mathit{SOME\_SET} \mid s.\mathit{some\_property}$ in terms of the existentially quantified one, as ← *Page 98.*

$$\textbf{not}\ (\exists\, s\colon \mathit{SOME\_SET} \mid \textbf{not}\ s.\mathit{some\_property})$$

By the previous convention, the whole $(\exists\, s\colon \mathit{SOME\_SET} \mid \dots)$ expression in parentheses has value **False** if $\mathit{SOME\_SET}$ is empty, so the $\forall$ expression, deduced from it by applying **not**, has value **True**.

Concretely, this simply means that we may consider every statement of the form “Every object of such-and-such a kind satisfies this property” as true if there is no object of the given kind. So I can say “I promise to you that every blond student in this room will be elected governor before the end of the year”, and even back it with “if not, I will pay every one of you a million euros on January 1st”, if I have (carefully) checked that everyone in the room has black hair. The statement “Every blond student in this room will be elected governor” is indeed true because it is of the form $\forall\, s\colon \mathit{SOME\_SET} \mid \dots$ for an empty $\mathit{SOME\_SET}$, which is true regardless of what comes after the “|”.

Having studied logic, however, you should never promise anything like "A blond student in this room will be elected governor" because it makes you responsible for identifying a fair-haired student *and* rigging the election.

As a result of these observations, the official name of the universal quantifier, "*For all*", is not so good because "*all*" suggests, at least informally, that there are some elements to be talked about. Better names would be "*For all, if any*", or just "*For any*". (It would be nice indeed to call the two quantifiers ∃ and ∀, symmetrically, "*For some*" and "*For any*".) This would not absolutely preclude confusion anyway, so we will continue saying "*For all*" (and "*There exists*") like everyone else, but you have to remember that this is just an informal name and that the mathematical interpretation of *For all* gives a **True** answer — just as *Exists* gives **False** — if there are no elements to be probed for the given condition.

> Another way to express this property is that if we consider an existential quantification on a set of values to mean a_1 **or** a_2 **or** … **or** a_n, and the universal quantification to mean a_1 **and** a_2 **and** … **and** a_n, then as n goes to zero the disjunction will yield false and the conjunction will yield true. This is in line with earlier observations that a **or** b is true if and only if at least one of a and b is true, and a **and** b is false if and only if at least one of a and b is false.
>
> Yet another informal interpretation relates this property to the earlier discussion of how "implies" always yields **True** when the antecedent is **False**. We might understand ∀ *x*: *SOME_SET* | *x*.*some_property* as a way of saying that "*x* is a member of *SOME_SET*" implies *x*.*some_property*. If *SOME_SET* is empty the antecedent is **False** for every possible *x*, so the implication is true.

5.5 FURTHER READING

The material in this chapter is introductory; as part of a computer science curriculum you will most likely take a course specifically devoted to logic. A standard textbook on the topic, which requires a solid background in general mathematics but defines all the concepts it uses, is

> Elliot Mendelson: *Introduction to Mathematical Logic*, fourth edition, Chapman & Hall/CRC, 1997.

The following are directly intended for computer scientists:

> Zohar Manna and Richard Waldinger: *The Deductive Foundations of Computer Programming*, Addison-Wesley, 1993.
>
> Mordechai Ben-Ari: *Mathematical Logic for Computer Science*, 2nd edition (2001), Springer-Verlag, third corrected printing, 2008.

5.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Logic defines the techniques for reasoning in a precise and rigorous way. It provides the basis of both mathematics and programming.

- *Propositional calculus* defines operations on “boolean variables” that can take either of the values **True** and **False**. The basic “boolean operations” are negation (**not**), non-exclusive disjunction (**or**) and conjunction (**and**).

- Disjunction and conjunction are *dual* of each other: replacing either of them by the other one, negating the operands and negating the result yields a property of the other. This is expressed in particular by “De Morgan’s Laws”.

- Disjunction and conjunction can be generalized to any number of operands through the quantifiers $\exists$ (existential) and $\forall$ (universal) of *predicate calculus*, which apply to the members of a given set.

- For an empty set, regardless of the condition being probed, the existential quantifier yields **False** and the universal quantifier yields **True**.

- Implication can be defined simply in terms of disjunction: *a* **implies** *b* is the same as (**not** *a*) **or** *b*. Implication can be used to deduce new properties from previously proven ones; it does not connote causality. **False** implies **True**.

- In their application to programming, the boolean operations have *semistrict* versions that yield a value even in some cases for which the second operand is not defined. The semistrict variants of **or** and **and** are **or else** and **and then**; **implies** is best defined as semistrict.

New vocabulary

Antecedent **Boolean value** **Boolean expression**
Boolean operator **Boolean variable** **Conjunction**
Consequent **Contradiction** **Disjunction**
Existential quantifier **Implication** **Logic**
Negation **Opposite** **Predicate calculus**
Propositional calculus **Quantifier** **Satisfiable**
Satisfies **Strict** **Stronger**
Tautology **Truth assignment** **Truth table**
Universal quantifier **Weaker**

5-E EXERCISES

5-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

5-E.2 Concept map

1 Devise a conceptual map for the terms of the above vocabulary list.

2 Combine the result with the map obtained for the previous chapters.

← *Exercise "Concept map", 4-E.2, page 69.*

5-E.3 Properties of boolean operators

(Prove your answers.)

1 Is **and** reflexive?

2 Is **or** reflexive?

3 Is equivalence associative?

5-E.4 Twisted logic

"*Whenever the temperature in the city raises above 30 degrees a pollution alert will result, so because the temperature today is only 28 degrees there won't be a pollution alert.*"

1 Informally, what is wrong with this statement?

2 Introducing the appropriate boolean variables, express this statement as a boolean expression.

3 Prove that it is not a tautology. (*Hint*: give a truth assignment that makes it false).

4 Is it a contradiction? (Prove your answer.)

5-E.5 Appropriate warning?

A sign at the entrance to a computer center once read: "*Entrance is prohibited to people who are not authorized or accompanied.*" We accept that there is no ambiguity as to what "authorized" means (someone who has been granted the appropriate credentials), and that "*accompanied*" means "accompanied by an authorized person".

1 Introducing appropriate boolean variables, express this rule as a boolean expression.

2 Explain why the expression does not capture the interdiction that the sign's author probably intended. (*Hint*: Use De Morgan's Laws.)

3 Write the expression reflecting the rule that was most likely intended.

4 Using this expression as a guide, propose an improved rewrite of the English text for the sign.

5-E.6 Inequality

Write the truth table for the inequality operator /=.

← Introduced on page 81.

5-E.7 Associativity and implication

Is the **implies** operator associative? (Prove your answer.)

5-E.8 Signs of strength

We say that a is "stronger than or equal" to b if a implies b. Prove that a **and** b is stronger than or equal to a, and that a is stronger than or equal to a **or** b.

→ Property used by the inheritance mechanism, see "Precondition weakening and postcondition strengthening", page 582.

5-E.9 Police logic

Are Thomson and Thompson, the two policemen in the Tintin extract, justified in accepting Captain Haddock's final explanation?

← Page 87.

5-E.10 Implication and negation

The discussion of implication noted that

← Page 88.

$$\boxed{(a \textbf{ implies } b) \ = \ ((\textbf{not } a) \textbf{ implies } (\textbf{not } b))}$$

is not a tautology. By simplifying this expression — through theorems introduced in this chapter, not truth tables — show under what conditions (for example which truth assignments) it holds.

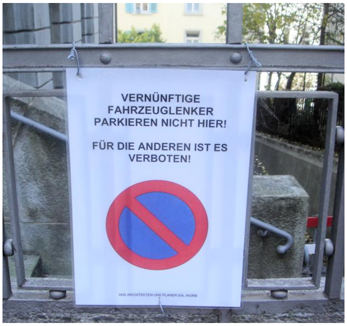
5-E.11 Implication

1 Prove that for any boolean expressions *a* and *b* the following is a tautology:

((*a* **implies** *b*) **and** ((**not** *a*) **implies** *b*)) **implies** *b*

2 The sign shown on the right, spotted in Zurich near the ETH, reads: “*Reasonable drivers don't park here. For others, it's forbidden!*”. Using appropriate boolean variables, including *is_reasonable*, *parks_here*, *parking_prohibited*, express this injunction as one boolean expression.

3 Prove that if this expression is a tautology, and drivers obey parking prohibitions, then *parks_here* is false.



5-E.12 “Exclusive or” as a germ of all things boolean

“Exclusive or”, written **xor**, is a boolean operator of two operands such that *a* **xor** *b* is true if and only if either *a* or *b*, but not both, is true. We may state this property by defining *a* **xor** *b* as

(*a* **or** *b*) **and** (**not** (*a* **and** *b*)) **[X1]**

1 Write the truth table for **xor**.

2 If *a* is a boolean variable, what is the value of *a* **xor** *a*? (Prove your answer from either the definition or the truth table.)

3 Prove that *a* **xor** *b* always has the same value as **not** ($a = b$)

For each of the following boolean expressions (with zero, one or two operands), give another boolean expression that for any value of the operands yields the same value as the given expression, and involves nothing else than the operands, **True** and **xor** (in particular, no other operator); prove your answers.

4 **False**

5 **not** *a*

6 $a = b$

7 *a* **and** *b*

8 *a* **or** *b*

9 *a* **implies** *b*

(The existence of an **xor** equivalent for every boolean operation makes **xor** a particularly interesting operator, holding the germ of all others. Designers of electronic circuits based on boolean logic have taken advantage of this property.)

5-E.13 Properties of “exclusive or”

Based on the above definition [X1] of **xor**, the “exclusive or” operator, prove or disprove the following properties:

1 **xor** is commutative.

2 **xor** is associative.

3 x **xor** $(a$ **xor** $x) = x$ for any a and x.

5-E.14 The blue hats and the red hats

A hundred persons are standing in line, each wearing a hat that is either blue or red. They can each see the hat colors of those ahead in the line, but neither their own nor those of people behind.

Starting with the back of the line — the person who sees all others — they will each, in turn, shout a color name, “**Red!**” or “**Blue!**”, which all can hear.

You are asked to devise a strategy, which they will all adopt beforehand, to maximize the number of people who are guaranteed to shout the color of their own hats — regardless of the distribution of hat colors, about which you know nothing.

Noting the following properties will help:

- A simple strategy is for the first person, the third, the fifth and so on to shout the color of the person immediately ahead (the second, the fourth, the sixth and so on), who then repeats that color, guaranteed to be correct. This gives a lower bound: a good strategy should guarantee *at least* 50 correct results.

- No strategy can guarantee that the first shouter, who does not see his or her color, will be correct. This gives an upper bound: a strategy can guarantee *at most* 99 correct results. We may restate the problem as asking how close we can get to this theoretical maximum.

- There is nothing probabilistic about the problem. Even if we had some information about the distribution of colors, it would not help since the strategy must maximize the number of answers *guaranteed* correct, not some probability of correct answers.

Hint: the preceding exercise helps.

5-E.15 Truth tables with undefinedness: semistrict boolean operators

Assume an extension of propositional calculus with three values instead of two: **True**, **False**, and *Undefined*. For example *l*.*i_th* (*i*) has value *Undefined* if *i* does not satisfy the precondition of *i_th*.

Considering that each of *a*, *b* and the resulting expression may take on any of these three values, write the truth tables (each with nine entries) for:

1 *a* **or else** *b*

2 *a* **and then** *b*

5-E.16 Truth tables with undefinedness: ordinary boolean operators

As in the preceding exercise, assume that boolean values include **True**, **False**, and *Undefined*. Explaining the reason for your answer, propose truth tables for:

1 *a* **or** *b*

2 *a* **and** *b*

(Here more than one set of truth tables may make sense, so what's interesting is how you justify your proposed solution.)

6

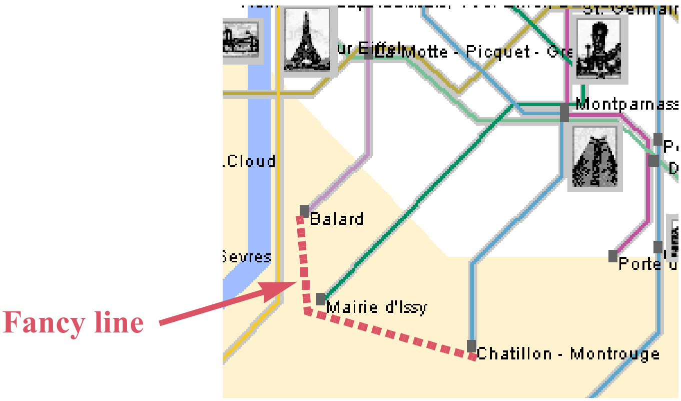
Creating objects and executing systems

After our excursion into the mathematical foundations we are back to the techniques of programming.

In earlier examples we have used names such as *Paris*, and *Route1* to access objects that someone else creates for us — mysteriously so far. It is time to see how we can, in our own programs, create our own objects ourselves. Object creation, the central topic of this chapter, is an interesting mechanism with several ramifications. It will lead us to the overall picture of system execution: how programs start, run and terminate.

To learn about these techniques, we will create a fictitious line of the metro, *fancy_line*, connecting some real stations. Contrary to our previous examples such as *Line8*, the line *fancy_line* is not predefined; we have to build it ourselves. This process will require that we create other objects, for example to represent stops on the line.

Three lines of the Paris metro terminate in the same general area on the south side, but too far to walk, and people living there would *really* like a connection other than through the city center and back. The purpose of *fancy_line* is to console them, if not in the real city at least in a virtual one:



6.1 OVERALL SETUP

Our system for this chapter is called *creation*. Open it now, using the same techniques as in previous chapters. Bring up the class of interest for the present discussion, *LINE_BUILDING*, which initially looks just like this:

In directory 06_creation.

```
class LINE_BUILDING inherit
    TOURISM
feature
        build_a_line
                -- Build an imaginary line and highlight it on the map.
            do
                Paris.display
                -- "Create new line, fill in its stations and add it to Paris"
                -- "Highlight the new line on the map"
            end
end
```

The line -- "Create new line, fill in its stations and add it to *Paris*", and the line that follows it, start with two hyphens and hence are comments, but of a special kind known as **pseudocode**, meaning that they stand for actual program text (also known as "code") that we intend to fill in later as we develop the program:

← *Page 39.*

Definition: Pseudocode

Pseudocode is informal text standing for program elements to be added later.

> Indispensable note based on my experience of teaching non-native English speakers and drawing blank stares when I utter the word "*pseudocode*": the "*p*" and the first "*e*" are **not** pronounced. Say "*SUDOKU*", only replacing the final "*U*" by "*ODE*". Do not blame me, I did not design the phonetics of English.

It is useful to rely on pseudocode (rather than a margin note such as "*The part you will fill in*") whenever we need to give an informal English description of program elements that we are not *yet* expressing as actual program text — either because we cannot, or because this would force us to dive into a particular corner of the program and lose track of the bigger picture.

← As in the example on page 16.

The process of progressively replacing pseudocode elements by actual code is called **refinement**. This technique will become ever more useful as we start writing more complex software. It is part of *top-down design* discussed in a later chapter.

→ "Bottom-up and top-down reasoning", 8.1, page 211.

Pseudocode will use the convention illustrated by the example:

Touch of Style: Highlighting pseudocode

Write pseudocode elements as comments, with their text enclosed in quotes and (if color is available) appearing in red.

Any actual program elements cited in such pseudocode, such as *Paris*, appear in their usual *blue* to signal that there is nothing "pseudo" about them.

By using comments for pseudocode you ensure that your program, even if not complete, is syntactically correct; it may not be interesting yet to execute it, but you can compile it, so that the compiler will find any errors that you let slip through, such as incorrect use of types. It is a basic methodological rule that **programs should be compilable at all stages of their development**. *Routines*, introduced in a later chapter, will provide a complementary technique (generally superior to pseudocode) towards this goal.

→ *"Functional abstraction", 8.7, page 220.*

Marking pseudocode comments in a special way (quotes and, in printed text, color) reminds you that they are not just ordinary comments annotating existing code, but placeholders for code that you must add at some point.

6.2 ENTITIES AND OBJECTS

Our class needs a feature (a query) representing the line to be built. We call it *fancy_line*. This is also an opportunity to start the refinement process, by turning part of the pseudocode (some of the first line and the entire second line) into actual code and making the remainder more precise:

```
class LINE_BUILDING inherit
    TOURISM
feature
        build_a_line
                -- Build an imaginary line and highlight it on the map.
            do
                Paris.display
                -- "Create fancy_line and fill in its stations"
                Paris.put_line (fancy_line)
                fancy_line.highlight
            end

        fancy_line: LINE
                -- An imaginary (but desirable) line of the Paris Metro
end
```

We are down to one line of pseudocode. The instruction after that line refines the original pseudocode text "add it to *Paris*"; it uses a command *put_line*, available on "city" objects to add a line, with its stations, to a city. The following instruction uses the command *highlight* to refine the second line of the original pseudocode.

Once the procedure *build_a_line* has been executed, *fancy_line* will denote an instance of class *LINE*, representing a metro line.

Identifiers may denote many things: they can be names of classes, like *STATION*, or of features, like *i_th*. An identifier such as *fancy_line* whose role is to denote run-time values, such as objects, is called an **entity**.

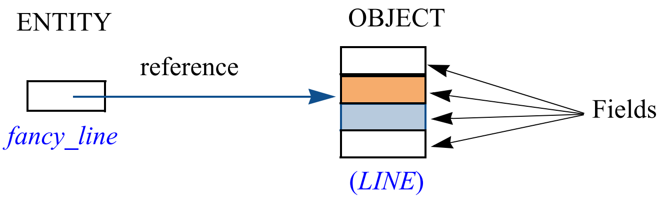
If you have programmed before you may be familiar with another important term: *variable,* denoting entities whose value may change. "Entity" is more general since some of our entities must have constant values. A later chapter studies variables in detail.

→ *Chapter 9.*

In this case the entity *fancy_line* is the name of a feature, but we will encounter other kinds of entity.

If, at some instant of the execution ("run time"), the value of an entity represents an object, we say that the entity is **attached** to the object.

The following picture, depicting a run-time situation, helps visualize the notion of entity and attached run-time object:



During execution: entity and attached object

This shows the relationship:

- The entity is a name in the program which at run time will denote, through a "reference", an object in memory. The notion of reference expresses the association and will be defined more precisely in a later chapter.
- The object, as defined earlier, is a collection of data; it is made more precisely, as suggested by the picture, of a set of **fields** each holding a data unit (for example an integer or boolean value). The data that our programs manipulate during execution is *entirely* made of such objects, each with its fields. The fields of a *STATION* object might, for example, include the station's coordinates on the map, the name of the station etc.

→ *"Reference assignment", 9.5, page 252.*

← *Definition of "object": page 29.*

Note the conventions in diagrams such as the above giving a snapshot of the object structure, or part of it, during execution:

- An object is represented as a **rectangle**, with its fields represented as sub-rectangles.
- Next to each object, usually below, you will see in parentheses the name of its generating class — the class of which it is an instance, here (*LINE*).

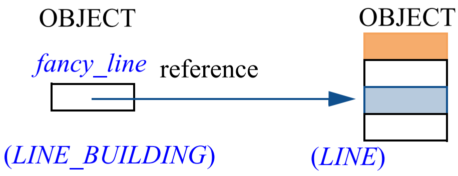
← *Definition of "generating class": page 50.*

6.3 VOID REFERENCES

In considering the execution of *build_a_line* and the value of *fancy_line*, we must pay particular attention to references and their relation to objects.

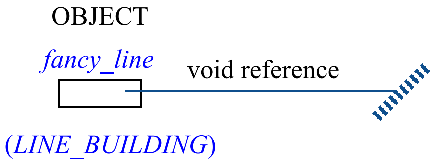
The initial state of a reference

Assume we have an instance of *LINE_BUILDING*. You might think that because the class declares a query *fancy_line* of type *LINE*, we may always assume that its instance contains a reference to an instance of *LINE* as suggested above:



Line entity and attached TOURISM object

Not so. We do have one object, the one on the left in the figure, an instance of *LINE_BUILDING*, with only one field corresponding to the query *fancy_line*. Let's assume this object has just been created; this is the result of a "creation instruction", which we will shortly see how to write. The instruction only gives us the *LINE_BUILDING* object. If you need any other, your program will have to create it explicitly. So the true state of program execution after creation of an instance of *LINE_BUILDING* looks like this:



Object structure at the beginning of execution

The field for *fancy_line* contains a reference. But because no instruction has been executed yet to create other objects, that reference is **void**, meaning that it is not attached to any object; the figure shows the convention for void references, reminiscent of the "grounded" symbol in electricity.

This is one of the two possible states for a reference:

Definition: States of a reference

At any time during execution, the value of an entity denoting a reference is one of:

- **Attached** to a certain object.
- **Void**.

The predefined feature **Void** denotes a void reference. So at any time during execution, if *x* denotes a reference, the condition

```
x = Void
```

has value *True* if and only if the value of *x* is a void reference, and

```
x /= Void
```

← */= is inequality; see page 81.*

if and only if it is attached to an object.

The trouble with void references

The basic mechanism of computation was introduced as *feature call*, of the form *x*.*f* or, with arguments, *x*.*f* (…). It applies feature *f* to the object to which *x* is attached. But now with void references we have the possibility that, at some time during execution, if *x* = **Void** holds, the reference that *x* denotes will not be attached to any object. The feature call is erroneous in that case.

← *"Dissecting the program", page 23.*

To see the effect of such a bug, try to execute the system in the following form:

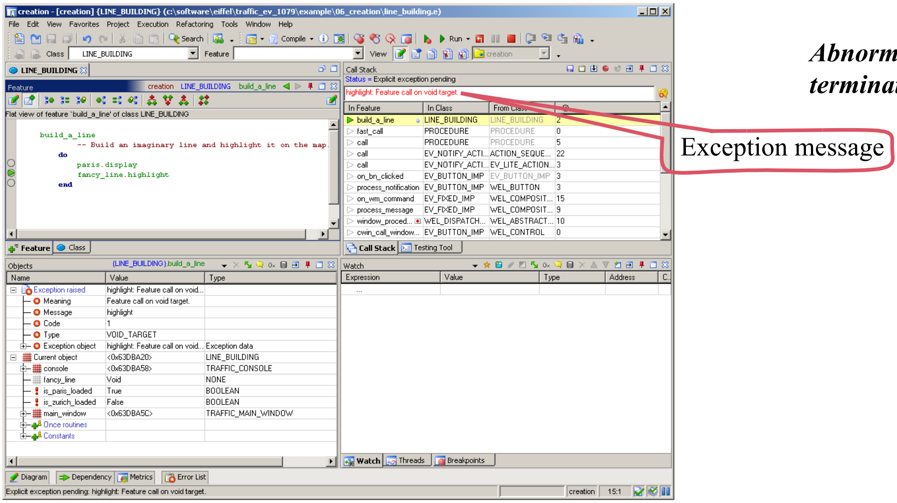
```
class LINE_BUILDING inherit
    TOURISM
feature
        build_a_line
                -- Build an imaginary line and highlight it on the map.
            do
                Paris.display
--              Paris.put_line (fancy_line)
                    -- The next line should have been replaced by code!
                -- "Create fancy_line and fill in its stations"
                fancy_line.highlight
            end
        fancy_line: LINE
end
```

As shown, you should temporarily "comment out" (turn into a comment) the line after *Paris*.*put_line* (*fancy_line*); the reason is that it would already produce an error, but not the one we want.

> To comment out a sequence of line in EiffelStudio, you may type two hyphens "--" at the beginning of each line, or simply select them all and type Control-K. To remove them ("*uncomment*" the line), use Control-Shift-K.

After the initial call (*Paris*.*display*) execution stops abruptly, displaying a message in EiffelStudio stating that an **exception** has occurred:

Abnormal termination



creation - [creation] {LINE_BUILDING} (c:\software\eiffel\traffic_ev_1079\example\06_creation\line_building.e)

Flat view of feature `build_a_line' of class LINE_BUILDING

```
build_a_line
        -- Build an imaginary line and highlight it on the map.
    do
        paris.display
        fancy_line.highlight
    end
```

Call Stack

Status = Explicit exception pending

highlight: Feature call on void target.

In Feature	In Class	From Class	@
build_a_line	LINE_BUILDING	LINE_BUILDING	2
fast_call	PROCEDURE	PROCEDURE	0
call	PROCEDURE	PROCEDURE	5
call	EV_NOTIFY_ACTI...	ACTION_SEQUE...	22
call	EV_NOTIFY_ACTI...	EV_LITE_ACTION...	3
on_bn_clicked	EV_BUTTON_IMP	EV_BUTTON_IMP	3
process_notification	EV_BUTTON_IMP	WEL_BUTTON	3
on_wm_command	EV_FIXED_IMP	WEL_COMPOSIT...	15
process_message	EV_FIXED_IMP	WEL_COMPOSIT...	9
window_proced...	WEL_DISPATCH...	WEL_ABSTRACT...	10
cwin_call_window...	EV_BUTTON_IMP	WEL_CONTROL	0

Objects {LINE_BUILDING}.build_a_line

Name	Value	Type
Exception raised	highlight: Feature call on void...	
Meaning	Feature call on void target.	
Message	highlight	
Code	1	
Type	VOID_TARGET	
Exception object	highlight: Feature call on void...	Exception data
Current object	<0x63DBA20>	LINE_BUILDING
console	<0x63DBA58>	TRAFFIC_CONSOLE
fancy_line	Void	NONE
is_paris_loaded	True	BOOLEAN
is_zurich_loaded	False	BOOLEAN
main_window	<0x63DBA5C>	TRAFFIC_MAIN_WINDOW
Once routines		
Constants		

Watch

Expression	Value	Type	Address	C..
...				

Explicit exception pending: highlight: Feature call on void target.

Exception message

What happened is a **void call**, or attempt to call a feature on a void target, meaning an object that does not exist. Such an attempt will never succeed:

Touch of Semantics: Attached Target Principle

A call of the form *x.f* (…) can only execute properly if, at the time of execution, *x* is attached (not void).

→ *Known as "qualified"; see "Definitions: Qualified and unqualified call", page 134.*

A void call causes an *exception*: an abnormal event occurring during program execution. Here the result of the exception is to cause the entire program execution to fail. The EiffelStudio message indicates the name of the exception in this case: "Feature call on void reference" (the full name of void calls).

As we will see when we study exceptions, it is possible to avoid failure by providing *exception handling* code which will attempt to recover. But this is only a technique of last resort; prevention is better than cure. The general rule, whenever your program includes a call *x.f* (…) and there is a risk that *x* might be void, is to protect the call so that it will only be executed if *x* is attached. In library code you will see many tests such as **if** *x* /= **Void then** …, and preconditions of the form *x* /= **Void** where *x* is an argument. It is your responsibility to ensure that the targets of all calls are attached in every execution.

→ *"An introduction to exception handling", 7.10, page 200.*

→ *Relying on Conditional (**if**) instructions studied in the next chapter.*

This situation is improving. The Appendix to this chapter describes recent evolutions that will entirely remove the risk of void calls.

If you now uncomment the line *Paris.put_line (fancy_line)*, recompile and execute, you will see the program fail in another way: it violates the precondition of *put_line*, which states that its argument must be non-void. (The reason for commenting it out was that we wanted to see a void-call exception first.)

Not every declaration should create an object

To avoid the void call and the exception in the last example, we may change the creation procedure *build_a_line* so that before the call *fancy_line*.*highlight* it will have created an object and attached it to *fancy_line*. We will do this shortly. You may, however, question the behavior. Why have void references at all and hence create the resulting risk of void calls at run time? Should not we be able to assume that a declaration such as

```
fancy_line: LINE
```

will at run time have the effect of creating an object — an instance of *LINE* — and attach it to *fancy_line*?

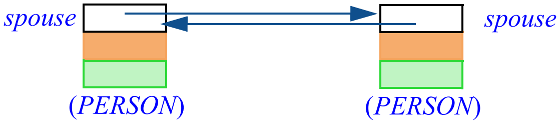
The answer is no. Several reasons justify the convention that references are initialized to void, and that you get objects only by creating them explicitly through your program.

The basic reason is that some objects simply do not exist. This is also true in the non-software world: a person *may* have a spouse, but not everyone is married. Software that models that world should retain such properties: in a class *PERSON*, appearing for example in tax management software, you may want to include a feature

```
spouse: PERSON
```

for which the possibility of a void reference is useful: it will represent the case of an unmarried person. Even if we assumed everyone is married, it would still make no sense to create an object for *spouse* every time we create an object of type *PERSON*: then it too would have its *spouse* reference, for which we would have to create another instance of the class, starting an infinite chain. So the reasonable solution is to initialize the field to a void reference, and let the program create an object when appropriate.

Consider the example a little more in depth. When a person does have a *spouse*, there is a constraint: the spouse is also married, and has, as a spouse, the original person. A picture shows this better than words:



Monogamy

and a formula says it even better than a picture: the invariant of class *PERSON* should have a clause that reads

monogamy: (*spouse* /= **Void**) **implies** (*spouse*.*spouse* = **Current**)

Current, used in relation to an object, denotes the object itself. The clause says that if a person has a spouse, then that spouse's spouse is the original person.

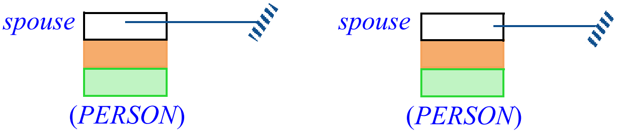
→ **Current** *will be seen more formally in "Definition: Current object", page 132.*

> **Current** is never void since it denotes an object. So from this invariant clause we may deduce another: (*spouse* /= **Void**) **implies** (*spouse*.*spouse* /= **Void**): if you are a married person, your spouse is married too. Do not dismiss the benefit of expressing such seeming banalities: software development involves clarifying the intuitive knowledge that we may have about a problem domain, and then formalizing it using the tools of logic, for example in class invariants.

> Another observation on the above invariant clause: if you have carefully followed the discussion of semistrict boolean operators, you will have noticed that this clause requires the semistrict version of **implies**, since the second operand would otherwise not be defined for a void *spouse*.

← *"Semistrict implication", page 94.*

This shows further why we shouldn't jump to create an object every time there is an entity declaration. Both objects on the preceding figure should probably start their lives celibate, with void *spouse* references:



Double celibacy

Later on, some instructions — for example a call to a command *marry* — will attach the *spouse* reference of each object to the other, yielding the state shown in the earlier figure. Such *reattachment* instructions do not create any new objects; they simply attach references to existing objects. We will study them in a later chapter.

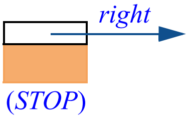
→ *Chapter 9.*

The role of void references

Consider a reference appearing in a field of an object, such as the *spouse* field of a person object. If itself attached to an object, it indicates the presence of certain information, represented by that object. If it is void, it indicates that such information does not exist. This is particularly useful when we use references to **link** objects in a more complex structure. Many interesting data structures, such as *linked lists* which will play a prominent role in our discussions, rely on this concept of linking.

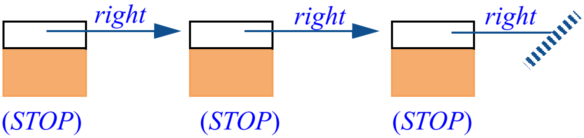
→ *"Using references for building linked structures", page 256 and "Linked lists", 13.7, page 400.*

Here is a simple example from Traffic. We may decide to represent a metro line (any instance of class *LINE*) by one or more instances of a class *STOP*, each representing a stop on the line. One possible technique (we will see many others) is to have, in every instance of *STOP*, a field *right* indicating the next stop on the route. So an instance of *STOP* will look like this:



A stop (provisional)

where the solid part represents fields providing other information on the stop. Then a full line will be a set of such objects, each but the last linked to the next one by a *right* reference:



A linked line

Note how the last object uses a void *right* reference to indicate that there is no *right* object in this case. Terminating such structures is one of the principal uses for void references.

The name *right*, for the field containing a reference to the next object, comes from the standard way of picturing such lists, with list items appearing left to right as above.

Calls in expressions: overcoming your fear of void

Before we come back to creation —which makes references non-void — we must look at a common scheme for using references in expressions and avoiding any fear of void call.

Because a feature call is only defined for a non-void target, you may wonder how to express conditions so that they are always defined even if they involve a call. A conditional instruction (after the **if**) or a class invariant might need a condition of the form

```
fancy_line.count >= 6
```

stating — as an example — that a line has at least six stations. But this is only defined if *fancy_line* is attached. If you do not know for sure, you need a way to state the condition that, in informal terms, holds if and only if

"*fancy_line* is defined, *and then* it has at least 6 stations"

You know the solution (I hope the use of "*and then*" rang a bell): *semistrict* operators are precisely designed for conditions of which one part makes sense only if the other has value **True** (with **and then**), or **False** (with **or else**).

← *"Semistrict boolean operators", 5.3, page 89.*

You may write the example condition correctly as

```
(fancy_line /= Void) and then (fancy_line.count >= 6)
```

This ensures that the condition is always defined:

- If *fancy_line* is void, the result is *False*; the evaluation of the expression is guaranteed not to use the second operand, which would cause an exception.
- If *fancy_line* is attached, the second operand is defined and will be evaluated, yielding the result of the expression as a whole.

A variant of this pattern uses **implies**, which we have defined as a nonstrict operator (along with **and then** and **or else**). A condition of the form

← *"Semistrict implication", page 94.*

```
(fancy_line /= Void) implies (fancy_line.count >= 6)
```

expresses a slightly different property:

"*If fancy_line* is defined, *then* it has at least 6 stations"

implying "if it is not defined I don't care, so it is fine too": the condition should yield **True** in that case, where the **and then** form yielded **False**. Such a pattern is frequently useful in class invariants; it figured in the clause that we included in our *PERSON* class:

```
monogamy: (spouse /= Void) implies (spouse.spouse = Current)
```

This stated that if you are married your spouse's spouse is yourself. But if you are *not* married the condition should also yield **True**; otherwise unmarried persons would violate the invariant, for no good reason. The **implies** operator achieves this, since **False** implies **True**. Semistrictness guarantees that no improper evaluation will occur in this case.

← *"Theorem: Implication Principle", page 84.*

6.4 CREATING SIMPLE OBJECTS

I hope you have not lost track of our goal in this chapter, which is to create our *fancy_line* as pictured at the very beginning, with three stations. We are almost there, but first we need to create the objects representing the stops on the line.

These auxiliary objects will be instances of the class *STOP* just mentioned. By the way, can you see why we need such a class?

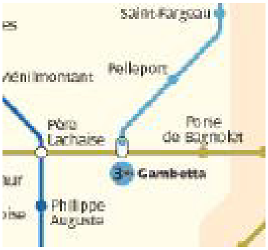
Quiz time!
The stops of a metro line are more than metro stations

To model a metro line, why do we need a new class *STOP* and not just instances of the class *STATION*?

The last figure gives a clue. A stop on a line is associated with a station, but it is a different object because it represents the station *as belonging to the line*.

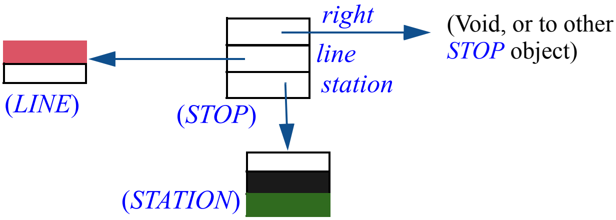
A query such as “What is the next station?” is not a feature of the station; it is a feature of the station as belonging to the line. The reason is that, in the words of our little requirements document, “*Some stations belong to two or more lines; they are called ‘exchanges’*”. On the following figure, the next (*right*) station for Gambetta (going as usual from South to North) depends on which of its two lines you take.

← *“Touch of Paris: Welcome to the Metro”, page 52.*



More than one “next” station

A *STOP* object will be very simple. It contains a reference to a station, another to the line to which the stop belongs, and a reference to the next object:



A stop (final)

It makes no sense to have a stop without a station and a line, so we will require *station* and *line* always to be attached (non-void); the class invariant should state this. The *right* reference may be void, to indicate that a stop is the last in its line.

We do not worry here with creating *STATION* objects, since the only ones we need come to us predefined from *TOURISM* through features called *Station_X* for the appropriate *X*: *Station_Montrouge*, *Station_Issy* and others. So we will learn about creation by creating instances of *STOP*.

A first version of class *STOP*, called *SIMPLE_STOP*, has the following interface (bring it up under EiffelStudio):

```
class SIMPLE_STOP feature

    station: STATION
            -- Station which this stop represents.

    line: LINE
            -- Line to which stop belongs.

    right: SIMPLE_STOP
            -- Next stop on same line.

    set_station_and_line (s: STATION; l: LINE)
            -- Associate this stop with s and l.
        require
            station_exists: s /= Void
            line_exists: l /= Void
        ensure
            station_set: station = s
            line_set: line = l

    link (s: SIMPLE_STOP)
            -- Make s the next stop on the line.
        ensure
            right_set: right = s

-- Missing invariant clauses: station /= Void and line /= Void; see discussion.
end
```

The query *station* yields the associated station and *line* yields the line to which the stop belongs; *right* yields the next stop. Associated commands are *set_station_and_line* to give the stop (in one sweep) both a station and line, and *link* to link it to another stop on the same line. Such commands, having as their main purpose to set the value of associated queries (although they may do more), are called *setters*.

→*More in "Setters and getters", page 248.*

Here is how to create an instance of this class. Assume that (along with *fancy_line*: *LINE*) we have declared

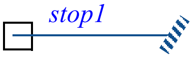
```
stop1: SIMPLE_STOP
```

Then in procedure *build_a_line* we may create a stop:

```
build_a_line
        -- Build an imaginary line and highlight it on the map.
    do
        Paris.display
        -- "Create fancy_line"
        Paris.put_line (fancy_line)
        create stop1
        -- "Create more stops and finish building fancy_line"
        fancy_line.highlight
    end
```

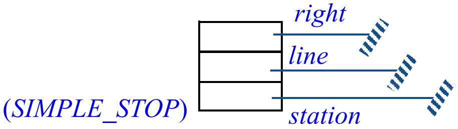
The remaining pseudocode has been refined into two parts: initially, create the line; at the end, create more stops and link them.

The instruction **create** *stop1* is a **creation instruction**. This is the basic operation to produce objects at run time. Its effect is exactly as the keyword **create** suggests: create an object, and attach the listed entity, here *stop1*, to that new object. In pictures: starting from a state in which *stop1* is void



Before creation instruction

executing **create** *stop1* attaches it to an object created for this purpose:



After creation instruction

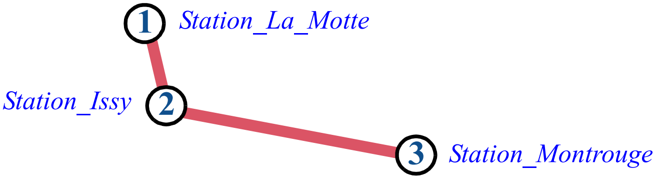
The **create** instruction does not need to specify the type of object to be created, since every entity such as *stop1* is declared with a type; here the declaration was *stop1*: *SIMPLE_STOP*. The type of the object to be created is the type declared for the corresponding entity, here *SIMPLE_STOP*.

As a consequence of the earlier discussion, all reference fields of the new object are set to **Void**. We can attach them to actual objects using the commands *set_station_and_line* and *link*. This enables us to build all the stops of *fancy_line* (the *LINE* object itself will follow). We declare the three stops:

```
stop1, stop2, stop3: SIMPLE_STOP
```

Note the syntax enabling you to declare several entities of the same type together, rather than writing a declaration for each. You will just separate the entities by commas and write the type once after the colon.

The numbers correspond to the order on our line:



Three stops on a line

This allows us to write the next version of *build_a_line*:

```
build_a_line
        -- Build an imaginary line and highlight it on the map.
    do
        Paris.display
        -- "Create fancy_line"
        Paris.put_line (fancy_line)
            -- Create the stops and associate each to its station:
        create stop1
        stop1.set_station_and_line (Station_Montrouge, fancy_line)
        create stop2
        stop2.set_station_and_line (Station_Issy, fancy_line)
        create stop3
        stop3.set_station_and_line (Station_Balard, fancy_line)
            -- Link each applicable stop to the next:
        stop1.link (stop2)
        stop2.link (stop3)
        fancy_line.highlight
        end
```

Note how pseudocode progressively shrinks as we add instructions — real code, not "pseudo" — to realize its intent. In the end we must have removed all of it.

The two highlighted calls to *link* chain the first stop to the second and the second to the third. The third stop is not chained to anything; its *right* reference, set to void on creation, will remain void. This is what we want since it represents the last stop on the line.

The calls to *set_station_and_line* must satisfy the precondition of this feature, which requires its arguments to be attached:

← *"Touch of Methodology: Precondition Principle", page 64.*

- *Station_Montrouge* and other stations come from class *TOURISM*, which indeed takes care of creating the necessary objects.
- *fancy_line* will be attached if the remaining pseudocode element does its advertised job of creating an object. That element will be refined below into another **create** instruction.

6.5 CREATION PROCEDURES

Procedure *build_a_line* uses the simplest form of creation:

```
create stop                                                        [2]
```

for *stop* of type *SIMPLE_STOP*. This does the job but deserves an improvement. As the last version of the procedure indicates, the typical scheme for creating a stop associated with a station *existing_station* is in fact

```
create stop                                                        [3]
stop.set_station_and_line (existing_station, existing_line)
```

which requires calling a feature, immediately after the creation instruction, to link the new object to a station and to a line. The object resulting from the first instruction is useless because, as noted, it makes no sense to have a "stop" object without an associated station and line. We would like to express this through an invariant

```
invariant
    station_exists: station /= Void
    line_exists: line /= Void
```

but then the class becomes incorrect since every instance must satisfy the invariant on creation, which will not be the case after a plain **create** in [2].

← *"Touch of Methodology: Class Invariant Principle", page 68.*

So we have two separate reasons leading us to merge the two instructions above, the creation and the call to *set_station_and_line*, into one:

- A reason of convenience: with the class as it stands, any client needing to create a stop must use both instructions; forgetting the second one will result in incorrect software and run-time failures. It is a general rule of software design that we should avoid producing elements that require specific prescriptions for use — "*When you do A, never forget to do B as well*!" — as it is all too easy for client programmers to miss the instructions. (Do you always read the manuals of the devices you use?) Better provide an operation that does everything, removing the need to learn a tricky interface.
- A reason of correctness: we would like to ensure that instances of the class, straight from their creation, are consistent — here, have a station and line.

To address both concerns, we may declare the class with one or more **creation procedures**. A creation procedure is a command that clients *must* call whenever they create an instance of the class, ensuring that the instance is properly initialized and, in particular, satisfies the invariant.

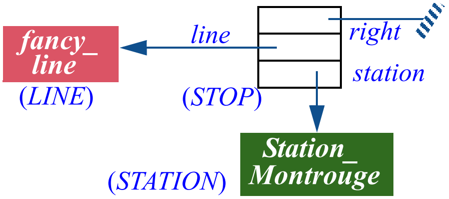
With a creation procedure, here *set_station_and_line*, and the stops now declared as

```
stop1, stop2, stop3: STOP
```

(rather than *SIMPLE_STOP* as before), the creation instruction as executed by clients is no longer just **create** *stop1* [2] but

```
create stop1.set_station_and_line (Station_Montrouge, fancy_line)          [4]
```

which has the effect achieved earlier by two separate instructions [3]:



After creation instruction using a creation procedure

The only difference between *STOP* and its predecessor is that *STOP* has the desired invariant *station* /= **Void** and declares *set_station_and_line* as a creation procedure. Here is how the class interface will look; other than the class name, only the highlighted parts have changed:

The class as it appears in Traffic, under the name TRAFFIC_ STOP per usual conventions, has all these features and a few more which you will see under EiffelStudio.

```
class STOP create
    set_station_and_line
feature
    station: STATION
            -- Station which this stop represents.
    right: STOP
            -- Next stop on associated line.
    set_station_and_line (s: STATION; l: LINE)
            -- Associate this stop with s and l.
        require
            station_exists: s /= Void
            line_exists: l /= Void
        ensure
            station_set: station = s
            line_set: line = l
    link (s: STOP)
            -- Make s the next stop on associated line.
        ensure
            right_set: right = s
invariant
        station_exists: station /= Void
        line_exists: line /= Void
end
```

Same as before, now also serves as creation procedure

At the top of the class interface we have a new clause

```
create
    set_station_and_line
```

using again the keyword **create**, and listing one of the commands of the class, *set_station_and_line*. This tells the client programmer that the class admits *set_station_and_line* as a creation procedure. This clause lists one creation procedure; it could also list none, or several (since there may be more than one way to initialize a newly created object).

The consequence of including such a clause in the interface of the class is that a client may no longer create an object using the basic form of the creation instruction, **create** *stop* [2]; because the class specifies creation procedures, you *must* use one of them, through form [4].

This rule enables the author of a class to force proper initialization of all instances that clients will create. It is closely connected with the notion of invariant: the requirement is that every object will satisfy, immediately after creation, the desired invariant; in our example the invariant is

```
station_exists: station /= Void
line_exists: line /= Void
```

which is in turn ensured by the precondition of *set_station_and_line*. This is a general principle:

Touch of Methodology: Creation Principle

If a class has a non-trivial invariant, it must list one or more creation procedures, whose purpose is to ensure that every instance, upon execution of a creation instruction, will satisfy the invariant.

"Non-trivial invariant" means any invariant other than **True** (which is usually omitted) or any property that would be ensured by letting all the fields take the default values ensured by the initialization rules (zero for numbers, **False** for booleans, **Void** for references).

Even in the absence of a strong invariant, it may be useful to provide creation procedures to enable clients to combine creation with initialization. A class *POINT* describing points in a two-dimensional space may provide creation procedures *make_cartesian* and *make_polar*, each with two arguments denoting coordinates, enabling clients to create points identified by their cartesian or polar coordinates.

→ The class does deserve an invariant; see the exercise "Invariant for points", 6-E.3, page 138.

In some cases — *POINT* is an example — you may want to allow both forms, [2] and [4]. The technique then is to use

```
class POINT create
    default_create, make_cartesian, make_polar
feature
    …
end
```

where *default_create* is the name of a feature (inherited by all classes from a common parent) with no arguments, which by default does nothing. To use this procedure you would normally write

→ *The common parent is the library class ANY, as we will see in "Overall inheritance structure", 16.10, page 586.*

```
create your_point.default_create
```

but this can be abbreviated into form [2], here

```
create your_point
```

which the **create** clause makes valid along with the other two forms

```
create your_point.make_cartesian (x, y)
create your_point.make_polar (r, t)
```

The general abbreviation rule is that:

- If a class has no **create** clause, it is as if it had one of the form
  ```
  create
      default_create
  ```
  listing *default_create* as the single creation procedure.
- Correspondingly, a creation instruction of the simplified form [2], **create** *x* with no creation procedure, is an abbreviation for the explicit form
  ```
  create x.default_create.
  ```

So conceptually you may always consider that a creation procedure is involved.

To complete *build_a_line*, we only need to refine the last remaining pseudocode line: -- "Create *fancy_line*". It is just another creation instruction:

← *In the latest version, page 121.*

```
create fancy_line.make_metro ("FANCY")
```

using *make_metro*, one of the creation procedures of class *LINE*, which creates the line as a metro line (rather than a bus line, a tram line etc.), taking as argument the name of the line, a string.

Since all this is available as part of the predefined examples, it is a good idea to go and read its final form:

Program reading time!

Creating and initializing a line

Look up the text of *build_a_line* in class *LINE_BUILDING* and make sure you understand all that it does.

As a consequence of the preceding discussion, it is easy to remember what you must do to create an object:

Creating an instance of a class

- If the class has no **create** clause, use the basic form, **create** *x* [2].
- If the class has a **create** clause listing one or more creation procedures, use

  create *x*.*make* (...) -- [4]

  where *make* is one of the creation procedures, and "(...)" stands for appropriate arguments for *make*, if any; there must be the right number of arguments, with the right types, satisfying the precondition of *make* if any.

6.6 CORRECTNESS OF A CREATION INSTRUCTION

For every instruction that we study, we must know precisely, in line with the principles of Design by Contract sketched in earlier chapters:

- How to use the instruction correctly: its *precondition*.
- What we are getting in return: its *postcondition*.

In addition, classes (and, as we will see, loops) have *invariants* describing properties that some operations will maintain.

Together, these contract properties define the **correctness** of any programming mechanism.

Here is the rule for the creation mechanism:

***Touch of Methodology*:**
Creation Instruction Correctness Rule

For a creation instruction to be correct, the following property (precondition) must hold *before* any execution of the instruction:

1 The precondition of its creation procedure.

The following properties (postconditions) will hold *after* a creation instruction with target *x* of type *C*:

2 *x* /= **Void**.

3 The postcondition of the creation procedure, if any.

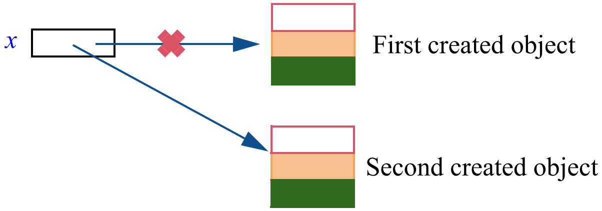
4 The invariant of *C*, as applied to the object attached to *x*.

The form without a creation procedure, **create** *x*, trivially satisfies clauses 1 and 3 since there is no applicable precondition or postcondition.

The precondition rule (clause 1) does not require *x* to be void. It is indeed not a mistake to create two objects successively for the same target *x*:

```
create x
    -- Here as a result x is not void (see clause 2)
create x
```

even though this specific example is wasteful since the object created by the first instruction will be forgotten immediately afterwards:



Creating two objects in a row

The second creation instruction reattaches the reference *x* to the second object, so that the first object is now useless. (We will see shortly what happens to such orphaned objects.)

Although two successive creation instructions of the exact form shown make no sense, variants of this scheme can be useful. For example there could be other instructions between the two **create** *x*, doing something interesting with the first object. Or if a creation procedure is involved, as in **create** *x*.*make* (...), it may record the first object somewhere.

Clauses 2 to 4 define the effect of executing a creation instruction:

- Whether or not *x* was void before the creation instruction, it will not be void afterwards (clause 2) since the instruction attaches it to an object.
- If there is a creation procedure, its postcondition will hold for the newly created object (clause 3).
- In addition, that object will satisfy the **class invariant** (clause 4). Already stated in the Invariant Principle, this requirement is essential for any creation instruction: it ensures that any object, when it starts out in life, satisfies the consistency condition that its class imposes on all instances, as expressed by the invariant.

← *"Touch of Methodology: Class Invariant Principle", page 68.*

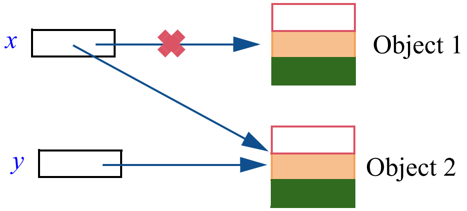
If the default initializations do not establish the invariant, it is then the duty of creation procedures to correct the situation by producing an initial state that satisfies that invariant.

6.7 MEMORY MANAGEMENT AND GARBAGE COLLECTION

In the situation pictured in the last figure, a reference that was attached to an object (the "First created object") gets reattached to another. What, you may wonder, happens to the first object? While the particular example (two successive **create** *x* for the same *x*) was unrealistic, useful reference reattachments are common and may raise the same question; in a while we will study *reference* assignments such as *x* := *y*, whose effect we may picture as:

→ *Chapter 9.*

Reference reattachment



The instruction reattaches *x* to the object to which *y* is attached. What happens to "Object 1"? More generally, we must ask ourselves, to complement this chapter's discussion of how to *create* objects, whether and how objects can ever be *deleted*.

Since there could be other references attached to "First created object" or "Object 1", the question of real interest is: when a reference to an object is removed, as in these examples, what happens to the object *if there remains no other reference attached to it*? There is no trivial answer since finding out whether some other object retains a reference to a given object, such as "Object 1" in the above figure, requires a deep understanding of the entire program and its possible executions. Three approaches are possible: *casual*; *manual*; *automatic*.

The casual approach simply ignores the problem, letting unused objects linger. It can cause memory waste to grow uncontrolled. Such *memory leaks* are unacceptable for continuously running systems such as those on embedded devices — a memory leak on your cell phone would eventually bring it to a halt — and, more generally, any system which creates and forgets many objects. The casual approach is inadequate for any non-trivial application.

The manual approach provides programmers with explicit facilities to return objects to the operating system. A C++ programmer may, for example, precede the reattachment of *x* (through **create** *x* or *x* := *y*) with the routine call *free* (*x*), which signals that the object attached to *x* is no longer needed so that the operating system can reuse its memory area for any future object creation.

The automatic approach frees programmers from *free* by entrusting a mechanism, the **garbage collector** ("GC" for short), with the responsibility of reclaiming unreachable objects. The GC runs as part of your program; more specifically it is part of the *run-time system*, a set of mechanisms supporting the execution of programs. Think of the program as a parade that goes around town, horses and all, and of the GC as the cleanup brigade that respectfully and efficiently follows the same route, a few hundred meters behind.

→ *"The runtime", page 339.*

C++ implementations, as noted, generally rely on the manual approach (because of problems with the language's type system), but other modern programming languages generally use the automatic model, relying on sophisticated garbage collectors. This is the case with Eiffel but also with Java and with .NET languages such as C#. There are two main reasons for the dominance of this approach:

- Convenience: putting programmers in control of *free* operations considerably complicates the program, forcing it to perform extensive bookkeeping to determine whether objects are still referenced. With the automatic approach this is the task of a universal program, the garbage collector, available as part of the language implementation.
- Correctness: because the bookkeeping is delicate, the manual approach is a source of nasty bugs, resulting from wrongly applying a *free* to an object even though some reference is still attached to it; if execution of some other part of the program later tries to follow that reference, it will cause incorrect and usually fatal behavior, typically a crash. With a general-purpose garbage collector, the matter is treated professionally and efficiently, not for one particular program but for all programs.

The only serious argument *against* garbage collection is the possible performance overhead. More precisely, since reclaiming objects would cost some time anyway (except in the unrealistic "casual" approach), the concern is that a GC will interrupt execution, causing bursts in response time. Today's GC technology is, however, sophisticated; good GCs are incremental, meaning that instead of stopping execution for a full collection cycle (like the cleaners stopping the parade to sweep the streets) they collect *some* of the garbage *some* of the time. The resulting interruptions are invisible in most applications. The only ones that still justify concerns about garbage collection are "hard real-time" systems, such as those embedded in transportation or military devices, which require guaranteed response times at the millisecond level of faster. Such systems, however, must also renounce many of the other benefits of modern environments, such as dynamic object creation and even virtual memory.

→ *"Virtual memory", page 288.*

In ordinary environments where you rely on garbage collection, its availability is not an excuse to ignore memory performance; you can still cause leaks and fill up memory through inconsiderate usage, or by forgetting to reset references. The data structure and algorithm techniques of subsequent chapters help you avoid this pitfall.

6.8 SYSTEM EXECUTION

A final consequence of the creation mechanism is that we can now find out what the process is for *executing* a system (an entire program).

Starting it all

With object creation, execution is in fact a simple concept:

> ***Definitions*:**
> **System execution, root object, root class, root creation procedure**
>
> Executing a system consists of creating an instance — the **root object** — of a designated class from the system, called its **root class**, using a designated creation procedure of that class, called its **root creation procedure**.

The reason this suffices is that the root creation procedure (also called **root procedure** for short) may perform any actions that you have specified; in general it will itself create new objects and call other features, which may in turn do the same and so on. So you may think of your system — a collection of classes — as a set of balls on a billiards table; the creation procedures kicks the first ball, which will hit other balls that in turn will kick more.

System execution as a pool game

What is special about our billiards tables (our systems) is that a ball, when kicked, can create new balls to be kicked, and that we may end up in a single execution with millions of balls rather than a dozen or so.

The root class, the system and the design process

The root class and root procedure are there to start a process that relies on mechanisms found in the classes of the system and their features. It is important to think of these classes as interesting on their own, independently of any particular system and of its choice of root class and root procedure. As we have repeatedly seen, the classes are machines, each with its own role. A system is a particular assembly of such machines, where we have chosen one of them to start execution. But the classes exist beyond that system; a class may, for example appear in several systems, combined in each case with different other classes.

> A class that provides features of general interest, enabling it to appear in many different systems, is said to be **reusable**; classes designed for reusability will be grouped into **libraries**. Even when designing specific applications rather than libraries, you should always strive to make your classes as reusable as possible, since the potential always exists that you will again run into a similar need.

In older views of software engineering, a program was conceived as a monolithic construction consisting of a "main program" divided into "subprograms". This approach made it difficult to reuse some of the elements for new purposes, since they had all been produced as part of the fulfillment of one specific overall goal; it also hampered efforts to change the program if that particular goal changed, as it often does in practice.

More modern techniques of software architecture, based on the *object-oriented* ideas that we use in this book, fight these deficiencies by dividing the software into classes (a more general concept than subprogram) and encouraging the designer to give proper attention to each individual class, making it as complete and useful as possible.

To obtain an actual system that handles a certain computer application, you must select and combine a number of classes, then devise a root class and root procedure to kick off the execution process. In this role the root procedure resembles the traditional main program. The difference is methodological: unlike a main program, the root class and root procedure are not a fundamental element of the system's design; they are just a particular way to start off a particular execution process based on a set of classes that you have decided to combine in a particular way. The set of classes remains the center of attention.

These observations reflect some of the key concerns of professional software engineering (as opposed to amateur programming): **extendibility**, the ease with which it will be possible to adapt a system when user needs change over time; and **reusability**, the ease of reusing existing software for the needs of new applications.

Specifying the root

After this short foray into design principles, we come back to more mundane issues. One immediate question is how you will specify the root class and root creation procedure of a system.

The development environment — EiffelStudio — is there to let you define such properties of a system. They are just part of the "Project Settings" of a system, which you can access through the File → Project Settings menu. A section of the EiffelStudio appendix gives the details.

→ "Specifying a root class and creation procedure", E.4, page 845

The current object and general relativity

The perspective we have now gained on system execution enables us to understand a fundamental property of the object-oriented form of computation, which it might be tempting to call *general relativity* if the phrase had not already been preempted, a while ago, by an ETH graduate. The question is very basic: when you see a name in a class, for example the attribute name *station* in class *SIMPLE_STOP*, what does it really mean?

In principle we know, if only through the declaration and header comment:

```
station: STATION
          -- Station which this stop represents.
```

But what stop is "this stop"? In an instruction using the attribute, such as *station.set_name* ("*Louvre*"), of which station are we changing the name?

The answer can only be relative. The attribute refers to the *current object* at applicable times during execution. We have already encountered this concept informally; here is a precise definition:

← **Current** *appeared in "Not every declaration should create an object", page 114*

Definition: Current object

At any time during the execution of a system, there is a current object determined as follows:

1 The root object is, at the start of execution, the first current object.

2 At the start of a qualified call *x.f* (…), where *x* denotes an object, that object becomes the new current object.

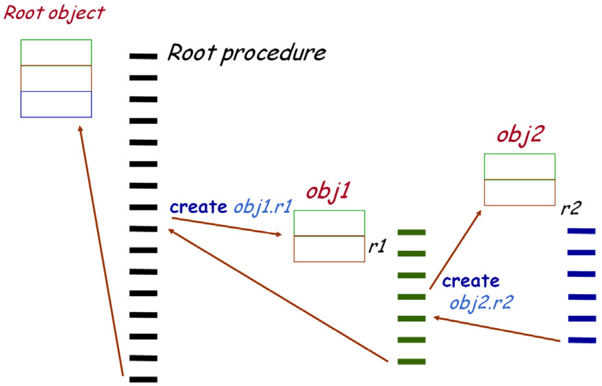
3 When such a call terminates, the previous current object becomes current again.

4 No other operation causes a change of current object.

To denote the current object, you may use the reserved word **Current**.

"Qualified" calls are the only kind we have seen so far. See the following definition.

So if you follow the execution of a system: the root object gets created; after possibly some other operations, in particular to create objects, it may perform a call using as its target one of these objects, which becomes current; it may again perform a call on another target, which will become current; and so on. Whenever a call terminates the previous current object resumes its role.



Scheme for system execution

This answers the question of what a feature name means when it appears in an instruction or expression (other than after a dot, as *f* in *x*.*f* (…)): it denotes *the feature applied to the current object*.

In class *SIMPLE_STOP*, any use of *station* — such as *Console*.*show* (*station*.*name*) to display the name of a stop's station — denotes the "station" field of the current *SIMPLE_STOP* object; this also explains "this" in header comments, as in "Station which *this* stop represents".

This convention is central to the object-oriented style of programming. A class describes the properties and behavior of a certain category of objects. It achieves this goal by describing the properties and behavior of a typical representative of the category: the current object.

These observations lead us to generalize the notion of **call**. We know that an instruction or expression with a period, such as

```
Console.show (station.name)   -- An instruction
station.name                  -- An expression
```

is a feature call, applied, like all calls, to a target object: the object denoted by *Console* in the first example and *station* in the second. But what about the status of *Console* and *station* themselves? They are calls too, with a target that is the current object. In fact you might also write them as

```
Current.Console
Current.station
```

where, as noted above, **Current** denotes the current object. You do not need, however, to use this *qualified* form in such cases; the *unqualified* forms *Console* and *station* have the same meaning. The definitions are as follows:

Definitions: Qualified and unqualified call

A feature call is **qualified** if it explicitly lists the target object, for example with dot notation, as in *x*.*f* (*args*).

A call is **unqualified** if it does not list its target, which is then taken to be the current object, as in *f* (*args*).

It is important to realize here that many expressions of whose status you may not have been quite sure until now are actually **calls** — unqualified. Examples as diverse (in the discussions so far) as uses of

```
Paris, Louvre, Line8            -- In our original class PREVIEW (chapter 2)
south_end, north_end, i_th      -- In the invariant of LINE (chapter 4)
fancy_line                      -- In the present chapter
```

belong to this category. When the invariant of *LINE* stated

← *"Class invariants", page 67.*

```
south_end = i_th (1)
```

it meant that the south end *of the current metro line* is the same as the first station *of that same line*.

← *Page 132.*

In the above definition of "current object", case 4 tells us that operations other than qualified calls and returns do not change the current object. This is true of unqualified calls: while *x*.*f* (*args*) makes the object attached to *x* the new current object for the duration of the call, the unqualified form *f* (*args*) does not cause a change of current object. This is consistent with the above observation that you may also write it **Current**.*f* (*args*).

The ubiquity of calls: operator aliases

The preceding observations show how fundamental and ubiquitous *calls* are in our programs. Along with qualified calls in dot notation, which clearly stand out as calls, simple notations like *Console* or *Paris* are calls too, unqualified.

Calls are actually present in even more deceptive guises and disguises. Take, for example, an innocuous-looking arithmetic expression, like *a* + *b*. Certainly (you might think) *this* cannot be a call! Those object-oriented folks do not respect anything, but there has to be limits; some things are sacred.

They are not. The notation *a* + *b* is, formally, just special syntax — in programming language jargon, "*syntactic sugar*" — for a **qualified call** *a*.*plus* (*b*).

The convention is simple. In the classes representing basic numerical types — bring up for example class *INTEGER_32* under EiffelStudio — you can see that features such as addition are declared in the following style:

```
plus alias "+" (other: INTEGER_32): INTEGER_32
    … Rest of declaration …
```

The **alias** specification provides the necessary syntactic sugar by allowing the form *a* + *b*, known as **infix notation**, as a synonym for *a*.*plus* (*b*), the usual object-oriented dot notation.

This is by no means restricted to integers and other classes describing basic types; you can add an **alias** clause to the declaration of:

- Any query with one argument, such as *plus*, allowing calls in infix notation (so named because the operator comes in-between the two operands).
- Any query with no argument, such as *unary_minus* **alias** "–", allowing calls in **prefix notation**, such as – *a* as a synonym for *a*.*unary_minus*.

It is permitted for the same operator to appear in aliases for both binary and unary operators; this is indeed the case for "–", which is also an alias for the binary query *minus* so that you may write *a* – *b* for *a*.*minus* (*b*).

The operators that you can use for an alias are not limited to the usual arithmetic (+, –, ∗, / etc.), boolean (**and** etc.) and relational (<, <= etc.) operators, but may be multi-character sequences not involving letters, digits and underscores and not conflicting with predefined language elements. This mechanism is particularly useful if you are writing classes representing mathematical or scientific concepts, and want to provide operator notations familiar to experts in the relevant field.

In a later chapter we will see another example of how object-oriented mechanisms swallow traditional notations, sweetened by syntactic sugar: *bracket notation* to express access to an element of a structure in an array or dictionary (hash table), as in *your_matrix* [*i*, *j*] or *phone_book_entry* ["Jane"], as just an abbreviation for query calls: *your_matrix*.*item* (*i*, *j*), and similarly *phone_book_entry*.*item* ("Jane"). It suffices for this purpose to declare the feature, called *item* in both of these examples, with the “bracket alias”, similar to operator aliases: *item* **alias** "[]".

→ *“Bracket notation and assigner commands”, page 384.*

← *Page 132.*

Object-oriented programming is relative programming

The “general relativity” nature of object-oriented programming can make you a bit dizzy at first — maybe it did until the preceding explanations — since it prevents you from understanding program elements entirely by themselves: you must interpret them in terms of the enclosing class.

I hope that by now you understand both the big picture and its influence on the writing of individual classes. They result from the modularity of the approach: its rejection of monolithic, all-in-one program architectures in favor of highly decentralized systems made of components to be developed autonomously and combined in many different ways.

6.9 APPENDIX: GETTING RID OF VOID CALLS

This is supplementary material, describing recent developments in the process of being deployed at the time of publication.

The "plague" of void calls, discussed in this chapter, is not inevitable. Recent evolution of programming languages, notably Spec# (from Microsoft Research) and Eiffel, are in the process of relegating this problem to the past.

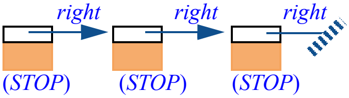
The ISO-standard version of Eiffel is indeed **void-safe**; this means that a compiler can guarantee that no system it accepts can produce any void call in any of its executions. Because the implementation is still very recent at the time of writing, we are not using void-safe Eiffel in this book. Here are a few elementary notions about the void-safe variant in case you are curious.

In ISO-standard Eiffel a type declared in the normal way, say *CITY*, is called an **attached** type and guaranteed to prevent void references. With the declaration *c: CITY*, the reference denoted by *c* will, by construction, always be attached at run time. A type only allows void references if it is used with the **detachable** keyword, as in *s:* **detachable** *STOP*.

Both examples are representative:

- Types representing objects from the application domain usually should be attached and hence exclude void: there is no such thing as a void city.
- Types representing linked data structures generally must support void values. Here we wanted to chain *STOP* instances to make up a line, where the last stop is chained to void.

A linked line



(*From original figure on page 116*.)

Guaranteeing the absence of void calls relies on two complementary techniques:

- If an entity *x* is of an attached type, it must have an associated initialization mechanism — not **Void**, the default initialization cited for references earlier in this chapter — so that before its first use in a call *x*.*f* (…) it will have been attached to an object.
- If *x* is of a detachable type, any call *x*.*f* (…) must occur in a context where *x* is guaranteed to be non-void, for example **if** *x* /= **Void then** *x*.*f* (…) **end**. There are only a small number of such recognized safe possibilities, described in the language standard and known as Certified Attachment Patterns or CAPs.

The design of this mechanism favors both compatibility and safety: the compiler will in many cases accept ordinary code — especially older code — as it is, except for the addition of **detachable** where a type should support void values; when it does reject code, this generally reflects a *real* problem: the code did carry a risk of void call at run time. In such cases, removing the problem means correcting a bug, not just causing a nuisance for the programmer.

6.10 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A reference is either *attached* to an object, or *void*.
- A feature call on an entity, such as *x*.*f* (...), will only execute properly if the value of *x* is attached to an object.
- Every reference is initially void, and remains void in the absence of any operation such as creation that explicitly attaches it to an object.
- Void references serve to indicate missing information, and to terminate linked structures.
- A creation instruction of target *x* creates a new object and attaches *x* to it.
- The form of the creation instruction is **create** *x*, or — using a creation procedure *p* specified in the class — **create** *x*. *p* (*arguments*).
- Prior to the execution of a creation procedure if any, the fields of a newly created object are initialized to standard default values, including zero for numbers and void for references.
- A creation instruction must ensure the invariant of the corresponding class. If the default initializations do not achieve this, the instruction must use a creation procedure that corrects the problem.
- Executing a system consists of creating an instance of a specified "root" class, with an associated root creation procedure.
- At any time of execution there is a *current object*: the object on which the routine last started operates.
- Calls can be qualified, applied to a target named explicitly, or unqualified, applied to the current object.
- Every unqualified mention of a feature must be understood ("general relativity" principle of O-O programming) as applying to an implicit object — the current object, a typical representative of the class.
- Calls cover many traditional operations, some of which do not need to use dot notation. Operator expressions, in particular, are special cases of calls, with an infix or prefix notation, achieved by giving features an "operator alias".
- New "void-safety" mechanisms make it possible, through static checks performed by the compiler, to guarantee the absence of void calls at run time.

New vocabulary

Alias	**Attached**	**Creation procedure**
Current object	**Detachable**	**Entity**
Exception	**Extendibility**	**Failure**
Library	**Main program**	**Operator alias**
Pseudocode	**Qualified call**	**Reference**
Reusable	**Reusability**	**Root class**
Root creation procedure	**(= Root procedure)**	**Root_object**
Unqualified call	**Void reference**	**Void-safe**

6-E EXERCISES

6-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

6-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise “Concept map”, 4-E.2, page 69.

6-E.3 Invariant for points

Consider a class *POINT* with queries *x* and *y* representing cartesian coordinates, *ro* and *theta* representing polar coordinates. Write the part of the invariant involving these queries. You may assume exact arithmetic, and the availability of appropriate mathematical functions (such as trigonometry functions).

6-E.4 Current and Void

Can **Current** have the value **Void**?

6-E.5 Attached and detachable

The appendix to this chapter describes the new void-safety mechanism based on defining every type as either attached or detachable. Examine the example classes of this chapter (and chapters 2 to 4) and state whether they should in your view be used as always attached, always detachable, or either variant depending on the context.

← “Appendix: getting rid of void calls”, 6.9, page 136.

7

Control structures

We by now have a first grasp of the *data* structure of program executions, made of objects connected by references. It is time to look at the *control* structure, which determines the order in which an execution will apply instructions to these objects.

7.1 Problem-solving structures

You may have heard this satire of the reasoning skills supposedly taught to engineers:

How to boil a pot of water

1 If the water is cold: put the pot on the fire, and wait until it boils.

2 If the water is hot: wait until it cools down. Then — as the appropriate condition is now met — apply case 1.

As a water-boiling technique it may not be the most efficient, but it provides an example of combining some of the fundamental control structures:

- The *conditional*: "if this condition holds then do this, else do that".
- The *sequence*: "do this and then do that".
- The *routine*, which enables us to name a previously identified problem-solving technique (possibly parameterized), and reuse it in any applicable context.

Remembering the discussion of *contracts* in earlier chapters, we also note that the throwback to case 1 in case 2 is only possible — as explicitly mentioned in the phrasing of case 2 — because the first step of case 2 guarantees the **precondition** of case 1 (water is cold). Preconditions and other contract techniques will indeed play a large role in getting our control structures right.

In its own light-hearted way, this example sets the proper context for our study of control structures: they are **problem-solving techniques**. To program is to solve a problem; each kind of control structure reflects a particular *strategy* for finding a solution to a problem.

The problem will always be expressed as: starting from known properties K, reach a certain goal G. In the example, K is the property that we have a pot of water and G that the water in the pot is boiling. The "strategies" provided by control structures are ways of reducing the problem to *easier* problems of that kind. For example:

- You may apply the **sequence** control structure if you find an intermediate goal I such that both of the following new problems are easier than the original (achieving G directly from K): achieve I from K; achieve G from I. Given a solution to the first new problem and a solution to the second one, the sequence control structure will apply them one after the other.
- The **conditional** control structure is the strategy of partitioning the set of possible initial situations, K, into two or more disjoint domains, so that it is easier to solve the problem separately on each of these domains.
- The **loop** structure, of which we have yet to see an example, is the strategy of solving the problem on a subset (possibly trivial) of its domain and extending the solution repeatedly until it covers the whole domain.
- The **routine** control structure is the strategy of solving a problem by recognizing that it matches another problem — often of a more general nature — to which you already know a solution.

→ *Routines are the topic of the next chapter.*

The **recursion** technique, important enough to occupy a chapter of its own, is applicable if you demonstrate that you can derive a solution by *assuming* a solution to the *same* problem applied to one or more smaller data structures.

→ *Chapter 14.*

Since programming is about solving problems, it will be particularly useful to study these and other control structures in this light.

For each of the control structures we will successively explore:

- The general idea, through *examples*.
- The *syntax* of the corresponding language construct.
- Its *semantics*: the run-time effect, in this case how the control structure governs the order of execution of the instructions it contains.
- *Correctness* rules based on Design by Contract principles, ensuring that the semantics is what we want — that executing the control structure produces a meaningful result rather than a program crash or some other unpleasant consequence.

7.2 THE NOTION OF ALGORITHM

Control structures take care of scheduling the operations in the processes carried out by computers. Such processes are called *algorithms*; this is one of the fundamental concepts of computing science. You may have seen the term already, even in the popular press which nowadays discusses things like "*cryptographic algorithms*" in reporting security issues. For the study of control structures we need a precise understanding of the concept.

Example

In general terms an algorithm is a *description* of a computational process, sufficient to enable a machine — for our interests, a computer — to carry out the process on any input data without further instructions.

You already know many algorithms. To add two integers, as in

$$\begin{array}{lr} & 687 \\ + & 42 \\ \hline = & 729 \end{array}$$

you apply the following rules (probably without thinking of them explicitly):

Touch of Elementary Math: Adding two decimal numbers

The process consists of a number of *steps*, each working on a **position** in the numbers. The position for the first step is the position of the rightmost digit of both numbers; for every subsequent step, it is the position immediately to the left of the previous one.

At every step, there is a **carry**. The initial carry is 0.

At every step, let m be the digit from the first number at the step's position and n the corresponding digit from the second number, with the convention that if either number has no digit at that position the corresponding value (m or n) is 0.

At every step, the process performs the following:

1 Compute s as the sum of three values: m, n and the carry.

2 If s is less than 10, write s at the step's position in the result line, and let the carry for the next step be 0.

3 If *s* is 10 or more, write *s* – 10 (which is a single digit, as *s* cannot be more than 19) at the step’s position in the result line, and let the carry for the next step be 1.

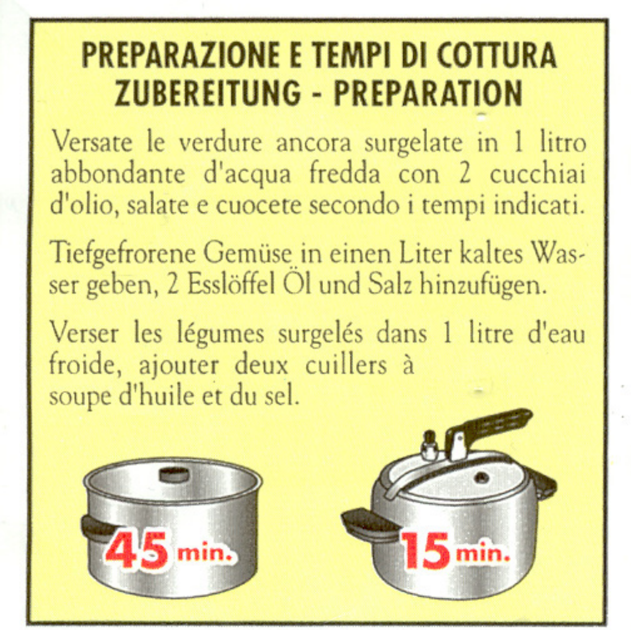
The process stops when there are no digits at the step’s position on either line and the carry is 0.

Operation 3 relies on an assumption: “*s* cannot be more than 19”. Without it, the process would not make sense, since we want to write single digits. To guarantee the correctness of the algorithm, we have to prove that the property holds at every step. Indeed, *m* and *n* are at most 9 each, so their sum is at most 18; at the first step the carry is 0, and at every following step it can only (as a result of that same operation 3) be either 0 or 1, so its sum with *m* and *n* will at most be 19. This is an example of an **invariant property**, a concept that we will study in more detail with loops.

→ *“Including the invariant”, page 158.*

Precision and explicitness: algorithms vs recipes

Although less precise than the standard for publishing algorithms, the preceding specification is more punctilious than most of the prescriptions we are used to following — with, it must be said, varying degrees of success — in ordinary life. Here is for example a trilingual set of directions on a bag of (excellent) ready-to-cook minestrone:



PREPARAZIONE E TEMPI DI COTTURA
ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiai d'olio, salate e cuocete secondo i tempi indicati.

Tiefgefrorene Gemüse in einen Liter kaltes Wasser geben, 2 Esslöffel Öl und Salz hinzufügen.

Verser les légumes surgelés dans 1 litre d'eau froide, ajouter deux cuillers à soupe d'huile et du sel.

***Not an algorithm* (*see English translation in text*)**

Credits: page 847.

In German and French the instructions state: *"Pour the frozen vegetables into one liter of cold water, add two tablespoonfuls of oil and salt*". What's striking is not so much the lack of precision ("tablespoonful" can be given an exact conventional value, and anyway the idea of using such a general term is that it does not matter too much whether you take a slightly bigger or smaller tablespoon) as the absence of the key instruction: if you want to get an edible result, you'd better heat the thing up at some point. Only the Italian version mentions this detail — "*cook according to the times given*" — which makes the pictures meaningful.

For such instructions intended for human interpretation, lack of explicitness is not an issue; it will be immediately clear to most readers that they cannot prepare such food without heating it, and that the pictures indicate cooking times (even I succeeded!). But what works for a cooking recipe would not suffice for an algorithm. You must specify every operation, every detail of the process; and you must specify them in a form that leaves no room for ambiguity.

Properties of an algorithm

For algorithms, as opposed to informal recipes, we expect a number of properties captured by the following definition:

Definition: Algorithm

An algorithm is the specification of a process acting on a (possibly empty) set of data, satisfying the following five rules:

A1 The specification defines the applicable sets of data.

A2 The specification defines a set of elementary actions, from which all steps of the process are drawn.

A3 The specification defines the possible order or orders in which the process may carry out these steps.

A4 The specification of the elementary actions (rule A2) and of the permitted orderings (rule A3) relies on precisely defined conventions, allowing the process to be carried out by an automaton (such as a computer) without human intervention, with results for the same set of data guaranteed to be the same on two different automata following the same conventions.

A5 For any set of data to which the process is applicable (as per rule A1), the process is guaranteed to terminate after executing a finite number of the algorithm's steps.

The above method for adding two numbers possesses the required properties:

A1 It describes a process to be applied to some data, and specifies the kind of data: two integers expressed in decimal notation.

A2 The process relies on well-defined basic actions: set a value to zero or to a known number, add three numbers, compare a number to 10.

A3 The description specifies in what order to apply such actions.

A4 It is precise. That precision should be enough for any two people to understand and apply the algorithm in the same way, although, as noted, it may not suffice for other goals.

A5 For any applicable data — two numbers in decimal notation — the process will terminate after a finite number of steps. This is intuitively clear but must be ascertained rigorously; we will see how to do this by showing that the quantity *M – step* + 1 is a *variant*.

→ *"Loop termination and the halting problem", page 161.*

Rule A3 of the definition mentions the possible "order or *orders*" of the steps. A *sequential* and *deterministic* algorithm defines a single order of steps for any possible execution. This is not the only possibility:

- *Non-deterministic* algorithms specify for certain steps a set of actions of which one will be executed, but do not specify which; a *probabilistic* algorithm, as a special case, defines a random strategy for these choices.
- *Concurrent* algorithms specify for certain steps a set of actions to be executed in parallel, as appropriate for networks and multicore computers.

In this book we need only consider sequential, deterministic algorithms.

Algorithms *vs* programs

You may wonder, in light of the preceding definition, what distinguishes an algorithm from a program. The basic concept is indeed the same.

It is sometimes said that the difference is the *abstraction* level: that a program is meant to execute on a particular machine, whereas an algorithm is an abstract definition of a computing process, independent of any computing devices. This made sense a few decades ago, when programs were expressed in low-level codes for specific computers. Algorithms then served to express the *essence* of programs: the computing process described independently of any computer. But that view is no longer applicable today:

- To express *programs*, we can use clear, high-level notations, defined at a level of abstraction far above the details of any particular computer. The Eiffel notation used in this book is an example.
- To express an *algorithm* in a way that fully meets the definition's requirements, in particular the requirement of precision — condition A4 —, we will need a notation with rigorously defined syntax and semantics, making it in the end equivalent to a programming language.

It is true that practical descriptions of algorithms often refrain from specifying some details, such as choices of data structures, which a program cannot omit since it wouldn't then compile and execute. This practice does seem to suggest that algorithms are more abstract than programs. But it is only a useful convention to facilitate publication; for that purpose, it renounces some of the precision that true algorithms require (condition A4 again), and every reader of the description understands that to get an algorithm in the official sense one would need to bring the missing details back in.

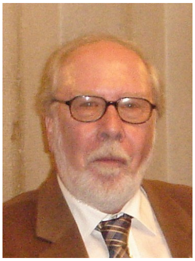
So we cannot rely on the level of abstraction to distinguish algorithms from programs. Two differences — or nuances — are more significant:

- An algorithm describes a single computing process. Decades ago this was also the goal of a typical program — "*Compute the monthly payroll!*" — but programs today involve *lots* of algorithms. We have already seen several in the Traffic system (display a line, animate a line, display a route…) and there are hundreds more. The same observation would apply to any significant software product. It is the reason why this book tends to use, rather than "program" (which may still suggest the idea of doing just one task), the word *system*.

- As important to a program as the description of the processing steps is the description of the data structure — in the object-oriented approach of this book, the **object structure** — to which they apply. This criterion is not absolute either, since you cannot really separate the algorithmic steps from the structure they manipulate. But in describing programming concepts we may sometimes want to emphasize the processing aspect — the algorithm in a narrow sense of the term — and sometimes the data aspect. This explains the title of a classic programming book by Niklaus Wirth (published in 1976):

Algorithms + Data Structures = Programs.

The object-oriented approach to software construction gives the central role to the data, more specifically to the object types: the classes. *Every* algorithm is then attached to a particular class. Eiffel applies this rule without exception: every algorithm that you write will appear as a *feature* of some class. This approach is justified by considerations of software quality that we will explore in later chapters. It implies, for this book, that we will study the algorithm and data aspects in close connection.



Wirth (2005)

Control structures, as reviewed in this chapter, are one example of an algorithmic concept not directly related to a particular kind of data structure.

7.3 CONTROL STRUCTURE BASICS

The specification of an algorithm must include items of two kinds:

- The elementary steps to execute (clause A2 of the definition of "algorithm"). ← *Page 143.*
- The order of their execution (clause A3).

Control structures handle the second of these needs. Precisely:

> ***Definitions*: Control flow, control structure**
>
> The scheduling of a program's operations during execution is called its **control flow**.
>
> A **control structure** is a program construct affecting the control flow.

Or "flow of control".

There are, as previewed, three fundamental forms of control structure:

- The **sequence**, consisting of instructions listed in a certain order; its execution consists of executing these instructions in the same order.

  It is the control structure we have been using implicitly in all the examples so far, since we have been writing instructions under the assumption that they would be executed in the order given.
- The **loop**, containing a sequence of instructions to be executed repeatedly.
- The **conditional**, consisting of a condition and two sequences of instructions; its execution consists of executing one or the other of these sequences depending on whether the condition — a boolean expression — evaluates to **True** or **False**. It can be generalized to a choice between more than two possibilities.

These are mechanisms for scheduling the execution of our programs' instructions, taking advantage of three fundamental capabilities of computers:

- Executing *all* of a set of specified actions, in a specified order.
- Executing a *single* specified action, or some variants of it, *many times*.
- Executing *one* of a set of specified actions, depending on a specified condition.

Such control structures assume that the program's execution will be doing at most one thing at a time. With several computers, or a single computer sharing its time between different programs, you can have *parallel (or concurrent*) execution, leading to new control structures that we will not study here.

Our basic control structures can be combined without restriction, so that you may for example write a conditional involving two sequences of instructions, some of which are in turn loops or conditionals, themselves involving further substructures. Such a description of a computing process, consisting of instructions grouped into control structures describing their run-time scheduling, constitutes an algorithm.

These notions are the subject of the following sections. In addition we will review two other forms of control structuring:

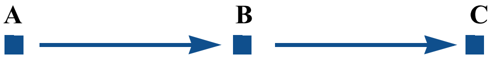
- The *branching instruction*, also known as **goto** ("Go to" written as one word), which has fallen from grace as a tool for programmers — we will see why — but still plays a role in computer instruction codes.
- *Exception handling*, providing ways to recover from abnormal run-time events (such as a void call) that interrupt the usual flow of control.

When you have defined an algorithm, you will often want to wrap it into a program unit with a name, which you can then use through that name. Such a grouping is known as a **routine**, a fundamental form of program structuring, achieving on the control side what classes give us on the data side. Routines enable you in effect to add *new* control structures to the available repertoire, by abstracting particular combinations of existing structures. They are the topic of the next chapter.

7.4 SEQUENCE (COMPOUND INSTRUCTION)

The sequence control structure applies a problem-solving pattern familiar to everyone: identify one or more intermediate goals, so that we can proceed in steps. If there is only one intermediate goal we will solve two separate problems:

- Achieve the intermediate goal from the hypothesis.
- Achieve the final goal from the intermediate goal.



Reaching a goal through an intermediate step

More generally, with n intermediate goals we will have $n + 1$ steps, where step i (for $2 \leq i \leq n$) has to achieve the i-th intermediate goal from the preceding one.

Examples

In our application domain of city travel, a typical example of sequence is a possible strategy going from a place *a* to a place *b*:

1 On the map, find the metro station *ma* closest to *a*.

2 On the map, find the metro station *mb* closest to *b*.

3 Walk from *a* to *ma*.

4 Take the metro from *ma* to *mb*.

5 Walk from *mb* to *b*.

This is a human strategy, not a program. A program might build a *route* from *a* to *b*. A route, as you remember, is made of *legs*. Using declarations for the route and its legs

← “Objects you can and cannot kick”, page 25.

```
full: ROUTE
walking_1, walking_2, metro_1: LEG
```

you may build the route through the sequence of instructions

```
                                                    -- Version 1
create walking_1.make_walk (a, ma)
create walking_2.make_walk (mb, b)
create metro_1.make_metro (ma, mb)
create full.make_empty
full.extend (walking_1)
full.extend (metro_1)
full.extend (walking_2)
```

With a condition on ma and mb, see below.

This takes advantage of the following creation procedures of class *LEG*:

- *make_walk*, producing a walking leg from one place to another.
- *make_metro*, producing a metro leg from one place to another (with a precondition requiring existence of a line that goes through both places, since a leg of metro route must all be on one line).

and the following features of class *ROUTE*:

- The creation procedure *make_empty*, producing an empty route.
- The command *extend*, adding a leg at the end of a route.

Programming time!
Creating and animating a route

Using the above scheme, write and execute a routine that will create a route from *Elysee_palace* to *Eiffel_tower* (both place names are defined as features in class *TOURISM*), and animate the route.

Put the corresponding software elements, and the remaining ones for this chapter, in a new class called *ROUTES*. The name of the system for the examples and exercises of this system is *control*.

← Taken as usual from the chapter name; the directory is 07_control. The conventions were given in 2.1, page 15.

For this and future programming exercises, you will no longer be given a step-by-step description of how to write, compile and run the example, unless this involves some EiffelStudio mechanism that we have not seen yet. All the necessary techniques have been seen before; if you have any hesitation consult the EiffelStudio appendix or go back to the earlier examples.

→ E, page 843.

Compound: syntax

As the above example shows, the sequence control structure is not new with this chapter: we have seen it many times before — in fact, ever since our very first program example — without having a name for it. We simply wrote several instructions in the intended order of execution, as in

← *Feature explore, page 18.*

```
Paris.display
Louvre.spotlight
Metro.highlight
Route1.animate
```

Since it is often useful to consider such a sequence of instructions as a single instruction — for example to make it part of a bigger control structure — it is also called a **compound instruction**, or just “compound”.

The syntax rule is very simple:

Syntax: Compound instruction

To specify a sequence, or *compound*, of zero or more instructions, write them one after the other in the desired order of execution, optionally separated by semicolons.

We have not used semicolons so far. The style rule indeed suggests not to bother with them:

Touch of Style: Semicolons between instructions

- If (as should almost always be the case) successive instructions appear on separate lines, **omit** the semicolon.
- In the occasional case of two instructions appearing on the same line (to be used only for very short instructions and if there is a good reason to save on the number of lines), **always** separate them by a semicolon.

So if you will be printing out the above “Version 1” example and are down to your last roll of paper (or have a very environmentally-conscious boss), you might write the last three instructions as

```
full.extend (walking_1) ; full.extend (metro_1) ; full.extend (walking_2)
```

but there is seldom a reason to do so. Instead, you will usually have one line per instruction; then you can just forget the semicolons.

It is important to remember that the separation into lines does **not** by itself carry any semantic value; line return is just a "break" character, which has the same effect as a space or a tab. So nothing prevents you from writing

← *From "Breaks and indentation", page 45.*

```
full.extend (walking_1) full.extend (metro_1) full.extend (walking_2)
```

Ugly!

Nothing, that is, except good taste, elementary common sense, the official style rules, and any hint of a trace of a shadow of a tinge of concern for whoever is going to try to read your program later, including two readers of particular interest: the instructor (if you are taking a course); and — after a few days, weeks or months — yourself.

Even on separate lines, some people are initially nervous about omitting the semicolons, perhaps because many commonly used programming languages have strict rules requiring them in many places and prohibiting them in others. To get over semicolon addiction, a simple test suffices: put two versions of the same program side by side, both with a single instruction per line, but one with semicolons and the other without; you will see right away that the second one is cleaner and more readable.

If you do use semicolons, mistakenly including an extra one will be harmless, because *instruction_1* ; ; *instruction_2* is formally understood as *three* instructions, of which the second is an *empty* instruction whose semantics is to do nothing. So this will not cause any trouble. All the same, it is better to clean up your code and remove any unneeded element.

Compound: semantics

The run-time behavior of a sequence is what the name of this control structure and the earlier informal discussion suggest:

***Semantics*: Compound instruction**

Executing a sequence of instructions consists of executing each instruction in turn, in the order given.

Note that the syntax description talks of "zero or more" instructions (not one or more) and hence permits an empty sequence, whose semantics is the same as for an empty instruction: do nothing. Not a very exciting case, but sometimes useful for a sequence that is part of a larger structure.

Order overspecification

You may have noticed that in the above example ("Version 1") the chosen order is only one of a number of possibilities. For example we could add each leg to the full route as soon as we have created it:

← Page 148.

```
                                                        -- Version 2

    -- Create the route:
create full.make_empty

    -- Create and add the first leg:
create walking_1.make_walk (a, ma)
full.extend (walking_1)

    -- Create and add the second leg:
create metro_1.make_metro (ma, mb)
full.extend (metro_1)]

    -- Create and add the third leg:
create walking_2.make_walk (mb, b)
full.extend (walking_2)
```

Many other orders are possible; the only constraints for this example are that any instruction using an object (route or leg) must come after the creation instruction for that object, and that we add legs in the right order.

Using the sequence control structure often creates such cases of **overspecification**, that is to say, of a solution that is not the most general possible one. This does not directly harm the software, but one must be conscious that the solution is only one of a set of possibilities.

When execution speed is a concern, it is sometimes possible to obtain faster execution by executing some group of instructions *concurrently* (in parallel) with others. The four initial creation instructions of the above example can, for example, be executed concurrently without affecting the result. Concurrency, however, is a delicate matter; programmers usually do not explicitly prescribe it for such elementary cases, but good compilers may be able to produce concurrent code if the underlying computer architecture supports concurrency, as with modern "multicore" computer systems.

Compound: correctness

We have seen that a feature may have a contract including a precondition and a postcondition. These properties govern calls to the feature, such as the above call *full.extend (walking_1)*. The precondition tells the client (the calling feature) what it *must* guarantee to be correctly serviced. The postcondition tells the client what it *may* assume on termination of such a correct call.

Similarly, every control structure reviewed in this chapter has an associated correctness rule, which puts some conditions on the contracts (preconditions and postconditions) of its constituent instructions, and defines the resulting contract for the structure as a whole. For Compound, the correctness rule reflects the property that the constituent instructions will be executed in the order given:

Correctness: Compound instruction

For a Compound instruction to be correct:

- The program must ensure that the precondition of the Compound's first instruction, if any, holds prior to any execution.
- The postcondition of every instruction in the Compound must imply the precondition of the following one if any.
- The postcondition of the last instruction must imply the postcondition desired for the entire Compound.

Special case: an empty Compound is by itself always correct, but achieves no new postcondition.

In our example, you may check the contract for the feature *extend* of class *ROUTE* by bringing up the class in EiffelStudio. With some postcondition clauses omitted, it reads

```
extend (l: LEG)
    require
        leg_exists: l /= Void
    ensure
        lengths_added: count = old count + 1
```

Every creation instruction of the form **create** *x* or **create** *x.make* (…) ensures that the condition *x* /= *Void* will hold after its execution. So our example satisfies the correctness rule for compound instructions; this is true in both "Version 1" and "Version 2", but would not hold any more if we changed the order of the instructions to start with

←*Clause 2 of "Touch of Methodology: Creation Instruction Correctness Rule", page 126.*

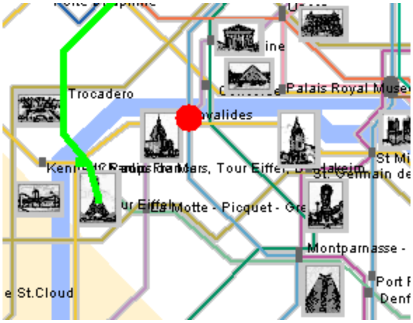
```
                                                        -- Version 3
    -- Create the route:
create full.make_empty

    -- Create and add the first leg:
full.extend (walking_1)
create walking_1.make_walk (a, ma)
```

where *walking_1*, at the place of use, denotes a non-existent *LEG* object that the extract mistakenly attempts to add to the route.

7.5 LOOPS

Our second control structure, the loop, taps into one of the most amazing features of computers: their ability to repeat an operation, or variants of that operation, many times — *very* many times by human standards.

A typical example of a loop is an animation scheme to highlight a metro line by displaying a red dot on each of its stations in turn, for half a second. The system *Show_line* in the Traffic delivery does this. You can execute it now if you wish; the effect at one of the intermediate steps is this:



Highlighting a station

Here is a loop that achieves this effect. It uses *show_spot* (*p*) to display a red spot at point *p* on the screen for half a second (the value predefined for *Spot_time*). To understand the details we need concepts introduced later in this discussion, so you should just take this example as an introduction to how a loop looks:

```
from
    Line8.start
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
end
```

The loop body

The loop moves a “cursor” (a virtual marker) to the beginning of the line (*start*); then until the cursor is beyond the last position (*is_after*) it performs the following for each successive station (*item*): display the red spot at the *location* of the station, and advance the cursor to the next station through command *forth*.

Each such execution of the loop body is called an *iteration* of the loop.

This shows some of the key ingredients of a loop: initialization (**from**), exit condition (**until**), and actions to be repeatedly executed (**loop**). To get a full understanding of this loop we must first explore some of the underlying concepts.

→ *In “Animating a metro line”, page 166.*

Programming time!
Animating Line 8

Put the preceding loop in a feature *traverse* of class *ROUTES* (the example class for this chapter). For this example and subsequent variations, update the class and run the system to observe the results.

Loops as approximations

As a problem-solving technique, the loop is the method of approximating the result on successive, ever bigger subsets of the problem space.

In the metro line animation example, the problem is to display a red dot on each station of the line. Successive approximations are: display a dot on no station at all; display it on the first station; display it successively on the first two stations; and so on.

Here is another example. Assume you want to know the maximum of a set of one or more values N_1, N_2, …, N_n. The following strategy, described informally, will work:

I1 Define *max* to be N_1. It is then true, trivially, that *max* is the maximum of the set of values containing just one value, N_1.

I2 Then for every successive $i = 2$, …, n do the following: if N_i is greater than the current *max*, redefine *max* to be N_i.

This ensures that at the i-th step (where the first step corresponds to case I1 and the subsequent steps to case I2 for $i = 2$, $i = 3$ etc.) the following property, called a **loop invariant** of the loop, holds:

Loop invariant of the "maximum" strategy, at step *i*

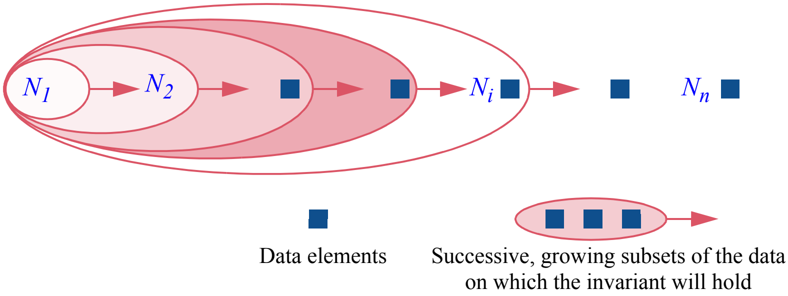
max is the maximum of $N_1, N_2, \ldots, N_i$

Stating this invariant for the n-th step, case I2 for $i = n$, gives

"*max* is the maximum of $N_1, N_2, \ldots, N_n$"

which is the desired result.

The following picture illustrates the loop strategy in this case:


N_1
N_2
N_i
N_n
Data elements
Successive, growing subsets of the data on which the invariant will hold

Finding a maximum by successive approximations

The loop establishes the invariant property "*max* is the maximum of the first i values" for a trivial value: $i = 1$; then it repeatedly extends the subset of the data on which the invariant holds.

In the metro line animation example, the invariant would be: "A red dot has been displayed on all the stations visited so far".

The notion of invariant is not new, since we have already encountered *class* invariants. The two forms of invariant are related, since both describe a property that certain operations must preserve. But their roles are different: a class invariant applies to an entire class and must be maintained by the execution of *features* of the class; a loop invariant applies to a single loop and must be maintained by every iteration of the *loop body*.

← "Class invariants", page 67.

The loop strategy

Although many loops are more sophisticated, the "maximum" example illustrates the general form of loops as a problem-solving strategy.

The strategy is useful when a problem consists, starting from some initial property *Pre*, of establishing a certain goal *Post* characterizing some set of data *DS*. This set is finite, although it might be very large.

To use a loop is to find a weaker (more general) form of the goal *Post*: a property *INV* (*s*) — the loop's invariant — defined on subsets *s* of *DS* (not just *DS* itself), with the following properties:

L1 You know an initial subset *Init* of *DS* such that the initial condition *Pre* implies *INV* (*Init*); in other words, the invariant holds for the initial subset.

L2 *INV* (*DS*) (that is to say *INV* applied to the whole set) implies your goal *Post*.

L3 You know a technique, applicable when *INV* (*s*) holds for a set *s* that is not yet all of *DS*, to make *INV* (*s'*) hold for a larger subset *s'* of *DS*.

The "maximum" example has all these ingredients: *DS* is the set of numbers $\{N_1, N_2, \ldots, N_n\}$; the precondition *Pre* is the property that *DS* has at least one element; the goal *Post* is the property that we have found the maximum of these numbers; and the invariant *INV* (*s*), where *s* is a subset $N_1, N_2, \ldots, N_i$ of *DS*, states that we have found the maximum of *s*. Then:

M1 If *Pre* is satisfied, meaning that there is at least one number, we know an initial subset *Init* such that *INV* (*Init*) holds: just take the set consisting of only the first number N_1.

M2 *INV* (*DS*) — the invariant applied to the whole set $\{N_1, N_2, \ldots, N_n\}$ — does imply the goal *Post*; actually, it is identical.

M3 When *INV* (*s*) holds for a set $s = \{N_1, N_2, \ldots, N_i\}$ which is not all of *DS* — in other words, $i < n$ — then we can establish *INV* (*s'*) for a larger subset *s'* of *DS*: we just take *s'* to be $\{N_1, N_2, \ldots, N_i, N_{i+1}\}$, and the new maximum to be the greater of the previous maximum and N_{i+1}.

Note — in the general case — how carefully the invariant is devised to fit our general strategy of solving a problem by successive approximation:

- *INV* is sufficiently **weak** that we can establish it easily for some initial subset, usually very small, of the whole data set.

- It is sufficiently **strong** to give us the entire desired goal, *Post*, when applied to the whole set.

- It is sufficiently **flexible** to let us extend it from any applicable subset to a slightly larger one.

By repeatedly performing this extension, having started by establishing the invariant on the initial subset, we will get to the desired result. This strategy of successive approximations of the goal, on progressively larger sets, might take many iterations; but computers are fast, and they do not go on strike to complain of repetitive work.

These observations define how the loop works as a control structure. Its execution will:

X1 Establish *INV* (*Init*), taking advantage of L1. This gives you *Init* as a first subset *s* on which *INV* holds.

X2 As long as *s* is not the complete set *DS*, apply the technique of L3 to establish *INV* on a new, larger *s*.

X3 As soon as *s* is the whole of *DS*, stop: you have established *INV* on *DS*, which thanks to L2 establishes your goal *Post*.

This process is guaranteed to terminate because we always assume *DS* to be a finite set; since *s* remains a subset of *DS*, and grows by at least one element at every step, it has to reach the full *DS* after a finite number of iterations of step X2. In some cases, however, establishing termination will require more care.

← Page 155.

→ "Loop termination and the halting problem", page 161.

Loop instruction: basic syntax

To express the loop strategy as a program text we will use, in the "maximum" example, the following general structure:

```
from
    -- "Define max to be N1"
    -- "Define i to be 1"
until
    i = n
loop
    -- "Redefine max as the greater of the current maximum and Ni+1"
    -- "Increase i by one"
end
```

← Remember that comments in red are pseudocode: "Touch of Style: Highlighting pseudocode", page 109.

For the moment, all the constituent instructions are still in pseudocode. The example illustrates the three required parts of a loop construct (to be complemented later by two *optional* parts):

- The **from** clause introduces the initialization instructions (X1).
- The **loop** clause introduces the instructions to be executed in each of the successive iterations (X2).
- The **until** clause introduces the exit condition: the condition under which the iterations will terminate (X3).

The run-time effect of this construct, suggested by the keywords (**from**, **until**, **loop**), is in line with the previous discussion:

- First, execute the instructions in the **from** clause (*initialization*).
- Then execute the instructions (*body*) in the **loop** clause until the condition stated in the **until** clause (*exit condition*) holds.

The last point means more precisely that after the initialization the body will be executed:

- **Not at all**, if the exit condition holds immediately after the initialization.
- Once, if one execution of the body leads to the exit condition being true.
- More generally: i times for some i, if the exit condition will be false after j executions of the body for $1 \leq j < i$, and true after i executions.

Syntactically, the **from** and **loop** clauses each contain a compound instruction. You may as a consequence include any number of instructions, including zero. It indeed happens that a loop does not need an explicit initialization instruction (in cases when the context before the loop already implies the invariant); then the **from** clause will be empty:

```
from
                ←———————————— Nothing here

until
    "Exit condition"
loop
    -- "Loop body"
end
```

This does not apply to the **loop** clause, since it must make some progress in the approximation (bring at least one new element to the subset s of the previous discussion); otherwise the loop process would never terminate. So in a realistic program you will never write an empty **loop** clause.

Including the invariant

The basic form of loop, as just seen, does not show the loop invariant. This is regrettable since the invariant is essential to understanding what the loop is about. The optional but recommended **invariant** clause takes care of this. With this clause our example becomes:

```
from
    -- “Define $max$ to be $N_1$”
    -- “Define $i$ to be 1”
invariant
    -- “$max$ is the maximum of $\{N_1, N_2, \ldots, N_i\}$”
until
    $i = n$
loop
    -- “Redefine $max$ as the greater of the current maximum and $N_{i+1}$”
    -- “Increase $i$ by one”
end
```

The invariant in this example is still pseudocode, but useful nonetheless to convey essential information about the loop.

Loop instruction: correctness

The invariant of a loop has two characteristic properties:

Correctness: Loop Invariant Principle

The invariant of a loop must be:

I1 Ensured by the initialization (**from** clause)

I2 Preserved by the body (**loop** clause) whenever this body is executed with the exit condition not satisfied.

An instruction “preserves” a property if its execution, started with that property satisfied, terminates with the property satisfied again. This preservation property explains the name “invariant”, applied here to loop invariants (as earlier to *class* invariant, which must similarly be ensured upon instance creation and preserved by features of the class).

← *“Class invariants”, page 67.*

As we have seen, the purpose of a loop is to achieve a certain outcome by successive approximations. The steps towards this goal are the initialization and then successive executions of the body. After each of these steps, a property holds that is an approximation of the final desired outcome; it is the invariant. In our two examples the invariants, in pseudocode, are:

- “max is the maximum of $\{N_1, N_2, \ldots, N_i\}$”, as the i-th approximation, for $1 \leq i \leq n$, of the final property “max is the maximum of $\{N_1, N_2, \ldots, N_n\}$”.
- “A red spot has been displayed on all stations visited so far, in their order on the line”, as an approximation of the final property that the spot has been displayed on all stations in order.

Let *Loop_invariant* be the invariant. When the loop execution terminates, the invariant will still hold because of properties I1 and I2 of the Loop Invariant Principle. In addition, the exit condition *Loop_exit* will hold: otherwise the loop would not have terminated yet. So the final condition produced by the loop is

```
Loop_invariant and Loop_exit
```

This is the outcome achieved by the loop:

> ***Correctness*: Loop Postcondition Principle**
>
> The condition achieved by the execution of a loop is the conjunction of its invariant and its exit condition.

The syntax highlights the Loop Postcondition Principle by putting the **invariant** and **until** clauses next to each other. So if you see a loop with its invariant and want to know what it achieves, just look at the two clauses together:

```
from
    -- "Define max to be $N_1$"
    -- "Define $i$ to be 1"
invariant
    "max is the maximum of $\{N_1, N_2, …, N_i\}$"
                                                    and      Final condition
until
    $i = n$
loop
    -- "Redefine max as the greater of the current maximum and $N_{i+1}$"
    -- "Increase $i$ by one"
end
```

The effect of the loop is their conjunction (their **and**).

In the metro line animation example, the exit condition is, informally, "all stations have been visited"; conjoined with the invariant stated above, this tells us that a red dot has been displayed on all stations, in the order of the line.

The Loop Postcondition Principle is of course the direct consequence of how loops were defined in the first place, as an approximation mechanism. Quoting from that earlier discussion, the idea was to choose as invariant a generalization of the final goal, choosing it so that it is:

← *Page 156.*

- "*Sufficiently* **weak** *that we can establish it easily for some initial subset of the whole data set*": this is the role of the initialization.

- "*Sufficiently* **flexible** *to let us extend it from any applicable subset to a slightly larger one*": this is the role of the body, executed when the invariant is satisfied and the exit condition is not satisfied; it then yields a state where the invariant is satisfied again.

- "*Sufficiently* **strong** *to give us the entire desired goal when applied to the whole set*": this is achieved on exit, as per the Loop Postcondition Principle, by the conjunction of the exit condition and the invariant.

Loop termination and the halting problem

The loop execution scheme, as described, repeatedly performs the loop body until the exit condition is satisfied. If the loop derives from a well-devised approximation strategy as above, its execution will terminate after a finite number of iterations: since the set being approximated is finite and each iteration adds a new element to its approximation, the process cannot go forever. But the loop syntax permits an arbitrary initialization, exit condition and loop body, so it could in principle execute forever, like

← As noted on page 157.

```
from
    "Any instruction here (or none at all)"
until
    0 /= 0
loop
    "Any instruction here (or none at all)"
end
```

In this extreme example the exit condition — which of course you may also write as just **False** — can never be satisfied, so the loop cannot ever terminate. If you execute the corresponding program, you will be sitting at the terminal with nothing happening; after a while you will probably realize that something's wrong, so you will interrupt the program (EiffelStudio has a button for that purpose). But you have no way to know — if you are just a user of the program, and have no access to its text — whether the program is really looping forever or just taking a long time to execute.

The failure of a program to terminate is not always, mind you, erroneous behavior: some programs are expressly designed to run forever or until explicitly stopped. An example is the "operating system" (OS) that runs a computer: as I am typing the text of this paragraph, I would be very upset if the OS terminated, which could only mean that my system has crashed or that I kicked the power switch with my foot. Same story with many "embedded systems" (programs running on devices): you do not want your cell phone's program to terminate while you are talking.

On the other hand ordinary programs — in particular most of the programs we discuss in this book — are expected to process some input, then yield a result after a finite time.

When writing such programs you may inadvertently produce a non-terminating loop, which makes the whole program also non-terminating. To avoid this unpleasant result the best technique is to make sure you define, for each loop that you write and that is meant to terminate, a **loop variant**:

***Definition*: Loop variant**

A variant for a loop is an integer expression possessing these properties:

V1 After execution of the loop initialization (**from** clause), the variant has a non-negative value.

V2 Every execution of the body (**loop** clause), when the exit condition is not satisfied and the invariant satisfied, decreases the value of the variant.

V3 Every such execution also keeps the variant non-negative.

If indeed you can find such an expression, then you have shown that the loop will terminate after a finite number of iterations: it is not possible for a non-negative integer value to decrease forever while remaining non-negative. In fact, if we know the original value V of the variant after initialization, we can tell that the loop will terminate after at most V iterations, since each iteration decreases the variant by at least 1.

> For this reasoning to hold, the variant must indeed be an integer. Real numbers would not work, since it is perfectly possible (in mathematics, if not on a computer) for an infinite sequence of real numbers, such as the sequence $1, 1/2, 1/3, \ldots, 1/n, \ldots$, to consist of ever decreasing values.

If you know a variant, the syntax lets you specify it in a **variant** clause after the loop body (**loop** clause). For example we may add a specification of the loop variant to our computation of the maximum:

```
from
    -- “Define $max$ to be $N_1$”
    -- “Define $i$ to be 1”
invariant
    $1 <= i$
    $i <= n$
    -- “$max$ is the maximum of $\{N_1, N_2, …, N_i\}$”
until
    $i = n$
loop
    -- “Redefine $max$ as the greater of the current maximum and $N_{i+1}$”
    -- “Increase $i$ by one”
variant
    $n – i$
end
```

The variant identified here is $n – i$. It indeed satisfies the conditions:

V1 The initialization sets i to 1. The program assumes that $n \geq 1$. So the variant is initially non-negative.

V2 The loop body increases i by one, therefore decreases the variant by one.

V3 When the exit condition is not satisfied, i will be less than n ($i < n$, not just $i <= n$), and hence $n – i$, when decreased by one, will remain non-negative.

For the last point V3, it is not sufficient to consider the negation of the exit condition, which only tells us that $i /= n$: we need to be sure that $i < n$. But note the new invariant properties added above: $1 <= i$ and $i <= n$. These are ensured by the initialization and preserved by the body when executed with $i /= n$, so they are indeed invariant. Then when the exit condition is not satisfied, that is to say, $i /= n$, we know from the invariant property $i <= n$ that in fact $i < n$.

You may well feel at this point that I am splitting hairs and that the loop as given is evidently correct — that it will always terminate, having computed the maximum of the given set of values. But in practice it is a common mistake to write a loop that will not terminate. If you have ever tried to use a program only to see it “hang”, it might very well have been the result of such a mistake on the part of its author. Maybe the problem did not appear in the program tests; tests can only capture a small part of all possible cases. Only through the kind of reasoning illustrated above can you guarantee — for your own programs — that a loop will *always* terminate, regardless of the program’s inputs.

Considering the possibility of non-termination leads to important notions which you will study in more detail in a course on the theory of computation:

Touch of Theory: The Halting Problem and undecidability

The prospect of a loop that runs forever is disturbing. Isn't there some automatic way, given a program, to check that every loop in it will terminate? Compilers already perform some other verifications for us, in particular *type checks* (if *x* is of type *STATION* and you write a feature call *x*.*f*, the compiler will issue an error message and refuse to compile your program unless *f* is a feature of class *STATION*). Perhaps they could also check loop termination?

The answer to this general question is *no*. A theorem states that — assuming a programming language powerful enough for practical needs — it is *impossible* to write a program (such as a compiler) that will correctly report, when fed any program text, whether that program will always terminate. This is known as the **undecidability** of the **Halting Problem**:

- The *Halting Problem* is whether a program will terminate (halt).
- A problem is *undecidable* if no effective technique exists that will yield a correct solution in every case.

The Halting Problem is the most famous undecidability result in the theory of computation, although not the only one.

Depressing as it sounds, this result does not prevent you in practice, when you write a program, from guaranteeing — as you should! — that it will terminate. The undecidability theorem rules out any *general* automatic mechanism that would ascertain termination for *any* program, but not *specific* techniques for demonstrating that *some* programs will terminate. The use of an explicit loop variant is such a practical technique — a very effective one. If you can prove that an integer expression has the variant properties V1 to V3 (initially non-negative, decreased, and maintained non-negative by every iteration), then you have the guarantee that the loop will terminate.

Commercial-grade compilers are not yet able to perform such proofs, so you will have to do them manually by inspecting the program, and, if there is any doubt, let EiffelStudio check at run time that the variant decreases on each iteration. Unlike the general Halting Problem, this is not a fundamental impossibility but a limitation of current technology.

We will be able to *prove* the theorem asserting the undecidability of the Halting Problem once we have studied routines.

→ *"An application: proving the undecidability of the halting problem", page 223.*

Touch of History: Tackling the Halting Problem

The Halting Problem was described — as a special case of the “decision problem”, or *Entscheidungsproblem*, a general question going back to Leibniz in the 17th-18th century and Hilbert in the early 20th —, and its undecidability proved, a decade before the appearance of actual stored-program computers, in a famous mathematical paper of 1936, “*On Computable Numbers*, *with an Application to the Entscheidungsproblem*”.

The author, the British mathematician Alan Turing, relied on an abstract model for a computing machine, known today as the **Turing machine**. The Turing machine — a mathematical concept, not a physical device — is still actively used to discuss general properties of computation, independent of any particular computer architecture or programming language.

Turing did not stop at mathematical machines. He went on during the second World War to lead the successful effort to decrypt the German cryptographic machine, the Enigma, and afterwards to build several of the world’s first actual computers. (The end of his life was marred by — let us be polite — insufficient recognition of his achievements by the authorities of his country.)

Alan Turing introduced many of the seminal ideas of computing science. The highest distinction in the field, the Turing Award, honors his memory.

The undecidability of the Halting Problem belongs to a series of *negative* results that burst into one science after another in the early 1900s, crashing the great science party that the new century had appeared to herald:

- Mathematicians saw the validity of set theory — and, as it turned out, of the basic techniques of logical reasoning — put into question by the emergence of apparent *paradoxes*; then just as an enormous 30-year effort to repair the foundations, by such mathematicians as Bertrand Russell and David Hilbert, seemed to have a chance of succeeding, Kurt Gödel proved that in any axiomatic system powerful enough to describe ordinary mathematics there will be properties that can be neither proved nor disproved. This **incompleteness** theorem is one of the most striking examples of the limitations on our ability to *reason*.
- At about the same time, physics had to accept the Heisenberg uncertainty principle and other results of quantum mechanics that put limits on our ability to *observe*.

Undecidability results, for the Halting Problem in particular, are the computing science version of such seemingly absolute limitations.

The theoretical undecidability of the Halting Problem should not directly affect — except for the emotional trauma of coming to terms with our intellectual limitations, but I trust you will recover — your practice of programming. Yet non-terminating programs are not just a theoretical possibility but a very real threat. To avoid its unpleasant occurrence, the advice is clear:

> ***Touch of Methodology*:**
> **Loop termination**
>
> Whenever you write a loop, examine the question of its termination. Convince yourself — by identifying a suitable variant — that it will always execute a finite number of iterations. If you can't, rework the loop until you can equip it with a variant.

Animating a metro line

As a simple loop example, we come back to the problem sketched at the beginning of this section: "animating" Line 8 by having a red dot move through its stations. We may use:

← Page 153.

- From class *STATION*, a query *location*, indicating the station's place on the map; the result is of type *POINT*, representing the notion of point in a 2-dimensional space.
- A command *show_spot* from class *TOURISM*; *show_spot* (*p*), for *p* of type *POINT*, will display a red spot at location *p*.
- *Spot_time*, also from *TOURISM*, a predefined value for the time to leave the red spot on each station; it is set to 0.5 seconds.

← Remember that the actual class names are TRAFFIC_STATION and TRAFFIC_POINT; see "Convention: Traffic library class names", page 53.

The task of the loop will be to call *show_spot* at the location of every station of the line, in sequence.

To get to the successive stations we could (with the help of operations on variables studied in the chapter after next) use the query *i_th* which gives us the *i*-th element of a line, through the call *some_line*.*i_th* (*i*), for any applicable *i*; the loop would have to perform

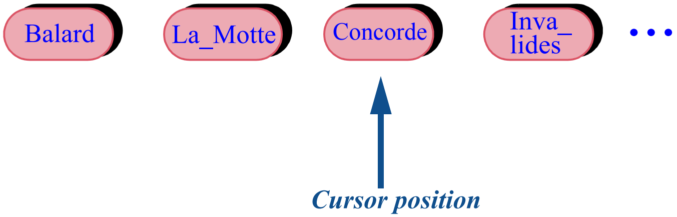
→ Assignment: chapter 9.

```
show_spot (Line8.i_th (i ).location)
```

for successive values of *i*, ranging from 1 to *Line8*.*count*. Let us instead use this opportunity to discover a typical form of loop used for *iterating* over object structures such as lists. To "iterate" over a data structure is to perform an operation on each one of its elements, or on a subset of its elements selected by an explicit criterion. Here the operation consists of calling *show_spot* on the location of each selected station.

→ Chapter 13 has more on iteration, particularly "Iterating on data structures", 13.13, page 431

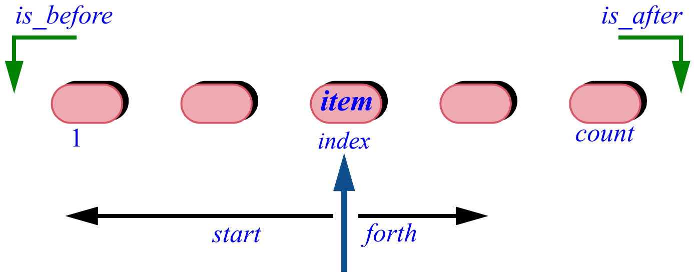
Classes such as *LINE*, and in general classes describing ordered lists of objects, support iteration by letting you move a **cursor** (a marker) to successive places in the list. The cursor does not have to be an actual object, simply an abstract notion denoting, at any point in time, a position in the list:



A list and its cursor

In the state shown, the cursor is on the third station. *LINE* and other list classes include the following four key features — two commands and two queries — for iterating over the corresponding object structures:

- The command *start*, which brings the cursor to the first item. (An "item" is an element of the list, in this example a metro station.)
- The command *forth*, which advances the cursor to the next item.
- The query *item*, which yields the item, if any, at cursor position.
- The boolean query *is_after*, yielding **True** if and only if the cursor is at the extreme right, past the last element if any. For symmetry there's also *is_before*, although we do not need it now.



Basic list features

Also useful is the query *index* which, as illustrated, gives the index of the current cursor position. It is 1 for the first item and *count* for the last.

This is enough to give us the general iteration scheme for lists, and its application to our example:

```
from
    Line8.start
invariant
    -- "For all stations before cursor position, a spot has been displayed"
    -- "More invariant clauses (see below)"
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
variant
    Line8.count – Line8.index + 1
end
```

This scheme using *start*, *forth*, *item* and *is_after* to iterate over a list occurs so frequently that you must make sure to understand its details and convince yourself of its correctness. Informally, its effect in this example is clear:

- Bring the cursor to the first item of the list, if any, through the call to *start* in the initialization.
- At each step through the loop, display for *Spot_time* seconds a spot on the station *Line8.item* at cursor position.
- Also at each step, after displaying the spot, advance the cursor by one position, through *forth*.
- Stop when the cursor *is_after*, that is to say, past the last item.

To avoid any confusion (I hope the previous discussion does not leave room for any, but just in case): there is no connection between the position of a station on the map or, as we have called it, its **location**, and the notion of **cursor position**:

- A station has a geographical location in the city, determined by its coordinates.
- The cursor exists only in our imagination and in our program, not in the world out there. It is an internal marker enabling the program to iterate over a list of stations, remembering from one iteration step to the next which item it last visited.

Programming time!
Terminating and non-terminating loop

Update the loop in feature *traverse* of class *ROUTES* to read as the last version, with the variant and (informal) invariant. Run it.

Now remove the instruction *Line8.forth*, introducing an error. Run the system again and observe what happens.

(Then restore the missing line for future exercises.)

To stop execution from EiffelStudio, you have a button at your disposal.

Let us now consider the loop constituents in more detail. The initialization uses *start* to bring the cursor to the first position. In the Contract View of class *LINE* (and of any similar class based on the notion of list) you may see that the specification of *start* reads:

```
start
        -- Bring cursor to first element
        -- (No effect if empty)
    ensure
        at_first: (not is_empty) implies (index = 1)
        empty_convention: is_empty implies is_after
```

The boolean query *is_empty* indicates whether the list is empty. For the moment, consider only the case of a non-empty list (like *Line8*). The first postcondition clause at_first of *start* indicates that after initialization the cursor is on the first element (*index* = 1), as we would expect.

The loop's exit condition is *Line8*.*is_after*. For a non-empty list it will not hold after initialization; you can in fact check this through the clause in the class invariant that reads

```
is_after = (index = count + 1)
```

Since this is an equality (equivalence) between two boolean values, it means that *is_after* is true if and only if *index* = *count* + 1; for a non-empty list *count* will be at least 1, so after the initialization, when *index* = 1, it is impossible for *is_after* to hold. In this case the loop body will be executed at least once.

Each execution of the loop body performs

```
show_spot (Line8.item .location)
```

which displays a red spot at the geographical *location* of the item at the current cursor position in the list.

Understanding and verifying the loop

Let us gain a deeper understanding of our loop example by verifying that it is correct — that is to say, performs as expected in all cases. It is a good idea, as you read, to use the debugger to examine the objects involved at various stages of the execution.

Programming time!
Using the debugger

As you read through the complementary explanations of this example, and in particular its correctness arguments, it is useful to get a concrete picture by following what happens at run time. The EiffelStudio **debugger** provides this capability. Use it to execute the program as a whole, or instruction by instruction in the feature *traverse* of class *ROUTES*, and to stop it at any time, then traverse the object structure and examine the contents of relevant objects.

For example you can see the instance of *LINE* and check that the results of queries such as *is_before* and *is_after* agree with the expected values as deduced from the analysis of the program carried out below.

Such a run-time inspection tool is not a substitute for systematic reasoning about programs. Reasoning yields the properties that will hold in all executions of the program; run-time inspection can only tell you that a particular property holds at one point of one execution. But it is still very helpful as a way to gain a practical understanding of what is going on; it lets you, literally, *see* your program as it executes.

As its name indicates, the debugger helps, when a program does not function as expected, to find out what the error — the *bug* — is. But its scope is broader; bug or no bug, it gives you a direct window into program execution. Do not wait until something goes wrong to take advantage of it.

A section of the EiffelStudio appendix (actually a link to an online document) tells you how to run the debugger to examine the execution of a program.

→ *"Controlling execution and inspecting objects", E.6, page 846.*

For ease of reference here is the loop again:

```
from                                                              [5]
    Line8.start
invariant
    -- "For all stations before cursor position, a spot has been displayed"
    -- "More invariant clauses (see below)"
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
variant
    Line8.count – Line8.index + 1
end
```

We may deal first with the case of an empty list. As noted, the postcondition of the command *start* reads

```
ensure
    at_first: (not is_empty) implies (index = 1)
    empty_convention: is_empty implies is_after
```

so that — by "convention" — an empty list will, after a call to *start*, satisfy *is_after*. Of course this convention is not there by accident, but intended precisely to ensure that the typical iteration scheme on a list, using the form illustrated by our example — start with *start*, exit on *is_after*, and each time through the loop do something with *item* and then move on with *forth* — stops immediately, having produced no visible effect, when applied to an empty list. This is indeed the case for our loop.

Touch of Methodology: Beware of the border cases!

Extreme cases, such as an empty list, are a frequent source of errors. It is all too easy, when you design your program, to think only of non-empty cases (and *test* it only on those). Then once in a while the execution of the program might use an empty structure, and fail. The issue arises not only for empty structures but for extreme cases in general. Another example is a structure sized for a limited number of items only: it might cause problems when *full*.

When designing a program and reasoning about its correctness, make sure to think of the extreme cases, and to verify that your reasoning holds for these cases as well as the more ordinary ones. This also applies to testing: always include extreme cases in your program tests.

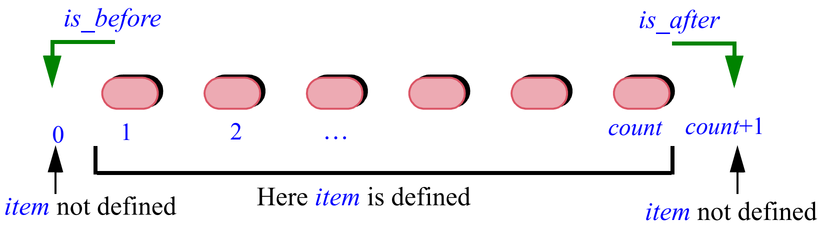
You may run the example on an empty line (class *TOURISM* defines a feature *Empty_line* for that purpose) and use the debugger to follow what happens.

So the loop does the right thing in the case of an empty list. For the rest of this discussion of correctness we assume that the list is not empty.

Bring up the specification for *item*. You will see that it has a precondition, stating that the query is only applicable if the cursor is on a list element:

```
item: STATION
        -- Current item
    require
        not_off: not (is_after or is_before)
```

as suggested by the following figure:



Where a list item exists

Since the loop body calls *item* — in the call to *show_spot* — we must verify that prior to the execution of this call the precondition will always hold.

Note first that the exit condition is **not** *is_after*, so *is_after* is not true when *show_spot* is called (if it were, the loop body would not be executed). Next, *is_before* must also not be true. To check this, we may add the following property to the loop invariant:

```
not_before_unless_empty: (not is_empty) implies (not is_before)        [6]
```

Let us check that this is indeed a loop-invariant property. As noted, we only consider non-empty lists (if *is_empty* is true, [6] trivially holds), so we only need to check that **not** *is_before* satisfies loop invariance. We note in the *class* invariant that

```
is_before = (index = 0)
index >= 0
index <= count + 1
```

In other words, *is_before* is true if and only if the cursor's *index* position is zero. After initialization, the postcondition of *start* — clause at_first as given above — indicates that *index* is one, so *is_before* is **False** and **not** *is_before* holds. Next we must check that every execution of the loop body preserves **not** *is_before*. The specification of *forth* reads

← *Page 169*.

```
forth
        -- Move cursor to next position
    require
        not_after: not is_after
    ensure
        moved_forth: index = old index + 1
```

Since *index* is never negative and has been increased by one, it cannot be zero, therefore *is_before* cannot hold. So [6] is indeed a loop-invariant property.

You should track the properties just seen on an actual execution of the loop; use the debugger to execute the loop iterations one by one, and explore the object structure at each step.

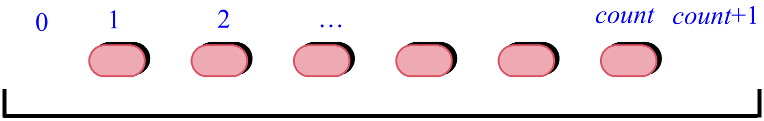
The cursor and where it will go

To complete our understanding of loops and of this example, it is useful to check a little further into the class invariant of *LINE*. If you bring it up you will see the following two clauses, also appearing in all the library classes having to do with list structures of any kind:

```
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

This expresses, as illustrated below, that we allow the cursor to be:

- On an item if any (if the list is empty there are no items)
- One position left of the first item, but no further to the left.
- One position right of the last, but no further right.



Possible cursor positions

Permitted cursor positions

Being able to go off by one position is useful for the general loop scheme illustrated by our spot-moving example:

```
from
    some_list.start
invariant
    -- "All items left of cursor, if any, have been processed"
until
    some_list.is_after
loop
    -- "Process item at cursor position"
    some_list.forth
variant
    some_list.count – some_list.index + 1
end
```

After the loop has processed the last item, the highlighted call to *forth* will move past that item. This will cause *is_after* to be true, so that there will be no further iteration; but it is essential that the call to *forth* be possible even though it leads to a position (at *count* + 1) where there is no list item. The invariant permits this; it is matched by the precondition of *forth*, cited above:

```
    require
        not_after: not is_after
```

This observation concludes our review of the essential properties of loops and of how to ensure that loops are correct.

7.6 CONDITIONAL INSTRUCTIONS

The next control structure, the conditional instruction, does not raise as many issues as the loop, but is also a fundamental program building block.

A conditional instruction (or just "conditional") involves a condition and (in the basic form) two instructions; it will execute one of these instructions if the condition holds, the other one if not.

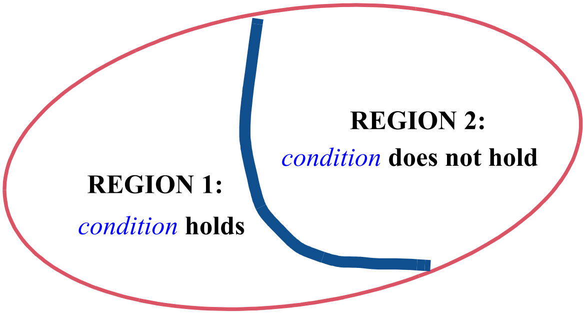
As a problem-solving technique, the conditional corresponds to *separating cases*: divide the problem space into two (or more) parts such that it is easier to solve the problem separately in each part. For example, when trying to get from the Eiffel Tower to the Louvre:

- *If* the weather is good and you are not too tired, *then* walk to the nearest station and take the metro.
- *Else* try to catch a taxi.

Or, in elementary mathematics, if you are asked for real roots of the quadratic equation $ax^2 + bx + c = 0$:

- *If* the discriminant δ, defined as $b^2 - 4\,ac$, is positive, *then* you can derive the two solutions $(-b \pm \sqrt{\delta})\,/\,2a$.
- *Else*, *if* δ is zero, *then* you can derive the single solution $-b\,/\,2a$.
- *Else*, there is no real solution (only complex ones).

You may picture the use of a conditional, as a problem-solving strategy, in terms of a partition of the set of cases to handle:



Conditional as a partition of the problem space

You have found a partition of the problem space into two parts, characterized by a certain *condition* that holds on one and not in the other, such that it is easier to find a separate solution for each part than to find a global solution directly. The basic form of the construct will be:

```
if condition then
    "Produce Region 1 solution"
else
    "Produce Region 2 solution
end
```

which we will soon generalize to partitions involving more than two cases.

Conditional: an example

As a typical example of conditional instruction, we may adapt our last loop example. The loop was displaying a red spot on each station. We refine this by stopping a little longer, with a yellow spot that blinks, on exchanges. Class *TOURISM* obligingly provides for that purpose a command *show_blinking_spot*, complementing *show_stop* used so far.

← *[5], page 170.*

We can achieve the result through this variation of the previous loop, where the only change is the highlighted part:

```
from                                                              [7]
    Line8.start
invariant
    not_before_unless_empty: (not is_empty) implies (not is_before)
    -- "For all stations before cursor position, a spot has been displayed"
until
    Line8.is_after
loop
    if Line8.item.is_exchange then
        show_blinking_spot (Line8.item.location)
    else
        show_spot (Line8.item.location)
    end
    Line8.forth
variant
    Line8.count – Line8.index + 1
end
```

The example conditional instruction uses three times the expression *Line8.item*, a query call. It is more elegant to compute the result once, give it a name, and then reuse that name whenever needed. We will soon learn how to do this.

→ *Assignment: chapter 9.*

Programming time!
Using a conditional

Update the preceding example — feature *traverse* in class *ROUTES* — to take into account the conditional instruction above. Run the result.

For the conditional instruction we need no less than four new keywords: **if**, **then** and **else**, as well as **elseif** which will appear next. The basic structure is straightforward:

```
if condition then
    Compound_1
else
    Compound_2
end
```

where *condition* is a boolean expression and *Compound_1* and *Compound_2* are compound instructions — sequences of zero or more instructions.

← *"Sequence (compound instruction)", 7.4, page 147.*

Conditional structure and variations

Being sequences of *zero* or more instructions, both *Compound_1* and *Compound_2* may be empty, so that you may write

Not the recommended style (see next).

```
if condition then
    Compound_1
 else
                  ◄─────────── Nothing here
end
```

with nothing in the **else** part. This corresponds to the frequent case of an instruction or sequence of instructions that you want to execute only if a certain condition holds, doing nothing otherwise. Rather than including an **else** clause with no instructions, you may in this case omit the clause altogether. You will just write:

Recommended style.

```
if condition then
    Compound_1      ◄─────────── No else clause
end
```

In either form — with or without an **else** clause — any of the instructions making up the compounds can itself be a control structure, for example a loop or another conditional.

Assume for example that you want to do something different — yet — for a metro station that connects to the railway network. You may use this scheme as a replacement for the previous loop:

```
from … invariant … until … loop                                        [8]
                    -- The omitted loop clauses are as in [7] above
    if Line8.item.is_exchange then
        show_blinking_spot (Line8.item.location)
    else
        if Line8.item.is_railway_connection then
            show_big_blue_spot (Line8.item.location)
        else
            show_spot (Line8.item.location)
        end
    end

    Line8.forth
variant … end
```

Not the recommended style; see [11], page 180.

Such inclusion of program structures within others is, as you know, called **nesting**. Here we have a conditional instruction nested in another conditional instruction, itself nested in a loop.

← *“Nesting and the syntax structure”, 3.5, page 40.*

Touch of Style: How deep a nest?

There are no theoretical limits on how deeply you may nest control structures. The limits are practical: good taste, and the desire to keep your programs readable.

The last example [8] has a depth of *four*: basic instructions appearing within a control structure, itself within a structure, itself within another. It is about the maximum that you should use in ordinary programming. This is not an absolute rule: some algorithms genuinely require a higher depth of nesting. But when you reach such a level you should ask yourself whether you can avoid the extra nesting.

The alternative is usually to carve out a significant part of the structure and give it an independent status as a *routine*, replacing its original occurrence by a *call* to that routine. We will meet routines soon.

→ *Chapter 8; see in particular “Encapsulating a functional abstraction”, 8.3, page 214.*

In examples such as the last one [8], the depth of nesting makes the structure appear more complex than it needs to be, and we can simplify it without recourse to routines. This simplification is applicable to conditionals repeatedly nested in the **else** part of other conditionals:

```
if condition1 then                                                      [9]
    …
else
    if condition2 then
    …
    else
        if condition3 then
            …
        else
            …
            … More nested occurrences of if … then… else… end …
            …
        end
    end
end
```

In this structure the nesting gives a deceptive impression of complexity, whereas in fact the decision structure is sequential:

- If *$condition_1$* holds, execute the first **then** part, and nothing else.
- For $i > 1$, if *$condition_i$* holds but none of the *$condition_j$* does for $j < i$, execute the i-th **then** part, and nothing else.
- If no *$condition_i$* holds, execute the innermost **else** part, and nothing else.

The keyword **elseif** enables you to remove the unnecessary nesting in this case by writing the successive cases at the same level:

```
if condition1 then                                                      [10]
    …
elseif condition2 then
    …
elseif condition3 then
    …
elseif … More conditions if needed … then
    …

else        -- As before, the else part is optional
    …
end
```

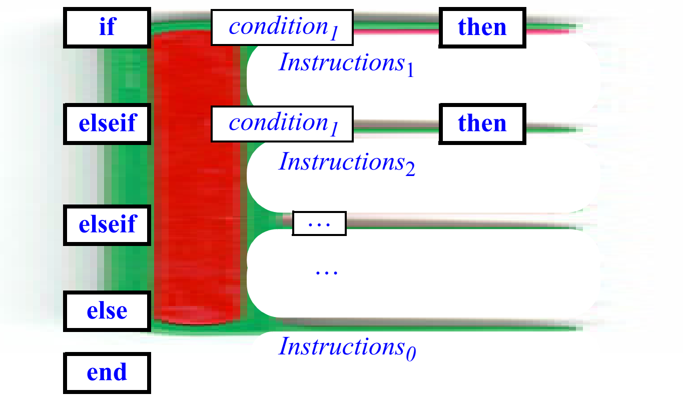
This replaces a *Matryoshka*-like structure

***Matryoshki* (*Russian dolls*)**

if c_1 then
...
else
if c_2 then
...
else
...
end
end

by a comb-like structure, less ambitious but easier to understand:

Comb-like structure



The keyword is **elseif** as a single word, not to be confused with **else** followed by **if** — two keywords — as used in the previous form, which calls for more nesting since every **if** must have its very own matching **then** and **end**.

A piece of trivia, useful in TV contests and in cocktail parties when the conversation dries up: **elseif** is the only Eiffel keyword made of two English words. Every other reserved word of the language is made of a single English word, unabbreviated, and chosen from everyday vocabulary. “Else if” is a simple notion, but no single English word exists to describe it.

With **elseif** we may rewrite the last metro line example [8] as a single conditional instruction with no further nesting:

```
from … invariant … until … loop                                    [11]
                    -- Omitted loop clauses as in [7]
    if Line8.item.is_exchange then
        show_blinking_spot (Line8.item.location)
    elseif Line8.item.is_railway_connection then
        show_spot (Line8.item.location)
    else
        show_spot (Line8.item.location)
    end
    Line8.forth
variant … end
```

→ We will obtain a better version, avoiding the repetition of Line8.item, in "Encapsulating a functional abstraction", 8.3, page 214.

Conditional: syntax

Here is a summary of the form of conditional instructions:

***Syntax*:**
Conditional

A conditional instruction consists, in order, of:

- An "If part", of the form **if** *condition*.
- A "Then part" of the form **then** *compound*.
- Zero or more "Else_if parts", each of the form
  elseif *condition* **then** *compound*.
- Zero or one "Else part" of the form **else** *compound*
- The keyword **end**.

Here each *condition* is a boolean expression, and each *compound* is a compound instruction.

If, by the way, you find this form of syntax description too verbose and at the same time not rigorous enough (for example we have to understand that each *condition* denotes a distinct boolean expression), you are right. The better technique for describing such non-trivial syntax constructs — such as the control structures of this chapter — is a mathematical notation known as BNF. We will learn it in the chapter on syntax. The informal specifications of the present chapter, aided by examples, will suffice in the meantime.

→ Chapter 11.

Conditional: semantics

The effect of the conditional instruction reflects the preceding discussions.

> ***Semantics*:**
> **Conditional**
>
> The execution of a conditional instruction consists of executing at most one of the compound instructions appearing in its "Then part", "Else If" parts if any and "Else part" if any, determined as follows:
>
> - If the condition following **if** has value **True**: the compound in the Then part.
> - If that condition has value **False**: the first compound in an Else_if part, if any, such that the corresponding condition has value **True**.
> - If none of the above applies and there is an Else part: its compound.
> - If none of the above applies: no compound (the conditional has no effect).

Conditional: correctness

The correctness of a conditional instruction is the separate correctness of both of its branches under the respective assumptions that the condition holds and that it does not:

> ***Correctness*:**
> **Conditional instruction**
>
> For a conditional instruction **if** *c* **then** *a* **else** *b* **end** to be correct, the program must ensure that prior to the conditional's execution:
>
> - If *c* holds, the precondition of *a* holds.
> - If *c* does not hold, the precondition of *b* holds.
>
> The postconditions of *a* and *b* — each executed under these conditions — must imply the postcondition desired for the conditional instruction.

You may easily generalize this rule, given here for the basic **if** … **then** … **else** … form, to the full form with **elseif** clauses.

7.7 THE LOWER LEVEL: BRANCHING INSTRUCTIONS

The combination of our three fundamental mechanisms — sequence, loop and conditional — provides the appropriate basis (when complemented by routines) to build the control structures that we need to write our programs.

These programming-language mechanisms have counterparts in machine code, which in most computer architectures are far more rudimentary. Normally you will not use them directly, as compilers are responsible for the mapping between the two levels. It is important, however, to be aware of the mechanisms actually available in the hardware to handle your programs' final flow of control.

Conditional and unconditional branching

Machine-language control mechanisms typically include:

- **Unconditional branch**: an instruction that transfers control to the instruction found at a given location in memory. In the example below, this instruction will appear as **BR** *Address* where *Address* is the location of the target instruction.

- **Conditional branch**: transfer control to a specified location if two specified values are equal, otherwise proceed to the next instruction. We may write it **BEQ** *Value1 Value2 Address*. The name stands for "**B**ranch if **EQ**ual".

The notion of "instruction found at a given location in memory" follows from the principle of the stored-program computer: the program, along with data, resides in memory. A branch instruction that mistakenly uses an address that does not hold an instruction would cause an abnormal condition, typically caught by the hardware and leading to program termination.

← *"The stored-program computer", page 10.*

With these hardware-level branching mechanisms the equivalent of

```
if a = b then
    Compound_1
else
    Compount_2
end
```

looks like this:

A conditional in machine code

```
100     BEQ loc_a loc_b 104
101     … Code for Compound_2
102         …
103     BR 106
104     … Code for Compound_1 …
105         …
106     … Code for continuation of program …
```

Here *loc_a* and *loc_b* stand for memory locations holding the values of *a* and *b* The numbers on the left are instruction locations; starting at 100 is just an example, and so are the numbers that the example uses for the locations of successive instructions. Determining the exact space taken up by machine instructions associated with every program element, and laying out everything in memory, can be a tricky task; since almost no one writes application at the machine-language level, this task is the responsibility of compilers (that is to say of compiler writers) rather than application programmers.

→ *Values on which machine instructions operate directly are usually held in special locations called* **registers**. *See "Registers and the memory hierarchy", page 287.*

From this *conditional* example, you can infer the code structure that a compiler would generate for a *loop*. This is the subject of an exercise.

→ *7-E.3, page 208.*

The goto instruction

Branching instructions, conditional and non-conditional, reflect basic operations that computers are able to perform: test certain boolean conditions such as the equality of two values held in memory; transfer control to an instruction stored at a specified location. So it is natural that we should find these instructions in machine code. But they were not always confined there. All programming languages used to have, and many still offer, a **goto** instruction, whose name comes from "go to" written as a single word. In such languages you may give a **label** to any instruction, as in

```
some_label: some_instruction
```

where some_label is a name — an identifier — of your choice. It is common for such languages to use a colon **:** between the label and the instruction it labels, but other conventions are possible. These labels correspond to the location numbers (100, 101, …) appearing in our machine-language example, but they are chosen by the programmer, who lets the compiler map them to memory locations. The language then includes an instruction of the form

```
goto label
```

whose effect is to transfer control — which would otherwise flow to the instruction appearing next — to the instruction with the given label.

The keyword is usually **if** *in such languages;* **test** *is used here to avoid confusion with the full-blown conditional instruction.*

Instead of an **if** *condition* **then** *Compound_1* **else** *Compound_2* **end** conditional, older programming languages have a more primitive choice instruction **test** *condition simple_instruction*, which executes the *simple_instruction* if the *condition* is true, otherwise proceeds sequentially. This closely reflects machine-level instructions such as **BEQ**. To express the equivalent of the conditional in such a language you would write:

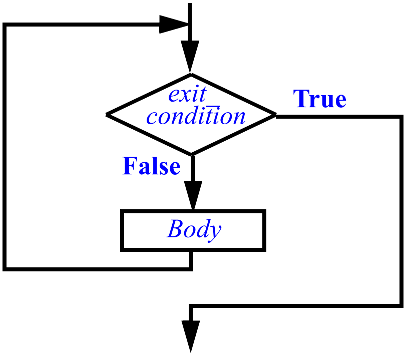
```
            test condition goto else_part
            Compound_2
            goto continue
else_part:  Compound_1
continue:   … Continuation of program…
```

This is less clear than the conditional instruction, with its hierarchical, symmetric structure.

The comparison is even less favorable for a loop which, ignoring the **from** part, would be represented as:

```
start:      test exit_condition goto continue
            Body
            goto start
continue:   … Continuation of program…
```

with its control flow involving two **goto** branches going in opposite directions:



Flowchart for a loop

Flowcharts

The last figure uses a representation of the control flow called a **program flowchart** or just flowchart. The box shapes are standardized: diamond for a test node, here with two outgoing branches for **True** and **False**; rectangle for a processing block, here for the *Body*. You may check your understanding of the concept by drawing a flowchart for the Conditional construct.

→ Exercise 7-E.4, page 209.

Flowcharts used to be a popular way of expressing the control structure of a program. Nowadays you may still encounter them in descriptions of non-software processes, but for programming they have fallen into disrepute (to the point that some authors call them "*flaw* charts"). It is easy to understand why. When programming languages gave you, as control structures, the unconditional **goto** and a conditional branching instruction such as **test** *condition* **goto** *label*, flowcharts provided a welcome higher-level view of the run-time flow of control, clearer than what could be inferred from reading the program text with its succession of branching and non-branching instructions. But this is obsolete for two reasons:

- Our programs do more complicated things. We nest compounds within loops within conditionals; big flowcharts quickly become messy.
- The mechanisms of this chapter — compound, loop, conditional — provide a higher form of expression for the control structure. A neatly formatted program text, with indentation clearly reflecting the nesting, carries a better representation of the run-time scheduling of instructions.

The move from flowcharts to carefully chosen and properly nested control structures belies the cliché that "a picture is worth a thousand words". In software we need many thousands or indeed millions of "words", but it is critical that they be the *right* words. For precise, unambiguous descriptions pictures lose their appeal.

> The correctness of a program may depend on fine details such as using a condition $i <= n$ rather than $i < n$; the best pictures in the world are largely helpless when it comes to getting such aspects right.

7.8 GOTO ELIMINATION AND STRUCTURED PROGRAMMING

Flowcharts are not the only casualty of the reexamination of control structure specification which occurred as software engineering was growing into a mature discipline: the **goto** instruction also lost favor.

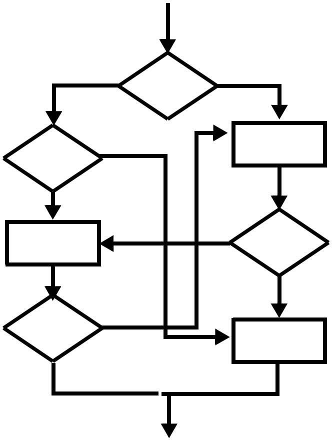
Goto harmful?

The reasons for distrusting the **goto** are pretty much the same as those behind the demise of flowcharts. The mechanisms that we have studied offer better control over execution. This comment actually contains two separate arguments:

- The first observation is that loop and conditional constructs (Compound does not express explicit transfer of control) are more readable — especially for complex structures that nest several such constructs within each other — than **goto**. This does not take much convincing; a simple look at the original structures and their **goto** variants suffices.

- That is not, however, the full story. By sticking to the three mechanisms listed, we are restricting ourselves as compared to a programmer who would be using arbitrary **goto** instructions — or, equivalently, arbitrary flowcharts with arrows, possibly crossing each other, from any decision box to any other box. The nickname for such contorted control structures is *spaghetti bowl*; the figure below shows an example, still small. The high-level control structures are clearly better for program readability, but that is only a methodological argument. Could it be that by restricting ourselves to our three control structures and excluding the **goto** we lose something essential? In other words, are there important algorithms that one cannot express without full **goto** power?

The implied slander on one of humankind's most creative culinary inventions is regrettable. On the other hand, real programmers mostly run on cold pizza.



Spaghetti bowl

The answer, remarkably enough, is **no**. A theorem proved in 1966 by two Italian computer scientists, Corrado Böhm and Giuseppe Jacopini, states that every flowchart of interest in the theory of computation has an equivalent expression using only sequences and loops (the conditional is not even needed).

Corrado Böhm and Giuseppe Jacopini: Flow diagrams, Turing machines and languages with only two formation rules. Comm. of the ACM, vol. 9, no. 5, pages 366-371, May 1966. (Requires computation theory background.)

The general transformation rules from arbitrary flowcharts to **goto**-less programs, as derived from their article, can be complex. For cases arising in practice, it is often possible to remove the **goto**s through an informal and fairly straightforward process. As this is a more specialized topic (and uses concepts such as assignment that we have not yet seen formally), its discussion on a specific example appears in an appendix at the end of this chapter. An exercise, which assumes you have read the appendix, asks you to produce a goto-less equivalent of another example structure.

→ "Appendix: an example of goto removal", 7.11, page 205. For another example do exercise 7-E.4, page 209.

Avoiding the goto

The reason we can relegate **goto** removal to an appendix is that it is not a task you will have to perform in ordinary circumstances. There is no need to use **goto**s and then remove them. You should build your program directly with the high-level control structures, which have amply proved their adequacy to express algorithms, simple and complex, in a clear way.

Touch of History: Quashing the goto

Today "Go to" is almost a dirty word in programming, but that was not always so. Once upon a time, branch instructions were the basic control structure. And then without warning appeared in the *Communications of the ACM* of March 1968 — the year, throughout the Western world, of youthful questioning of the established order — an article entitled "*Goto considered harmful*" by Edsger W. Dijkstra. To avoid delaying its publication, the editor at the time, Niklaus Wirth, had decided to run it as a "Letter to the Editor". Through careful reasoning, Dijkstra argued that unrestricted branching was detrimental to program quality.

This led to the mother of all programming polemics — then as now, programmers do not like their habits questioned — which still resurfaces once in a while. But no one would seriously argue any more for unrestrained gotos.

→ See reference & URL in "Further reading", 7.12, page 207.

Dijkstra's short paper, which every programmer must read, explained the challenge that we face when devising a program:

Touch of the Masters: Dijkstra on the program and its execution

Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Edsger W. Dijkstra, 1968

Dijkstra

No one, then or later, has said this better. A program, even a simple one, is a static view of a wide range of possible dynamic computations, determined by the wide range of possible inputs. So wide indeed is the range — in many cases, potentially infinite — that we cannot even picture it to ourselves; but to ensure the correctness of our program we must somehow infer the dynamic properties from the static view. The discipline of using a nested structure of clear, well-understood mechanisms such as the sequence, the loop and the conditional helps; accepting the unrestricted **goto** would defeat this goal.

Structured programming

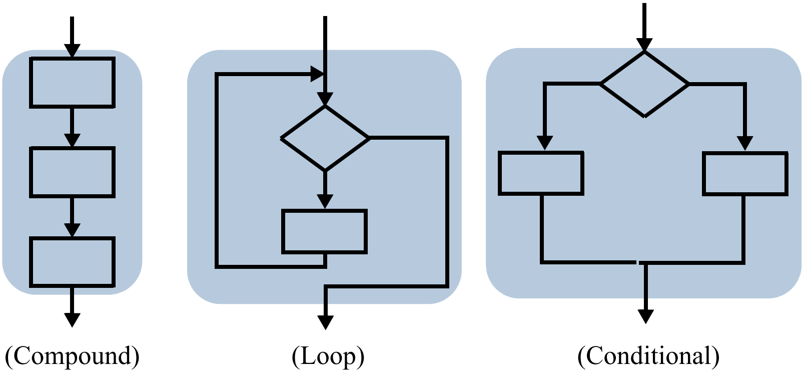
The revolution in views of programming started by Dijkstra's iconoclasm led to a movement known as **structured programming**, which advocated a systematic, rational approach to program construction. Structured programming is the basis for all that has been done since in programming methodology, including object-oriented programming.

As the first book on the topic shows, structured programming is about much more than control structures and the **goto**. Its principal message is that programming should be considered a scientific discipline based on mathematical rigor. (Dijkstra went further, describing programming as "*one of the most difficult branches of applied mathematics*".)

→ "Structured Programming", reference on page 207.

What stuck in the mind of the programming masses, however, is the elimination of the **goto** and the restriction of control structures to the three kinds seen in this chapter: sequence, loop and conditional, often called "*the control structures of structured programming*".

These control structures all have **one-entry, one-exit** flowcharts:



Three kinds of one-entry, one-exit structure

*The loop is shown without its initialization (**from** clause).*

In contrast, arbitrary control structures — see the blocks of our earlier spaghetti bowl — may have any number of entries and exits. Restricting ourselves to building blocks with one entry and one exit means that we can construct arbitrarily ambitious algorithms through three simple mechanisms:

← "Spaghetti bowl", page 186.

- **Serial connection**: use the exit of one unit as the entry of another, as an electrical engineer connects the output of a resistance to the input of a capacitor.
- **Nesting**: use a unit as one of the blocks within another.
- **Functional abstraction**: turn a unit, possibly with sub-units, into a routine, also characterized by one-entry, one-exit control flow.

→ Chapter 8 covers routines.

The Böhm-Jacopini theorem tells us that we are not losing any expressive power by limiting ourselves to these mechanisms. The gains in program simplicity and readability — and hence in guaranteeing that the programs are correct, extending them, reusing them — are considerable.

The goto puts on a mask

While few people would argue for a return to the general **goto**, the battle for simple control structures is not over. In particular, many programming languages support a form of loop that permits a "break" away from the middle. (There are also break instructions for "multi-branch" variants of the conditional, studied below.) The loop break instruction gives possibilities such as

```
from … until exit_condition loop
    Some_instructions
    if other_condition then break end
    Other_instructions
end
```

***Warning**: Illustration only. Not an Eiffel program text.*

meaning: if *other_condition* holds during an execution of the loop body, execution will terminate prematurely, skipping the *Other_instructions*, any further testing of *exit_condition* and any further iteration.

> Other constructs of a similar nature include an instruction **continue** that stops the current loop iteration to start the next one immediately.

→ See e.g. "Conditional and branching instructions", page 763 (about Java).

Such instructions are just the old **goto** in sheep's clothing. Treat them the same way as the original:

Touch of Methodology: Sticking to one-entry, one-exit building blocks

Stay away from any "break" or similar control mechanism.

It is easy to apply the advice to examples such as the above: just rewrite it as

← *“Flowchart for a loop”, page 184*.

```
from … until exit_condition loop
    Some_instructions
    if not other_condition then
        Other_instructions
    end
end
```

Other examples may require more rework but they do not affect the general rule.

The basic argument for that rule is the same one as against the general **goto**: the clarity and simplicity of one-entry, one-exit structures. There is also a fundamental criterion: our ability to reason about the semantics of programs; in Dijkstra’s terms, to “*shorten the conceptual gap between the static program and the dynamic process*”. With loops as we have seen them, a key technique for such reasoning is the **Loop Postcondition Principle**: to understand what a loop does, it suffices to combine its invariant (even if informal) with its exit condition. For example we devised the loop computing the maximum of a set of values to have the invariant

← *Page 160*.

“*max* is the maximum of $N_1, N_2, \ldots, N_i$”

and the exit condition

$i = n$

making it easy to ascertain the correctness of the loop by visual inspection: we simply check that the initialization succeeds in establishing the invariant, that the body succeeds in maintaining it, and that the loop terminates. The combination of these properties immediately implies that *max* is the maximum of $N_1, N_2, \ldots, N_n$. If we introduce **break** instructions or any other way to disrupt the basic control flow of the loop, such reasoning is no longer possible; the very notion of loop invariant goes away, at least in the simple, directly understandable form that we have studied. Another reason to stick to the one-entry, one-exit scheme.

The risk of misusing such goto-like constructs is not just theoretical. A major AT&T network failure in 1990, which cut off telephone service in the entire United States, was traced to a fault in the C code: a break that broke out of a switch but was intended for the enclosing structure.

See contemporary comments by Peter Neumann and others in Software Engineering Notes, tinyurl.com/d9j6dy, and Risks forum entries starting with www.risks.org/9.61.html#subj2.

7.9 VARIATIONS ON BASIC CONTROL STRUCTURES

Sequence, loop, conditional: the "structured programming" triad makes up the basis of structuring control flow. (By now you might have got the message.) They have some interesting variants, deserving a quick peek.

Since the Böhm-Jacopini theorem tells us that the triad is enough to express all meaningful algorithms, none of the extensions below is *theoretically* necessary; they can all be expressed, in a simple way, as combinations of sequences, loops and conditionals. But that does not disqualify them as useful tools for the programmer, since they may give us a more effective mode of expression in particular cases. Based on this criterion we may divide them into two categories:

1 Constructs that provide a welcome improvement over the basic ones, applicable to important practical cases.

2 Mechanisms that you need to know since they are present in some common programming languages, although no compelling argument exists for using them.

This difference is partly a matter of opinion, and you will be able to form your own.

Loop initialization

The **from** clause of our loop construct is a way to specify the control flow. It is of course redundant, since instead of

```
from
    Initialization_instructions
until condition loop
    Body
end
```

you may combine the "sequence" control structure with the loop, writing

```
Initialization_instructions
from
    -- Nothing here!
until condition loop
    Body
end
```

This achieves exactly the same effect. The loop constructs of some programming languages indeed start at the **until** or its local equivalent.

The rationale for including the **from** clause in the syntax is that most loop processes, like most approximation processes, need an initialization, without which the loop would not work properly. The initialization is not just a bunch of instructions that happen to be executed just before the loop, but an integral part of the loop. The loop correctness rules reflect this by assigning a precise role to the initialization: **ensuring the initial validity of the loop invariant** prior to any iterations of the loop body, which must then *preserve* that invariant.

← *In particular: "Correctness: Loop Invariant Principle", page 159.*

In languages whose loops do not have a **from** clause, you will write the initialization as a separate compound, perhaps with a comment explaining why it is there.

In Eiffel, this discussion gives us an answer to the question that you may have been asking yourself: if some operations are executed before a loop, should they appear in preceding instructions or in the loop's own **from** clause? Depending on the role of those operations, they might be in either place, or split across the two:

Touch of Methodology: Where to place pre-loop actions

If an instruction executed before a loop serves to initialize the loop process, in particular to establish its invariant, put it in the loop's **from** clause.

If it is part of a set of operations that simply happen to be executed before the loop in the algorithm of the enclosing routine, put it before the loop.

Other forms of loop

Many programming languages propose a form of loop, usually with the keyword **while**, highlighting the *continuation* rather than the exit condition:

```
while Continuation_condition loop
    Body
end
```

Warning: *Sample syntax; not valid Eiffel.*

Valid in Pascal if **loop** *is replaced by* **do**.

The semantics is: evaluate *Continuation_condition*; if it is false, do nothing; if it is true, execute *Body* and start again. This is equivalent, in our style, to using

```
until not Continuation_condition
```

or **until** *Exit_condition* where *Exit_condition* is the negation of *Continuation_condition*.

The difference is one of viewpoint:

- The **while** form emphasizes **execution**: it reflects that at run time the loop will execute its body as long as the *Continuation_condition* holds.
- The **until** form emphasizes **reasoning** about the program, its correctness and its effect: it reflects that the loop will yield a result that, together with the invariant property, satisfies *Exit_condition*.

Another from of loop should not be confused with **from**… **until**… **loop** … **end** even though it generally uses the keyword **until**, but at the end of the construct rather than the beginning. It typically appears as:

```
repeat
    Body
until
    Exit_condition
end
```

Warning: *Sample syntax; not valid Eiffel.*

This form is actually valid in Pascal.

The semantics is: execute *Body*; then if *Exit_condition* evaluates to true, stop; otherwise start again. Here the *Body* is always executed at least once, whereas the previous variants (**from** … **until** … and **while**) will not execute it at all if the exit condition is true (or the continuation condition false) on start.

Expressing the equivalent of a **repeat** loop in our notation is easy:

```
from
    Body
until
    Exit_condition
loop
    Body
end
```

This has the disadvantage of repeating the *Body*, whereas we should generally try to avoid code replication. We can minimize the replication by turning *Body*, if it includes several instructions, into a routine.

Here we reach the realm of opinions. Some people prefer the flexibility of having two constructs at their disposal: one for loops with zero or more iterations, another for the one-or-more case. I prefer to have a single loop construct, with its carefully defined semantics and its simple notion of invariant (whose counterpart in the **repeat** form is more complicated). I am willing to pay the occasional price of repeating a line of code or adding a routine.

The need is indeed occasional; in practice, programs need the zero-or-more kind of loop more often than the one-or-more variety.

Yet another common kind of loop, the "**for** loop", has a form such as

```
for i: 1 .. 10 loop
    Body
end
```

Warning: Sample syntax; not valid Eiffel.

with the semantics of executing *Body*, whose instructions generally use *i*, successively for all values of *i* in the given interval: here 1, then 2, and so on up to 10. The boundary values 1 and 10 are just an example, and generally you can choose variable values rather than constants.

In the C language and its successors such as C++, the form is

```
for (i=1; i <= 10; i++) {
    Body
}
```

Warning: This is C or C++ code, also valid in Java and C#. It uses the = symbol (:= in Eiffel) for assignment; see chapter 9.

The first element in parentheses is the initialization of *i*. The second one is the continuation condition. The last one is the incrementation operation to be performed after each execution of the *Body*; the notation *i*++ means "increase *i* by one". Ignoring the very visible differences of syntactical style — rather than keywords, C tends to use symbols such as braces, parentheses, semicolons — this implies a fine degree of control over the behavior at execution that is characteristic of this style of programming.

→ Appendix D presents the C language, based on appendix C presenting C++.

Loop forms relying on an explicit index are also known as "**do** loops" from the corresponding keyword in Fortran, which had a similar mechanism from early on.

The **from** construct of this chapter expresses such a loop as:

```
from
    i := 1
until
    i > n
loop
    Body
    i := i + 1
end
```

using the *assignment* instruction *a* := *b* ("Give to *a* the current value of *b*") studied in detail in a forthcoming chapter.

→ Chapter 9.

This is not the end of the story about the **for** style of loop. The **from** … **until** … **loop** equivalent does not do as good a job of immediately showing that the loop iterates over a certain interval, 1 **..** 10 in our example; that property is buried in the operations on *i*: initialization, test, incrementation. This is a strong argument for having a higher-level form of loop that simply prescribes: "**apply this operation to all elements of that set**".

The **for** style goes in this direction; here "that set" is a contiguous integer range. We need iterations on more general sets, for example lists such as a metro line seen as a list of stations; a general mechanism should enable us to ask, in high-level terms, "*apply this operation to all the stations on that line*".

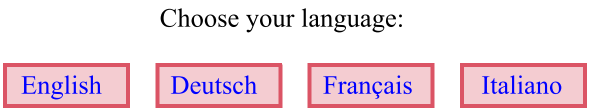
Such a mechanism has a name: **iterator**. In the discussion of data structures we will see that it is possible, without new control constructs, to define powerful iterators applicable to a wide range of object structures.

→ "Iterating on data structures", 13.13, page 431.

Multi-branch

Conditional instructions, as we have seen, solve a problem by partitioning the problem domain into disjoint subsets and using a separate solution for each of them. The basic **if** … **then** … **else** form uses two subsets; by including **elseif** clauses you may add any number of subsets to the partition.

Many programming languages offer another conditional construct for the case of a partition defined by a set of simple values such as integers or characters. A simple application is to let the user of an interactive program select a value from a set of visual alternatives:



Selecting from a list

Assume the graphics software enables you to get an integer *choice* that has value 1 to 4 depending on which of the four boxes the user has clicked. To process that answer you may use a standard conditional:

```
if choice = 1 then                                                    [12]
    … Bring up English interface …
elseif choice = 2 then
    …
elseif choice = 4 then
    … Bring up Italian interface …
else
    … This case should not occur (see below) …
end
```

A slightly more compact notation, “Multi-branch”, is available in this case:

```
inspect                                                               [13]
    choice
when 1 then
    … Bring up English interface …
when 2 then
    …
o
when 4 then
    … Bring up Italian interface …
else
    …
end
```

So far the simplification is modest, but you can see the general idea: if all the conditions used in a Conditional are of the form *choice* = val_i for the same expression *choice* and different constant values val_i — here 1, 2, 3 and 4 — you do not have to repeat the “*choice* =” part; instead, you simply list the constants in successive **when** clauses.

This is the Eiffel notation. Pascal and Ada have a similar construct with the **case** … **of** … keywords. C and its successors (C++, Java, C#) have a **switch** instruction, not exactly a Multi-branch but a **goto** with multiple targets depending on the value of an expression; it can be used to obtain the equivalent of a Multi-branch.

See for example the discussion of the C++ switch *in “Control structures”, page 833.*

Multi-branch is only available for expressions of specific types. In Eiffel you may use it for integers, as in this example, and characters. In both cases:

A few other types are also supported.

- There is a simple notation for the selection values (the val_i), such as 1 for an integer and 'A' for a character.
- The values are ordered. As a consequence it is possible to define intervals: of integers, such as 1 **..** 10; of characters, such as 'A' **..** 'Z'.

You may use the interval notation in a **when** clause, as in

```
inspect
    last_character  -- Representing for example a user-entered character
when ′a′ .. ′z′ then
    … Operations for a user entry that is a lower-case ASCII letter …
when ′A′ .. ′Z′ then
    … Operations for a user entry that is an upper-case ASCII letter …
when ′0′ .. ′9′ then
    … Operations for a user entry that is a digit …
else
    … Operations for a user entry that is not an ASCII letter or digit …
end
```

ASCII is the character subset covering basic English (no accents or other diacritics).

In a **when** clause you can also list several values or several intervals, or a mix, using commas as separators:

```
inspect                                                         [14]
    booking code    -- For an airline reservation
when ′A′, ′F′, ′P′, ′R′ then
    … Operations for a first-class booking …
when ′C′ .. ′D′, ′I′ .. ′J′, then
    … Operations for a business-class booking …
when ′B′, ′H′, ′K′ .. ′N′, ′Q′, ′S′ .. ′W′, ′Y′ then
    … Operations for an economy-class booking …
else
    … Handle the case of a non-standard booking code …
end
```

Information from "Travel class" article on Wikipedia, as of June 2008.

With these possibilities, the notational advantage over writing a plain Conditional becomes more significant. In addition, as you may have gathered from the examples, the Multi-branch enforces a **disjointness** rule: no value may appear (either explicitly, or as part of an interval) in two different **when** branches. Compilers check this, after expanding intervals if any, and will reject the program in the case of such an ambiguity.

This property sets the Multi-branch further apart from a Conditional: in **if** *c1* **then** … **elseif** *c2* **then** … **elseif** *c3* **then** … **end** it is possible for two or more of the conditions *c1*, *c2*, *c3*, … to hold; as the order of evaluation is explicitly sequential, the Conditional will execute the first branch whose condition is true. With a Multi-branch at most one of the conditions may hold.

The different cases of a Conditional are still mutually exclusive, in line with the "partitioning" strategy recalled at the beginning of this section: the i-th branch of a Conditional covers the case in which its i-th condition holds and none of the preceding conditions holds. This leads to disjoint cases, even if the conditions themselves are not disjoint. With a Multi-branch conditions are disjoint by design.

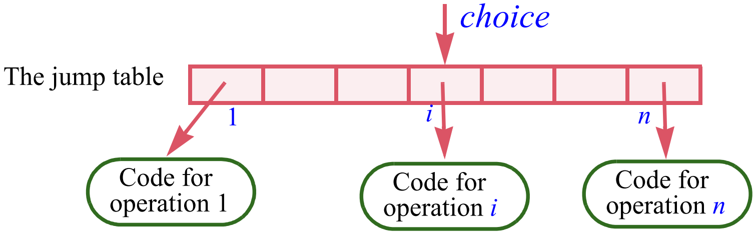
This semantic difference opens the way to a different implementation for Multi-branch, which may yield better run-time performance. Because conditions do not have to be evaluated in sequence, and thanks to the stored-program principle, a technique known as **jump tables** is available. More precisely it is available to compiler writers, so you generally do not need to use it directly, but it is useful to know the idea.

Assume a simple Multi-branch discriminating between consecutive integers, such as our first example using the values 1 to 4, extended for more realism to a larger number n of values. If the compiler implements this in the same way as a Conditional, the execution will perform successive comparisons: is *choice* equal to 1? If not, is it equal to 2? and so on — up to n comparisons in the worst case.

← *[13], page 196.*

Relying instead on the knowledge that the choice is between adjacent values in an integer interval, and that the various program elements to be executed for each case are stored in memory, we may introduce a data structure, the jump table, to find the right such element directly in each case:

Characters are also handled as integers, through their codes.



Using a jump table for Multi-branch implementation

The jump table is a set of contiguous locations in memory. (Soon we will study the corresponding high-level data structure: the *array*.) Each one of these locations contains the address of the code for the corresponding branch of the **inspect**, represented in the figure by an arrow pointing to a box denoting that code. (Again this has a counterpart in higher-level languages: the notion of *reference* or *pointer*.) Then the implementation of the Multi-branch, as translated into machine code by the compiler, is simple: use the value of the *choice* expression to find the corresponding location in the jump table; that location contains an address; execute the code at that address.

→ *"Arrays", 13.4, page 380.*

→ *"Reference assignment", 9.5, page 252.*

The important property of this scheme is that it does not require successive tests, only an access to a memory location based on an index (the value of *choice*) and an "indirection", that is to say, branching to an address obtained from a memory location. Both of these operations take constant time,

→ *The choice is "O (1)" rather than "O (n)"; see "Estimating algorithm complexity", 13.3, page 376.*

independent of the number of branches. Previewing techniques that we will encounter in the study of data structures, we see that the approach involves a *space-time tradeoff*: we hope to get faster execution (time) by sacrificing some memory (space) to store the jump table.

This example makes the jump table technique shine, especially for large *n*, since all values are in a contiguous integer interval. For more complex Multi-branch instructions with scattered values and intervals, such as the booking code example [14], the advantage over successive tests may not be so clear; it is the job of the compiler writer to devise the best combination of the two techniques for every particular case.

Touch of History: When implementation techniques influence language design

Historically, multi-branch instructions came from the jump table technique rather than the other way around. Early on, the Fortran language had a "Computed Goto" instruction of the form GOTO (L_1, …, L_n), *CHOICE* which branches to the label L_i if the integer expression *CHOICE* has value *i*. The switch instruction of C and its successors is similar. Such constructs directly reflects the jump table technique.

C.A.R. Hoare proposed a goto-free **case** instruction, included in Wirth's Algol W and Pascal languages; it is the origin of modern forms of Multi-branch.

Hoare (2007)

Any design for a Multi-branch must decide what to do if the expression's value matches none of the constants. Eiffel's **inspect** instruction may include, as you have seen, an **else** clause taking care of this case. The **else** clause is optional, as it is for Conditionals. If it is absent, the two instructions react differently:

- In the execution of a Conditional, if none of the conditions listed in an **if** or **elseif** holds and there is no **else** clause, the instruction does nothing.
- For an **inspect**, the instruction is considered erroneous and produces a run-time error (an *exception*).

→ Exceptions are studied next.

This policy may appear surprising at first; the rationale is that a Multi-branch explicitly lists a set of expected cases and their desired treatment. If other values are possible, you should write an **else** clause to deal with them. Not including this clause states your expectation that for any execution the expression will always have one of the listed values, such as 1, 2, 3 and 4 in the first example. If an execution violates this expectation, it is generally not the right idea to do nothing, as this can only lead to an incorrect computation and to other problems down the road. Better catch the anomaly closer to the source by triggering an exception.

← [13], page 196.

This rationale does not apply to the Conditional, which simply performs certain operations if certain conditions are satisfied, examining the various conditions in sequence.

7.10 AN INTRODUCTION TO EXCEPTION HANDLING

Happy families, as everyone knows from the first lines of *Anna Karenina*, are all alike; every unhappy family is unhappy in its own way. Happy program executions all use the same control structures; unhappy program executions are unhappy for many different reasons, called *exceptions*.

Exceptions complement the control structures of this chapter by providing a way to handle special cases without tampering with the default flow of control. As we will not need exceptions for the data structure and algorithm examples in this book — indeed they should be reserved for *exceptional* cases — this section will only introduce the basic ideas.

You may consider it as supplementary material and skip it (together with the last remaining section, an appendix) on first reading.

The role of exceptions

What is a "special" or "exceptional" case? We will see that it is possible to define this notion fairly precisely, but for now let us just rely on the intuition that it is an event that should not happen. Here are some of the kinds of event that can, each in its own way, shatter the bliss of a normal program execution:

- A division by zero, or some other arithmetic misfortune.
- An attempt to create an object after running out of memory.
- A void call, which we defined as an attempt to call a feature on an object that does not exist (*x*.*f* where *x* has a void value).

← *"The trouble with void references", page 112.*

- A contract violation, if you have enabled the mechanisms for monitoring preconditions, postconditions and invariants at run time.
- An execution of an instruction that signals an abnormal situation.

Such events will trigger an exception. In all cases but the last it will be a *system* exception, caused by external circumstances rather than explicitly by the program. The last example is a case of *programmer-defined* exception. This distinction illustrates the two possible roles of exception handling:

- You can use exception handling as a **technique of last resort** to handle unexpected events for which the normal control structures let you down. It is unrealistic to protect every division by a test that the denominator is not too close to zero, and every **create** instruction by a test that there is enough memory left. It would be even harder to plan for some of the other cases: a hardware failure or user interruption can happen at any time during execution. In this role, exceptions allow an entire program unit, for example a routine, to attempt recovery or at least graceful exit when any one of its instructions gets interrupted by an unexpected event. "Unexpected" means that the instruction has not set up its own protection against such events.
- The case of programmer-defined exceptions is different: here exception handling becomes a **control structure**, to be added to our previous catalog.

A precise framework to discuss failures and exceptions

To devise a proper strategy for using exceptions — in particular, define what is "normal" and "abnormal" — we may rely on the Design by Contract framework defined in previous discussions. We know that every routine, independently of its concrete implementation, has an abstract role defined by:

- A precondition: what it assumes from its callers.
- A postcondition: what it must deliver to its callers at the end of its execution.
- The invariant of the enclosing class: what properties it must maintain for the common benefit of all the routines of the class.

We may extend this rule to operations other than routines; a floating-point addition, for example, assumes that the operands are not too large, so that their sum can be approximately represented on the hardware; and it must in return deliver such an approximation. A **create** *x* instruction assumes that enough memory is available to allocate space for an object of the type of *x*; and it must in return update the memory with one more object, without affecting existing objects.

In a perfect world all families would be happy and all operations would meet their contracts. In the real world an operation will occasionally be unable to satisfy its contract. If you are the memory allocator and all available space has been used up, you cannot perform a **create**. If you are a routine and have been programmed incorrectly, you might not always satisfy your postcondition. These observations are sufficient to yield the precise definitions we seek:

Definitions: Failure, exception, recipient

A **failure** is the inability of an operation to fulfill its contract.

An **exception** is a failure of one of the operations in a routine execution. The routine is the **recipient** of the exception.

"Failure" is the primary notion and applies to operations of any kind: routines but also basic operations such as object creation. "Exception" is a derived notion and applies only to routines: a routine receives an exception if it executes an operation (a basic operation or a call to another routine) that fails.

Why is an exception not the same as a failure? Often it will cause its recipient to fail, but not always: this is where *exception handling* steps in. A routine may include "rescue code" that will attempt to recover from the exception and may be able to restart the execution on a better footing.

If a routine does not include exception handling, or includes it but is unable to recover, the routine execution will fail. In this case it triggers an exception in its caller, where the same two possibilities arise again: fail or recover. Exceptions, as a result, propagate up the call chain; if no routine along the way is able to recover, program execution will stop, an event known as “abnormal termination”. We saw an example, caused by a “call on void reference” exception.

← *See the figure “Abnormal termination”, page 113.*

We now know what failure means for routines: being the recipient of an exception, and not being able to recover from it.

Retrying

How is it possible for exception handling to “recover” from an exception? The idea is to have an alternate strategy, executing a new set of instructions if the original sequence has failed.

This subsection uses the concepts of variable and assignment, introduced in chapter 9; skip it on first reading if you are new to programming.

> Sometimes we might even reapply the *same* strategy, disregarding a possible accusation of insanity in the definition attributed to Einstein (“repeating the same thing and expecting different results”). Insane or not, this method can work in the case of exceptions caused by intermittent hardware failures.

To illustrate the idea, let us see how it works in the Eiffel exception mechanism, based on two keywords, **rescue** and **Retry**. The first introduces an optional clause in a routine, to be executed whenever an execution of that routine becomes the recipient of an exception. The second is a predefined boolean variable which, if true at the end of a **rescue**, will cause the body to be executed again; if **Retry** is false, the routine will fail. Here is an example:

Retry *is a keyword, denoting a local variable available in any routine. Like any boolean variable, it is initialized to false on routine entry.*

```
transmit (m: MESSAGE)
            -- Transmit m, if possible.
    local
        i: INTEGER
    do
        send_to_transmitter (m, i)
    rescue
        i := i + 1
        Retry := (i <= count)
    end
```

The assumption is that the call *send_to_transmitter* (*m*, *i*) attempts to send *m* through the *i*-th transmitter. We have *count* transmitters at our disposal; the ones with the lowest numbers are the fastest, so we want to try them first, but they have a higher likelihood of failing.

Like any other local variable of type *INTEGER*, *i* is initialized to zero on routine entry. If an exception occurs, as a result of a failed call to *send_to_transmitter*, the routine executes its **rescue** clause, which increases *i* by one. If more transmitters are available, **Retry** will be set to true; this causes a new execution of the routine body (the **do** clause) to try the next transmitter. If, however, *i* has passed *count*, the value of **Retry** is false and *transmit* fails, causing an exception in its caller.

This exception mechanism emphatically separates two roles:

E1 The normal routine body — the **do** clause — does not directly handle exceptions; it tries to fulfill the contract.

E2 The exception handler — the **rescue** clause — does not try to fulfill the contract; it handles exceptions. Like an emergency worker, it is only there to clean up the rubble and enable normal operation to resume if at all possible.

→ *"Starting it all", page 130*.

A routine that fails signals that it is unable to fulfill its contract. Unless it is the root procedure (the top level in the execution), this does not immediately cause the entire execution to fail; rather, the routine passes the problem up the chain in the form of an exception, which some higher-up may be able to handle through the **Retry** mechanism.

A consequence is that execution could come back, some time after a routine call *x*.*r* (…) has failed, to the object that was attached to *x*. A new call on that object can only work properly if it finds the object in a consistent state — a state satisfying the class invariant. As part of the exception rules, then, a **rescue** clause that leaves **Retry** false should restore the invariant. This is what "cleaning up the rubble" means in case E2, as expressed by the following principle:

Touch of Methodology: Failure Principle

A routine execution that fails should establish the class invariant, then cause an exception in its caller if any.

The principle helps define what will happen if an exception's recipient does not include a **rescue** clause. This is the case with the vast majority of routines; most programs include only a few **rescue** clauses located at strategic points high in the call chain, to enable recovery or graceful termination in the case of unexpected events. If an exception occurs in any routine with no **rescue**, the routine's execution will fail, passing on the exception to the caller. This is as if the routine had a **rescue** clause of the form

```
rescue
    default_rescue
```

where *default_rescue* is a library routine that by default does nothing. It is good practice to provide a new version of *default_rescue* in every class, with a simple implementation that establishes the class invariant in any available way.

default_rescue is inherited from the top-level class *ANY*, so that any class can *redefine* its implementation. These concepts will be part of the study of inheritance.

→ *Chapter 16.*

Exception details

In the *transmit* example, only one kind of exception can arise. Sometimes you may want to treat different kinds in different ways. All exception handling mechanisms allow you to access the details of the last exception, such as the exact exception type. In object-oriented languages such as Eiffel, Java, C# and C++, this information is accessible through an *exception object*, created automatically when an exception occurs.

In Eiffel, the object is available through the query *last_exception*; its type is a descendant of the library class *EXCEPTION*. To trigger a programmer-defined exception, use the library procedure *raise*, which creates an exception object.

→ *last_exception is also a feature inherited from class ANY; see "Overall inheritance structure", 16.10, page 586.*

The try-catch style of exception handling

Rather than the rescue-retry style based on Design by Contract principles, C++ uses a "try-catch" style which Java and C# also adopted with a few differences. The basic idea (you can see the details in the corresponding language appendices) is to write any exception-prone code in a **try** instruction, and process any resulting exceptions in one of its **catch** clauses:

→ *Java: "Exception handling", page 758; C#: "Exception handling", page 790; C++: "Exception handling", page 822.*

```
try
    … Process the most common case, possibly triggering an exception …
catch (e: T1, T2)
        … Process cases resulting from an exception of type T1 or T2 …
catch (e: T3, T4, T5)
        … Process cases resulting from an exception of type T3, T4 or T5 …
…
end
```

with specific processing for exceptions of every expected type. The **throw** instruction triggers a programmer-defined exception, creating an exception object. Unlike a **rescue**, a **catch** clause does not just "clear the rubble" but handles the exceptional case fully. The mechanism does not directly support the "retry" scheme, but you can program it by enclosing the **try** block in a loop (or, in C++, by using a **goto** if the preceding sections have not weaned you off the habit).

→ *This emulation is the topic of an exercise: "Emulating retry in a try-catch language", 7-E.8, page 210.*

Two views of exceptions

As language mechanisms, try-catch and rescue-retry are in the end equivalent: each will let you do, more or less conveniently, anything you can do with the other. Their spirits, however, reflect two different views of exceptions. At one extreme, exceptions are just another control structure, a generalized goto, handling any case that departs from the most common ones. At the other end, exceptions only address cases, particularly system errors, that no other technique can handle. There are also solutions in-between these extremes; as you start writing complex software and addressing its error-processing needs, you will find the style that suits you best.

→ *The exercise cited above explores emulation in one direction, for the reverse emulation, see the exercise "Emulating try-catch in a rescue-retry language", 7-E.9, page 210.*

7.11 APPENDIX: AN EXAMPLE OF GOTO REMOVAL

This last section is an opportunity to practice **goto** removal on a specific example, reasonably simple but not trivial. This is supplementary material, which you may skip on first reading, although a section (also supplementary) of the chapter on recursion relies on it.

→ *Chapter 14.*

It uses the concepts of variable and assignment which we have not formally studied yet; so if this book is your first encounter with programming you should come back here after reading the corresponding chapter.

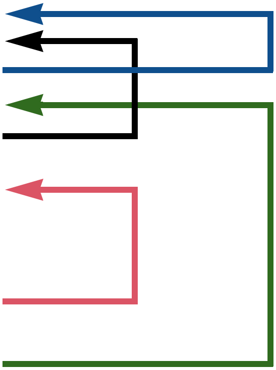
→ *Chapter 9.*

> If you do not want to wait just make sure you understand the following, which is all you need of chapter 9 material for the moment: a *variable* is a program entity that can take on different values during execution; attributes are variables, but you may also use *local variables* meaningful only during the execution of a routine. The way for a variable *x* to receive a new value, given by an expression *e*, is through the *assignment* instruction *x* := *e*.

Our target for **goto** removal is not an artificial example but a scheme that we will encounter in the discussion of the "Tower of Hanoi" program. It already includes a couple of higher-level **if** … **then** …**else** … **end** structures, which in a more elementary version would be represented in terms of **test** … **goto** … instructions. It does not matter for the present discussion what the basic operations — instructions *I0*, *I1*, *I2*, *I3* and conditions *C0*, *C1*, *C2* — are about, as long as we know that they do not themselves include any branching.

→ *"Simplifying the iterative version", page 494.*

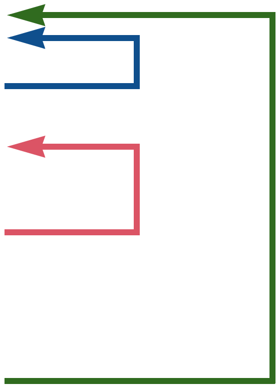
```
I0
start:   if C0 then
              I1
              goto start
after_1:      I2
              goto start
         end
after_2: if C1 then
              I3
              if C2 then
                   goto after_2
              else
                   goto after_1
              end
         end
```



The arrows on the right show the control structure; with its intertwined loops, it looks rather messy — another spaghetti bowl — and hard to modularize into a **goto**-less structure.

Note, however, that this intertwining is an artifact of the order of instructions (coming from the original recursive code given in the later discussion) and that we can easily get rid of it: since the whole after_1 block, highlighted, is only reachable through a **goto** (the instruction that precedes it in the code being itself a **goto** targeting another address), we may move it to any place we like outside of the other blocks, for example to the end:

```
I0
start:    if C0 then
                I1
                goto start
          end
after_2:if C1 then
                I3
                if C2 then
                      goto after_2
                else goto after_1 end
          end
after_1:        I2
                goto start
```



The spaghetti have been untangled: we see three loops with proper nesting. There remains the **goto** after_1, but since it branches to the immediately following instruction it is superfluous; we may just include the whole after_1 block in the **else** clause. So we can rewrite the entire structure without any labels or **goto**s, using instead two loops nested in another:

```
from I0 until over loop                          -- Formerly "start" position
      from until not C0 loop
            I1
      end
      from stop := not C1 until stop loop        -- Formerly "after_2" position
            I3
            stop := ((not C1) or (not C2))
      end
      over := ((not C1) and C2)
      if not over then I2 end                    -- Formerly "after_1 position"
end
```

Since two of the loops have non-elementary exit conditions (the second inner loop needs **not** *C1* before its first iteration, then **not** *C1* **and not** *C2*), this form of the program uses boolean variables *over* and *stop* to represent the conditions. This is a standard technique in **goto** elimination.

over is initialized to **False** *as usual through default initialization.*

7.12 FURTHER READING

George Polya: *How to Solve It*, 2nd edition; Princeton University Press, 1957.

> The acknowledged reference on becoming better at mathematical problem solving. Do not be put off by the publication date, this book is still a best-seller in its paperback edition.

Edsger W. Dijkstra: *Goto Statement Considered Harmful*, Letter to the Editor, in *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Available online at www.acm.org/classics/oct95/.

> A famous short paper that started the programming methodology revolution of the seventies and Structured Programming. Explains why the "Goto" is inappropriate for good programming but, even more importantly, illuminates the process of program construction, concisely (two pages) and effectively. Decades later, still a must-read.

← *"Goto harmful?", page 185*

Ole-Johan Dahl, Edsger W. Dijkstra, C.A.R Hoare: *Structured Programming*, Academic Press, 1972.

> A classic. Consists of three monographs, the first of which, Dijkstra's *Notes on Structured Programming* is the most famous; but the other two are just as interesting: Hoare's cogent description of the complementary need for *data* structuring, and Dahl's presentation (with Hoare) of the Simula 67 concepts now known as object-oriented programming. Few software books have had such an influence on the history of the field.

C.A.R. Hoare and D.C.S Allison: *Incomputability*, in ACM *Computing Surveys*, vol. 4, no. 3, September 1972, pages 169-178. Available online (with subscription) at portal.acm.org/citation.cfm?id=356606.

> Short, simple, clear and thought-provoking explanation of why certain functions — such as one to find out if a program terminates — can never be programmed on a computer.

← *"Loop termination and the halting problem", page 161*

7.13 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Control structures define the sequencing of actions in program execution.
- Control structures can be viewed as problem-solving techniques, reducing a possibly complex problem to a set of simpler problems.
- The main control structures are the compound, prescribing sequential execution of a specified list of actions; the conditional, prescribing execution of one among a specified list of actions, based on certain conditions; and the loop, prescribing repeated execution of a specified action.

- Correctness concerns are central to the proper use of control structures. A loop is characterized by an invariant, stating a condition maintained throughout, and a variant, an integer value that decreases on each iteration, ensuring termination.
- Lower-level structures such as the **goto** are important at the machine level but spurned in modern programming languages. Any program using them has an equivalent expressed in terms of the standard control structures.

New vocabulary

Algorithm
Branching instruction
Compound
Concurrent
Conditional
Conditional branching
Control structure
Cursor
Flowchart
Indirection
Iterate
Iteration of a loop
Jump table
Loop
Loop invariant
Loop variant
Overspecification
Parallel
Preserve
Sequence
Space-time tradeoff
Unconditional branching

7-E EXERCISES

7-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

7-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise “Concept map”, 6-E.2, page 138.*

7-E.3 Loops in machine language

Consider a loop of the form

```
from
    Compound_1
until
    i = n
loop
    Compound_2
end
```

Using the machine instructions **BR** and **BEQ** assumed in the discussion of branching, write the corresponding machine-language code.

← *"The lower level: branching instructions", 7.7, page 181.*

7-E.4 Flowchart for a conditional

Following the conventions of the flowchart for a loop, draw a flowchart for the conditional instruction **if** *Condition* **then** *Compound_1* **else** *Compound_2* **end**.

← *"Flowchart for a loop", page 184.*

7-E.5 Böhm-Jacopini in practice

Consider the following **goto**-based program extract relying on conditional **goto** instructions:

```
            Instruction_1
            test c1 goto t3
t2          Instruction_2
t3          Instruction_3
            test c2 goto t2
            Instruction_4
```

1 Draw the corresponding flowchart.

2 Propose a program extract that has exactly the same run-time effect but uses only compound, loop and conditional as control structures, without any **goto** instruction.

7-E.6 Forms of loop

Considering variants of the basic form of loop, show how to express:

← *"Other forms of loop", page 192.*

1 A **repeat** … **until** … loop as a **while** …loop.

2 A **while** … loop as a **repeat** … **until** loop.

3 The basic form (**from** … **until** … as in Eiffel) as a **while** … loop.

4 The basic form as a **repeat** … **until** … loop.

7-E.7 Emulating the variant

The loop variant provides the basis for demonstrating that a loop terminates. Assume that the loop syntax did not include a **variant** clause, but still had **invariant**. Using the example of the loop that computes the maximum of a set of values, and adapting the reasoning used for the variant, show how to demonstrate that a loop terminates. (**Hint**: introduce a variable to keep track of the previous value of the variant expression, and rely on the invariant.)

← *Page 163.*

7-E.8 Emulating retry in a try-catch language

Consider the rescue-retry scheme for handling exception cases, such as the *transmit* example, that may cause zero to *count* re-executions of the main algorithm. Show how to program it in a programming language offering an exception handling mechanism of the try-catch style.

← The code of transmit appears on page 202.

You may use the specific try-catch construct of any of the languages discussed in appendices: Java, C# or C++.

→ Java: "Exception handling", page 758; C#: "Exception handling", page 790; C++: "Exception handling", page 822.

7-E.9 Emulating try-catch in a rescue-retry language

Consider a try-catch style for exception handling, as sketched in this chapter. Show how to emulate it, or one of its specific variants (such as the Java form with a finally clause) using Eiffel's rescue-retry mechanism.

← "The try-catch style of exception handling", page 204.

To find out the type of the last exception, use *last_exception*.*type*. The notation $\{T\}$ denotes an object representing the type T, which can be an exception type.

8

Routines, functional abstraction and information hiding

The control structures of the previous chapter — compound, loop, conditional and their variants — give us basic mechanisms for scheduling instructions. If they were our only tools, we would always have to express the flow of control in full detail. For complex programs, the depth of nesting would soon defy understanding.

To keep that complexity under control, we resort to another time-honored problem-solving technique: **identify subproblems**. A subproblem is simply a problem whose solution may help solve other problems. If we are able to solve the subproblem and turn the solution into a control structure element, simple or complex, we can give that solution a name and use it through that name. This is known as *functional abstraction*; the corresponding programming mechanism is known as the *routine*.

8.1 Bottom-up and top-down reasoning

Why can it be useful to identify subproblems? Two complementary answers suggest themselves:

- In solving a problem, we may identify a subproblem to which we already know a solution. Then we will just plug that solution back into the solution of the larger problem. This is a **bottom-up** use of subproblems: work from what we already know to build solutions to bigger problems. Such a style of reasoning is, for example, fundamental in physics and engineering: an engineer will analyze an electrical system and model it in terms of some differential equation of a known type, then use known techniques to solve that equation and deduce properties of the system.
- In other cases we realize that part of a problem by itself constitutes a problem of its own — a subproblem — which we hope will be easier to solve than the overall problem. This insight may be useful even if we do not already have a solution to the subproblem, because it enables us to deal separately with various parts of the task. You may *assume* that there is a solution to the subproblem and use it to solve the larger problem; once you

have that larger solution, you will return to the subproblem and take care of its own solution. This is a **top-down** use of subproblems: work on the overall goal, and divide it into a set of smaller goals, to be solved separately. Top-down development is also known as "Divide and conquer" (or "Divide and rule"). We have already encountered a top-down technique: pseudocode, which lets us refer in an informal way to program parts that we intend to expand later through a *refinement* process.

← *Page 108.*

Whether in a bottom-up or top-down spirit, the use of subproblems is a form of **abstraction**: ignore the specifics of a particular situation to recognize it as an instance of a general scheme.

In programming, the corresponding construct, capturing the solution to a subproblem, is known as a **routine**.

Touch of Terminology: Routines by any other name

Routines have several other names. You may encounter the synonyms *subprogram* (suitably reminiscent of "subproblem"); also *subroutine*, out of fashion except for the Fortran programming language.

Routines may return a result, and are then called *functions*; a routine that does not return a result is called a *procedure*. Both of these terms are, however, sometimes used in reference to routines of the general kind; in particular, C and C-based languages use "function" for all routines.

As if this were not enough, you will also notice, for object-oriented languages, the word *method*, which means the same thing as "routine" but introduces confusions with the usual sense of "method", as in "*he writes his methods without any method*", or "*there is madness to her methods*".

Routines appear in both bottom-up and top-down development. In their bottom-up role, they support **reuse**: you can take advantage, for your program, of some algorithmic scheme that you or someone else has previously encountered and turned into a routine. In the top-down mode, you can use calls to routines that represent well-identified elements of the processing, and postpone the writing of the routines themselves. This is similar to using pseudocode, but more structured since you have to decide on a precise name and interface for the routine.

← *"Definition: Pseudocode", page 108.*

8.2 ROUTINES AS FEATURES

A routine captures an algorithm that is applicable to all instances of a class. As such it is one of the two kinds of **feature** of a class. The other kind, to be studied in the next chapter, is the *attribute*.

← *The definition of "feature" was on page 29.*

Like any other feature, a routine has:

- A **declaration**, which appears in the text of the feature's class, and describes all the properties of the routine; the declaration of a routine is also called its **implementation**.
- An **interface**, which retains only a subset of the properties of the routine, those interesting to clients that will use the feature; you can see routine interfaces in the *Contract View* of a class.

← *We studied Contract Views in "What characterizes a metro line", page 53.*

We have already encountered many routines, even though we knew them only as features. For example:

- Our very first feature, *explore* in class *PREVIEW*, was already a routine. So is the feature *traverse* that you have been asked to write, under successive variants, in the previous chapter.

← *"A class text", 2.1, page 15.*

- In studying how to use a class through its interface, we relied on a number of features from class *STATION*, some of which were routines, such as the command *remove_all_segments* and the query *i_th*. (Some others, such as *south_end* and *count*, are not routines but attributes.)

← *"Commands", 4.5, page 59 and subsequent sections.*

In the first case you had to write the entire routine declaration, but in the second case you only knew the routines through their interfaces, for example:

```
remove_all_segments
        -- Remove all stations except the South-West end.
    ensure
        only_one_left: count = 1
        both_ends_same: south_end = north_end
```

You can see the full routine text by looking up class *STATION*. You will now learn how to write your own routine declarations.

8.3 ENCAPSULATING A FUNCTIONAL ABSTRACTION

The last example of our study of conditionals provides a good case for defining a "functional abstraction" in the form of a routine. The overall loop, appearing in the routine *traverse* of our example class *ROUTES*, read:

← Originally on page 180, repeated here.

```
from ... invariant ... variant ... until ... loop                          [1]
    if Line8.item.is_exchange then
        show_blinking_spot (Line8.item.location)
    elseif Line8.item.is_railway_connection then
        show_big_red_spot (Line8.item.location)
    else
        show_spot (Line8.item.location)
    end
    Line8.forth
end
```

What's disturbing is not just the repetition, but the lack of recognition that the operations within the conditional all apply to the same object: the object denoted by *Line8.item*. This property will stand out much more clearly if we abstract the conditional structure into a routine. The loop then becomes:

```
from ... invariant ... variant ... until ... loop                          [2]
    show_station (Line8.item)
    Line8.forth
end
```

relying on a new routine *show_station* whose declaration will appear in the same class *ROUTES*:

```
show_station (s: STATION)
        -- Highlight s in a form adapted to its status.
    require
        station_exists: s /= Void
    do
        if s.is_exchange then
            show_blinking_spot (s.location)
        elseif s.is_railway_connection then
            show_big_blue_spot (s.location)
        else
            show_spot (s.location)
        end
    end
```

8.4 ANATOMY OF A ROUTINE DECLARATION

The declaration of *show_station* shows the typical form of a routine. Many of its elements are already familiar.

A routine is a software element denoting a certain set of operations to be performed on behalf of other software elements, said to **call** the routine. So far we only have one caller to *show_station*: our example loop [2], where the call reads

```
show_station (Line8.item)
```

Such a call usually appears in a routine; here we have assumed that the call is in the routine *traverse* of the same class *ROUTES*. Routine *traverse* is said to be a **caller** of routine *show_station*.

It could also appear in a contract clause.

If a routine of a class *C* is a caller of a routine of a class *S*, this makes *C* a *client* of *S*. Here the presence of the call makes *ROUTES* its own client.

← *"Client" was defined on page 47.*

In the overall system, a routine may be the target of zero, one or more calls, but it always has one declaration, which defines the routine's algorithm and appears in a class. Let us analyze the declaration of *show_station* as it appears on the previous page. The first line

```
show_station (s: STATION)
```

gives the name of the routine, as well as its **signature**: the list of its **formal arguments**, if any, and their types. Formal arguments represent values on which the routine will operate; each caller will pass these values through **actual arguments**, one for each formal argument.

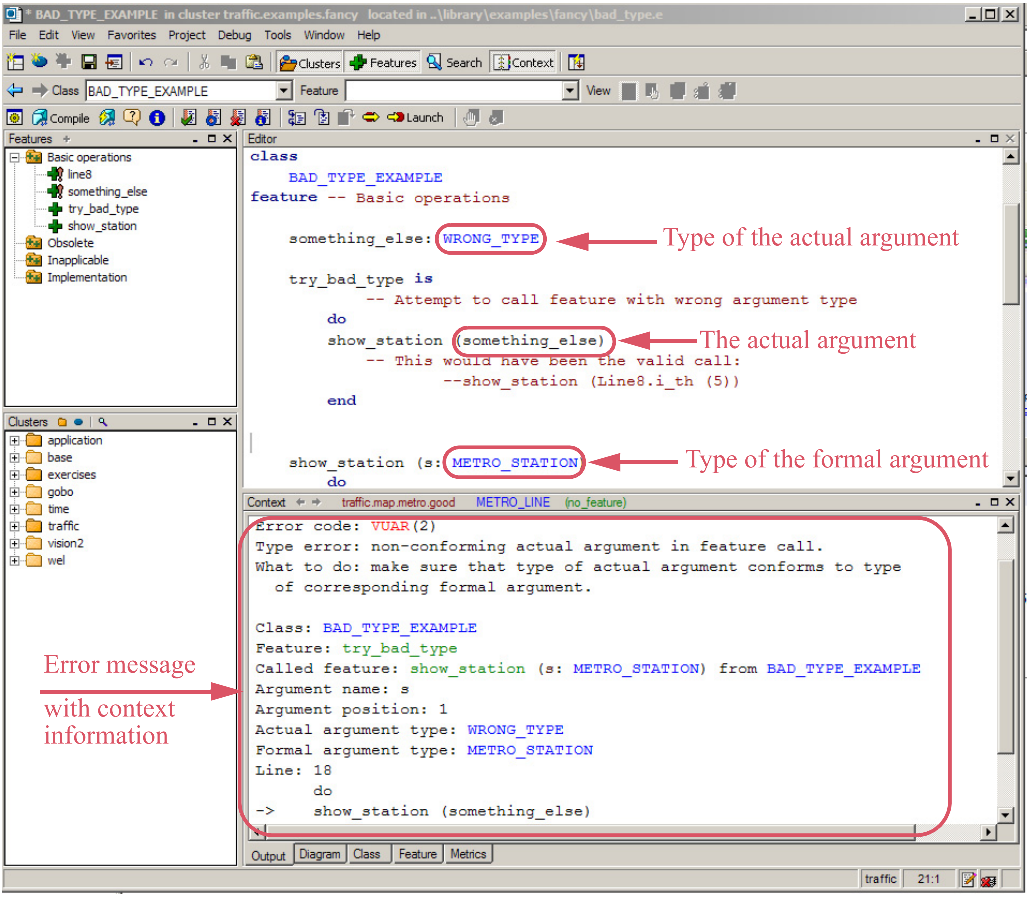
*Strictly speaking, this is the **argument signature**; we will see (page 220) that the signature of a function also includes a result type.*

An actual argument is an expression; its type must match the type of the corresponding formal argument.

The original definition of "argument" covered both formal and actual arguments.

← *"Features with arguments", 2.4, page 30.*

The signature of *show_station* involves one formal argument, *s*, of type *STATION*; in the example call in [2], the actual argument is *Line8.item*. The type of this expression is indeed *STATION*, since the query *item* of class *LINE* returns a station. If the types were incompatible, EiffelStudio would produce an error message when you attempt to compile the system:


* BAD_TYPE_EXAMPLE in cluster traffic.examples.fancy located in ..\library\examples\fancy\bad_type.e
File Edit View Favorites Project Debug Tools Window Help
Clusters Features Search Context
Class BAD_TYPE_EXAMPLE Feature View
Compile Launch
Features
Basic operations
line8
something_else
try_bad_type
show_station
Obsolete
Inapplicable
Implementation
Clusters
application
base
exercises
gobo
time
traffic
vision2
wel
Editor
Context traffic.map.metro.good METRO_LINE (no_feature)
Output Diagram Class Feature Metrics
traffic 21:1

```
class
    BAD_TYPE_EXAMPLE
feature -- Basic operations

    something_else: WRONG_TYPE

    try_bad_type is
            -- Attempt to call feature with wrong argument type
        do
            show_station (something_else)
                -- This would have been the valid call:
                    --show_station (Line8.i_th (5))
        end


    show_station (s: METRO_STATION)
        do
```

Type of the actual argument

The actual argument

Type of the formal argument

```
Error code: VUAR(2)
Type error: non-conforming actual argument in feature call.
What to do: make sure that type of actual argument conforms to type
  of corresponding formal argument.

Class: BAD_TYPE_EXAMPLE
Feature: try_bad_type
Called feature: show_station (s: METRO_STATION) from BAD_TYPE_EXAMPLE
Argument name: s
Argument position: 1
Actual argument type: WRONG_TYPE
Formal argument type: METRO_STATION
Line: 18
      do
->    show_station (something_else)
```

Error message with context information

In this example we have passed an actual argument of some arbitrary type *WRONG_TYPE* to a routine *show_station* that has a formal argument of type *METRO_STATION*. The error message explains what went wrong.

Within the routine *show_station*, we use the formal argument *s* as an expression denoting a station. The operations performed on *s* will, in any call, apply to the corresponding actual argument; in the example call, this is the station denoted by *Line8.item*.

Not all routines have arguments; *remove_all_segments* was an example without any.

The remainder of the declaration of *show_station* contains the following elements:

← *Page 214.*

- Like any feature, a routine should have a header comment explaining what it does. Here it is -- Highlight *s* in a form adapted to its status. Good practice for header comments requires mentioning all the formal arguments (here *s*) by name, so that client readers know the role of each, and not including redundant information (for example we say just "*s*", not "the station *s*", since the previous line declares *s*: *STATION*).
- A precondition, here *s* /= *Void*, states that we only want to work on actual arguments that are attached (not void).
- The **do** clause is called the **body** of the routine. It consists of a sequence of instructions — a Compound — defining the algorithm that the routine will execute.
- There could also be a postcondition, although none appears here.

Interface vs implementation

EiffelStudio lets you see both the implementation and the interface of a routine such as *show_station*:

- The implementation (declaration) appears in the default view for the class, known as the "Text View". It is the full declaration of the routine as we have seen it.

Or "Basic Text View".

- The interface appears if you request the "Contract View" by clicking the corresponding button. The result, of a form familiar to us from our earlier study of interfaces of features of class *LINE*, contains just:

← "What characterizes a metro line", page 53.

```
show_station (s: STATION)
        -- Highlight s in a form adapted to its status.
    require
        station_exists: s /= Void
```

The interface of a routine is intended for programmers of client classes; of the routine's elements listed above, it retains the signature, header comment, precondition and postcondition; but it discards the body, which describes how the routine is implemented. The interface of the routine should only describe *what* the routine does, not *how* it does it. The signature and contracts, complemented by the natural-language explanation that the header comment provides, suffice to express this "what".

The Contract View also differs from the Text View by omitting some syntactical details, such as the **end** keyword, that are necessary to avoid ambiguity in programs but not required in interface descriptions.

8.5 INFORMATION HIDING

The technique of presenting client programmers with an interface that includes only a subset of the properties of a software element — here a routine, but more generally a class or any other module — is called **information hiding**.

Information hiding is one of the key tools enabling you to build large software systems and cope with their complexity: provide the users of each element with *just what they need* to use it.

In spite of its name, information hiding is not about *preventing* client programmers from seeing the implementation of the mechanisms they use (classes, routines and other features): since Traffic and other Eiffel libraries are available in source form, you can use EiffelStudio to peek into the implementations of all the features of *LINE* and other Traffic classes. The actual purpose of information hiding is the converse: *not requiring* client programmers to look into the implementation of a software element, when all they need is to reuse it (as opposed, for example, to modifying or extending it).

If you had to read the full text of every routine you want to reuse, the amount of information to digest would quickly become enormous. Information hiding enables you to use software by reading only a small part of that information. It is our best ally in the programmer's constant effort to avoid getting swallowed by complexity.

> Not all libraries are available in source form; a library supplier may elect to provide interfaces only, usually to preserve proprietary know-how contained in the implementations. Whether to make the implementation available is a commercial or political decision; information hiding is a technical device, unrelated to that decision, for protecting programmers against having to learn heaps of irrelevant details.

Information hiding is a weapon not only against complexity but also against instability. One of the main characteristics of programs as developed in practice is the amount of *change* they undergo; there is a reason the field is called *soft*ware. Any change to a software element has the potential to affect every one of its clients, triggering a chain reaction of changes throughout the system. But if the elements have been well designed, with good choices of what goes into the interface and what remains an implementation decision, many changes will touch the implementation only. Clients will not be affected, since they only rely on the interface. This is an invaluable tool for keeping software projects under control.

Advanced object-oriented techniques of inheritance and dynamic binding will enable us to take information hiding one step further by letting clients ignore not only the internal details of operations they apply to objects, but also the exact *types* of these objects. We are not there yet; for now you should use EiffelStudio to discover what information hiding means concretely:

→ *"Beyond information hiding", 16.7, page 573.*

Programming time!
Experimenting with EiffelStudio and information hiding

When you hit the “Compile” button, EiffelStudio does not recompile the entire system, which could take too long. It only recompiles the classes that you have modified since the last compilation, plus any others that depend on them, directly or indirectly. This is known as *incremental compilation*.

EiffelStudio’s incremental compilation is *automatic*: you do not need to list the modified classes; EiffelStudio will detect them automatically, and will find out what other classes depend on them.

→ *See also “The melting ice technology”, page 357.*

In this dependency analysis, information hiding is essential: if you change only the implementation of a class, EiffelStudio will spot this, and will not recompile its clients. If your change affects the interface, EiffelStudio will recompile the clients. You may observe this now as follows:

1 Add a routine, say *r*, to *LINE*. It does not matter what the routine does, but give it an argument and a precondition.

2 In the routine *traverse* from *ROUTES*, add a call to *r*. Make sure the call is valid: it must use an argument of the right type.

3 Recompile the system. Notice what classes are being compiled. (To see the compilation messages, select the “Output” tab at the bottom.)

4 Change an element of the body of *r*, without touching the interface. Recompile, and observe what classes the compilation processes.

5 Now add a precondition clause to *r*; this changes its interface. Recompile, and notice how the compilation processes *ROUTES*.

6 To bring back the system to its previous state, remove *r* from *LINE* and the call to *r* from *traverse*. Recompile and execute to check that everything is back to what it was.

8.6 PROCEDURES VS FUNCTIONS

There are two kinds of routine:

- A **procedure** performs certain actions; a call to a procedure is, in the calling routine, an *instruction*. The preceding examples, such as *traverse* and *show_station*, are procedures. So are *creation procedures*, studied in an earlier chapter and serving to initialize class instances on creation.

→ *“Creation procedures”, 6.5, page 122.*

- A **function** computes a certain value (usually by performing actions too); a call to a function is, in the caller, an *expression*. We have not seen any function implementation yet, but several of the features that we used through their interfaces, such as *i_th* in class *LINE*, are functions.

The difference is closely related to one we already know:

- A procedure implements a *command* feature.
- A function implements a *query* feature.

← *"Features, commands and queries", page 26.*

> Commands can only be implemented by procedures, but for queries the next chapter will describe another possible implementation: through an *attribute*.

← *"Attributes", 9.2, page 238.*

We saw how the **signature** of a procedure is characterized by a list of formal arguments types, as in the beginning of the declaration of *show_station*:

```
show_station (s: STATION)
```

The signature of a function must, in addition, list the type of the value to be returned by the function. We saw this in the interface for functions such as *i_th* in *LINE*, which returns a result of type *STATION*, as expressed by the beginning of its declaration:

```
i_th (i: INTEGER): STATION
```

The rest of the declaration has the same elements as for a procedure: header comment, pre- and postcondition, **do** clause (body). In the body and the postcondition, we will need a name for the value to be returned by the function; it will be the reserved word **Result**, introduced in the next chapter.

8.7 FUNCTIONAL ABSTRACTION

Routines are the basic algorithmic blocks making up our classes and, through them, our systems.

Use routines as an **algorithmic abstraction** mechanism. To abstract means to concentrate on the essence rather than the circumstances, on the general concept rather than its instances. Abstracting almost always implies **naming**: once you have isolated a useful abstraction, you give it a name for ease of future reference. In programming we encounter two fundamental forms of abstraction:

- **Data abstraction**, which gives us the notion of *class* to describe the abstraction behind our program's data — objects.
- Algorithm abstraction, also called **functional abstraction**, to describe the abstractions behind our algorithms.

> In "functional abstraction" (the accepted term for this concept), the word "function" is taken in opposition to "data", rather than in its precise technical meaning defined above. "Routine abstraction" would be more accurate.

To keep your systems manageable even if their algorithms involve many details, you may rely on routines. Both the bottom-up and top-down views are attractive:

- When you have written an algorithmic element that covers a significant processing step, turning it into a routine enables you to give it a name and a precise specification (signature, header comment and contract); this makes it a well-defined software element and, among other benefits, facilitates the later **reuse** of the element. This is the bottom-up view.

- In the top-down view, you may use a routine to capture a step of the processing that you have identified while building a larger algorithm, but for which you have not yet written the details — and perhaps do not *want* to write the details yet as they would detract you from your main goal.

In this second role, routines are often a superior alternative to *pseudocode*. We saw the use of pseudocode, in a top-down development process, to capture elements of the algorithm that you don't yet want to develop. The example was a pseudocode comment

← *"Overall setup", 6.1, page 108.*

```
-- "Create line and fill in its stations"
```

which we could replace by a call to a **placeholder routine**, here a procedure *create_fancy_line*. For the system to compile, the routine must exist, even if it does nothing:

```
create_fancy_line
        -- Create fancy_line and fill in its stations
    do
        -- To be completed (your name, today's date)
    ensure
        line_exists: fancy_line /= Void
    end
```

Note the postcondition stating part of the contract: the routine must create an object for *fancy_line*.

Touch of Methodology: Placeholder routines

If you use a placeholder routine, always include information about *your name* and *today's date*, as well as a full header comment and any other explanation of what you intend to do, so that the purpose does not get lost if some time passes before the implementation gets written.

Also, ask yourself if the routine needs a contract (precondition and postcondition); if so, write it from the start, in the placeholder version. Such a contract is part of the routine's specification, helping you ensure that you understand what you need it for, and will provide precious guidance when the time comes to implement it.

8.8 USING ROUTINES

Routines — algorithmic abstraction — are one of your best tools in taming the beast of complexity. Use them generously to capture meaningful algorithmic elements. Use them bottom-up, to prepare existing elements for later reuse; use them top-down, to prepare for elements that you know you will need but do not yet want to write in full.

Programmers concerned with *efficiency*, in particular execution speed, are sometimes wary of using too many routines, since the machinery of calling a routine almost always means that a call to a routine takes longer than just executing the routine's instructions. A good programmer will, of course, pay attention to efficiency, as to all other qualities of software. But this is seldom a reason to limit the use of routines, for three reasons:

→ On general issues of software quality, see "Components of quality", 19.3, page 705.

- Modern computer architectures have drastically decreased the time penalty of routine calls.
- Except in the case of a routine call appearing in the body of an "inner loop" executed many times, any remaining penalty will remain proportionally small, at least for a program performing extensive computations. (If the program does not perform extensive computation, all this does not matter anyway.) There will typically be only a few such inner loops, making up a small part of the program, even if they account for a substantial proportion of its execution time. Then you should only worry about these elements, once you have identified them. The rest of the system, where such small-scale performance considerations have little effect, should be left alone.
- For those program elements where it does matter to avoid the price of a routine call, you do not always have to do the job yourself. The EiffelStudio compiler will, in its optimized ("finalization") mode, perform automatic *routine inlining*. This means that it will expand your routine calls to be executed as if the instructions had been written directly — "in line" — at the place of each call. The advantage of this approach is that you do not need to damage the structure of your program and risk introducing bugs. The inlining process is automatic, although you may change some parameters such as the largest size of routines to be inlined.

→ "The melting ice technology", page 357.

Studying *algorithm complexity* will give us a better framework to talk about efficiency, emphasizing overall performance as a function of the size of the data set.

← "Estimating algorithm complexity", 13.3, page 376.

8.9 AN APPLICATION: PROVING THE UNDECIDABILITY OF THE HALTING PROBLEM

An earlier comment stated that it is impossible to devise an algorithm (“effective procedure”) to determine whether an arbitrary program will halt. Let us prove this result, under the observation that if such a general algorithm existed we could write an Eiffel routine that implements it.

← *“Touch of Theory: The Halting Problem and undecidability”, page 164*.

Specifically, we would be able to write a function

```
terminates (root_directory: STRING): BOOLEAN
        -- Does execution of the system available in root_directory,
        -- if any, terminate?
    do
        … An appropriate algorithm …
    end
```

The argument, *root_directory*, is the name of a directory assumed to contain the system’s “ECF”, that is to say the description of the system’s setup, giving access to all its classes and specifying the root class and the root creation procedure. We assume for simplicity that the ECF will be a file called *system*.*ecf* in that directory. Being able to solve the Halting Problem then implies that we can complete the “appropriate algorithm” so that *terminates* (*r*) will return **True** if and only if there is indeed such a file in *r* and execution of the corresponding system will terminate.

An ECF (Eiffel Control File) is generated automatically from the settings you select in EiffelStudio. The format is XML.

You may change the conventions or adapt them to another programming language; instead of an ECF, the argument could simply be the name of a file containing the texts of all classes in the system, plus the names of the root class and root procedure. What matters is that it is possible, through arguments to the function *terminates*, to pass information allowing the function to obtain the text of the system. The function’s job is then to *decide* (as in “Decision Problem”, Entscheidungsproblem in German) whether the system terminates.

All this assumes that the system needs no run-time input. A more general form of the function would handle possible input:

```
terminates_on_input (root_directory: STRING; input: STRING ): BOOLEAN
        -- Does execution of the system available in root_directory,
        -- if any, terminate when applied to input?
    do
        … An appropriate algorithm …
    end
```

We stay with the first form, but the reasoning applies just as well to the second one.

That reasoning is simple. If we had an implementation of *terminates*, we could use it to write a one-class system with the following root procedure:

```
paradox
        -- Terminate if and only if not.
    do
        from
        until
            not terminates ("C:\your_project")
        loop
        end
    end
```

Note empty loop body.

C:\your_project is just an arbitrary directory name; it is a Windows-style directory (folder) name, so on another operating system you would use something else, for example */usr/home/your_project* on Unix. What matters is that we use the name of the actual directory where we will store the ECF file for our "paradox" system **itself**. Then the call to *terminates* decides whether that system terminates. Now consider what the creation procedure does:

- If the function *terminates* determines by analysis of the system's program text that its execution will **not terminate**, the loop's exit condition **not** *terminates* ("C:\your_project") will already hold the first time around, and the loop will **terminate** immediately; so will the entire system since it does nothing else. This is a contradiction.
- If the function determines that execution will **terminate**, the exit condition will never be true, so the (empty) loop body will execute forever, and the system will **not terminate** — contradiction again.

This shows that it is impossible to write a general-purpose *terminates* function that would ascertain termination for an arbitrary program.

> We will see a more concise version of the argument, using *recursion*, in a later chapter; an exercise will ask you to devise an even shorter one, ignoring files and directories, using *agents*.

→ "From loops to recursion", 14.6, page 471; and "The Halting Problem with agents", 17-E.8, page 661.

8.10 FURTHER READING

David Parnas: *A Technique for Software Specification with Examples*, in *Communications of the ACM*, vol. 15, no. 5, 1972, pages 330-336, and *On the Criteria to be Used in Decomposing Systems into Modules*, ibid, vol. 15, no. 12, 1972, pages 1053-1058.

> Two classic papers, which introduced the notion of information hiding. Still excellent reading.

Parnas (2007)

8.11 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Routines provide functional abstraction: the ability to give a name to a possibly parameterized algorithm.
- Routines may be used top-down, as placeholders for algorithms to be refined later in the design process, or bottom-up, to capture useful algorithms for reuse in several projects.
- In an object-oriented context, routines are one of the kinds of feature. They themselves have two categories: functions, which return a result, and procedures, which do not.
- Routines may have arguments, enabling callers to pass specific information to every call.
- A routine has a name, a signature defining the types of arguments and result if any, a contract, and a body describing its algorithm.
- The name, signature and contract define the interface of the routine, as available to authors of client modules.
- Information hiding is the mechanism separating interface information from implementation details, and enabling clients to use routines and other software elements on the basis of the interface only.
- Information hiding facilitates the writing of large systems, the reuse of software elements, and the smooth evolution of software.

New vocabulary

Actual argument
Body
Data abstraction
Declaration
Formal argument
Function
Functional abstraction
Implementation
Incremental compilation
Information hiding
Placeholder routine
Procedure
Routine
Signature

8-E EXERCISES

8-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

8-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise “Concept map”, 7-E.2, page 208.

9

Variables, assignment and references

Programs use names or “entities”, to denote run-time values. A distinctive property of most programs is that some of their entities, called “variable entities” or just **variables**, can denote different successive values during execution. The previous examples have implicitly relied on such changes of values, but we have not yet seen the basic change operation, assignment.

It is a fascinating concept, deceptively simple when you first see it, and full of surprising consequences. We will study it in this chapter together with a number of related techniques, in particular *references*, which define the run-time object structure.

Math is static, software is dynamic

The ability of a program to change its own environment is the most significant difference between software construction and mathematical reasoning, two activities that are similar in so many other respects.

Mathematics uses transformations, but they are mechanisms to describe certain values in terms of others, not to change any value that existed before. If I write “*Let* $y = cos\ (x)$”, I am not changing or even creating anything, just giving a name to a value, the cosine of x, that existed all along, whether or not anyone had bothered to talk about it. In particular I am not changing x.

Even if after talking about this y I want to contrast the properties of the sine and cosine functions, and continue “*Let’s now assume instead that* y *is* $sin\ (x)$, *then*…”, I am reusing the name y for convenience but talking about another mathematical object. If in describing a sequence I say “Let f_1 and f_2 be 1, and $f_{i+2} = f_i + f_{i+1}$ for every $i > 0$” I am speaking of an infinite sequence of values, not a value that changes as i increases.

→ *The Fibonacci sequence: “Recursively defined algorithms and routines”, page 438.*

The software perspective is different. We do not just describe results by the properties they must satisfy: we must *compute* them through algorithms whose implementation uses a computer and its memory.

The execution of these algorithms proceeds by storing successively computed values into memory. If memories were infinitely large and infinitely cheap, the execution might choose a different cell for every new value to be stored, such as successive f_i. But memory, however large, is finite: we must reuse cells when we do not need their values any more.

So in programming we will have *variables* which, unlike their counterparts in mathematics, deserve their names, as they change value during execution. The presence of such change is one of the major challenges in efforts to reason about programs using the basic tools at our disposal: the tools of logic and, more generally, of mathematics.

Not everyone accepts this distinction. *Functional programming*, and the supporting "functional languages", bring software construction closer to mathematical reasoning by eliminating or strictly limiting the amount of change, or *side effects*, that programs can explicitly perform. The basic construct is the function, in the mathematical sense, without side effects. Most programming languages are *imperative* (they allow side effects). A later chapter explores this issue further and presents some of the essential concepts of functional languages.

→ *"Functional programming and functional languages", page 324.*

Eiffel belongs to the class of imperative languages, although the Command-Query Separation Principle confines side effects to procedures, facilitating mathematical-like reasoning about programs.

9.1 ASSIGNMENT

Assignment is the instruction that allows us to change the value of a variable.

For the examples and exercises of this chapter, you will use a new class called *ASSIGNMENTS*.

Summing travel times

The following simple problem will serve as example: knowing the average time between adjacent stops on a Metro line, compute the average total time for traveling the full line. We will add to class *LINE* a function *total_time* taking care of this.

The principle of the algorithm is straightforward: follow the stops on the line in sequence, and add at each step the time from the previous stop. The basic information comes from a query of class *STOP*:

```
time_to_next: REAL
        -- Estimated travel time to next stop (departure to departure,
        -- except for next-to-last stop: departure to arrival).
    require
        has_next: is_linked
```

where *is_linked* tells whether the metro stop is linked to a successor. The type *REAL* is used for the computer approximation of the real numbers of mathematics.

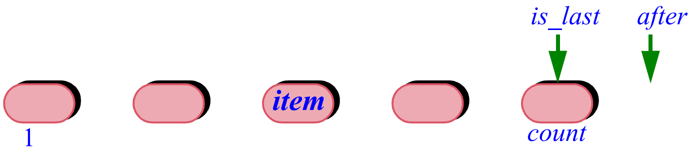
Our desired function *total_time* will have the following general form:

```
total_time: REAL
        -- Estimated travel time for full line.
    do
        from
            start
            -- "Set Result to zero"
        invariant
            -- "The value of Result is the time to travel from first station
            -- to station at cursor position"
        until
            is_last
        loop
            -- "Increase Result by the time to the next station"
            forth
        variant
            count – index
        end
    end
```

← *Elements in red are pseudocode; see "Definition: Pseudocode", page 108.*

Result, a variable, denotes the result to be returned by the function. The two pseudocode instructions will be replaced by assignments.

The boolean-valued function *is_last* tells us whether the cursor is on the last element. Note the difference with the loop schemes of the previous chapter, which stopped on *after* rather than *is_last*.



Basic list queries

Quiz time: When to exit from the loop

Why does the loop for *total_time* use *is_last*, rather than the usual *after*, as exit condition?

(*Hint*: compare the number of stops with the number of intervals between successive stops. Also, compare the variant with the earlier one.)

The two instructions still in pseudocode must update the value of **Result**. This is what assignments are good for.

An assignment instruction has the form

```
target := source
```

where *source* is an expression, and *target* is a variable such as **Result**. The run-time effect is to change the value of *target* to that of the *source*. To be precise:

Touch of Semantics: The effect of an assignment

The execution of an assignment instruction *target* := *source* consists of:

A1 Evaluating (that is to say, computing the value of) the expression *source*.

A2 Causing the variable *target* to denote that value from now on (and to retain it until the execution of any later assignment to *target*.)

This is the only effect of the instruction. In particular, there is no consequence (aside from the evaluation) on *source* and its components.

If you programmed before reading this book and view assignment as an old friend, you may find this definition pedantic. But precision matters; and it seems that some people intuitively understand, the first time around, that in $x := y$, where y is a variable, the value of y somehow “flows” to x, and y reverts to a default value. Nothing of the sort occurs; y is unaffected. (Don’t laugh. Programming is a strange world for a novice; it is this book’s job to dispel any potential confusions, and indeed bring you to the stage where you could laugh about them.)

An expression such as *source* usually involves variables; “evaluating” it (A1) will use their values, as set by previously executed assignments.

We can make good use of assignment to complete our example function:

```
total_time: REAL
        -- Estimated travel time for full line.
    do
        from
            start
            Result := 0.0
        invariant
            -- “The value of Result is the time to travel from first station
            -- to station at cursor position”
        until
            is_last
        loop
            Result := Result + item.time_to_next
            forth
        variant
            count – index
        end
    end
```

Each time through the loop, we add to the current **Result** the time to the next station. Since we also perform a *forth*, this preserves the invariant. On exit from the loop, that invariant tells us that **Result** denotes the time to travel to the station at cursor position; since *is_last* is now true, the cursor is on the last station, so **Result** gives us the total traveling time.

Programming time!
Estimating the time to travel a metro line

Write a function *total_time8* to compute and display the travel time on the Metro Line 8. Use the above model, but avoid modifying *LINE*: make the function part of *ASSIGNMENTS*, the class for this chapter, adapting it to use *Line8.start* instead of *start*, *Line8.count* instead of *count* and so on.

Local variables

In a previous chapter we saw an algorithm scheme for computing the maximum of a set of values. In the absence of assignment, it resorted to pseudocode elements of the form

← *See e.g. page 163.*

```
    -- "Define max to be $N_1$"
    -- "Define i to be 1"
    -- "Redefine max as the greater of the current maximum and $N_{i+1}$"
    -- "Increase i by one"
```

We may now express the algorithm fully using assignment. Let us write it as a function that computes the greatest name, alphabetically, of all the station names on a line:

```
highest_name (line: LINE): STRING
        -- Alphabetically last of names of stations on line.
    require
        line_exists: line /= Void
    local
        i: INTEGER
        new: STRING
    do
        from
            Result := line.south_end.name
            i := 1
        invariant … --- As before
        until
            i = line.count
        loop
            new := i_th (i).name
            if new > Result then
                Result := new
            end
        i := i + 1
end
```

We have indulged in a little orgy of assignments. The **from** clause initializes **Result** to the name of the first station, *south_end*, and the integer *i* to one. Then in the loop we find out if the name of the current station, denoted by *new*, is greater than the name of the current maximum, and if so we replace the value of **Result** by the value of *new* (if not, we leave **Result** unchanged, as there is no **else** clause to the **if**). The correctness of this algorithm depends on two properties expressed by the invariants of the corresponding classes:

- A *LINE* always has at least one station, accessible as *south_end* or, equivalently, *i_th* (1).
- Every metro station has a non-void *name*.

Also note that order comparison for strings uses alphabetical order: *s2 > s1* has value *True* if and only if *s2* is after *s1* alphabetically.

Programming time!
Alphabetically highest station name

Add the function *highest_name* to the example class for this chapter, *ASSIGNMENTS*, and use it to display the alphabetically highest name of stations on Line 8 of the Metro.

The principal novelty of this example is its use of *local variables*. The declarations

```
local
    i: INTEGER
    new: STRING
```

introduce two entities, *i* and *new*, which the routine may use to store intermediate results that its algorithm may need. "Local variables" are such entities, local to a routine and introduced by the keyword **local**. You could do without local variables, declaring *i* and *new* (in this example) as features of the class, more precisely *attributes* as studied next. But this would be giving them a status they do not claim: a feature is a property of the class, applicable to every one of its instances; here we only need *i* and *new* temporarily for each execution of the routine. When such an execution terminates, *i* and *new* can go away.

You can choose names of local variables freely as long as they do not cause any ambiguity:

Local Variable Rule

A local variable may not have the same name as a feature of the enclosing class or as an argument of the enclosing routine.

In principle it would be possible to allow the reuse of feature names as names of local variables, with the convention that within the function the name denotes the local variable; but this would be foolish language design, inviting confusion and errors. Names are cheap; when you need a new variable, choose a new name.

Nothing prevents you from using the same names for local variables of *different* routines, in the same way that different classes may use the same feature names (some names such as *item*, *count*, *put* … occur in many different classes). Such cases do not cause any risk of ambiguity since the homonyms appear in different contexts (or "scopes").

Function results

Result, as used in the last two examples, may appear in a function, where it denotes the result being computed by the function. Remember that *functions* are one of the two kinds of *routine*; procedures, the other kind, can change objects but do not return a result. A function returns a value. **Result** serves to denote, within the function's text, that future result as computed so far. (Obviously, you may not use **Result** in a procedure.)

← *"Procedures vs functions", 8.6, page 219.*

As a consequence, the instruction (in a routine of class *ASSIGNMENTS*)

```
Console.show (highest_name (Line8))
```

will call *highest_name* and display its value, which is the last value of **Result** as computed by the function's body just before its execution terminates. You will have seen this if you took the last "*Programming time*".

Result is, formally, a local variable. Its only distinction is that you do not declare it as you do with your own local variables (in declarations of the form *i*: *INTEGER*); it is automatically available in any function, and implicitly declared for you with the return type you specified for the function: *REAL* for *total_time*, *STRING* for *highest_name*.

This also means that **Result** is a **reserved word** of the language: you may not use it for any of your own identifiers.

> **Definition: reserved word**
>
> A **reserved word** is an identifier that has a special role in the programming language, and as a consequence may not be used to denote elements (such as class names, feature names, variables) specific to a particular program.

Reserved words generalize the notion of *keyword* introduced earlier. The example of **Result** illustrates why keywords are only one of two kinds of reserved words:

← *Page 17.*

- Keywords — such as **class**, **do**… — play a syntactic role only, as markers; they do not denote any run-time value.
- Other reserved words, such as **Result**, directly carry a semantic value. **True** and **False**, which denote boolean values, are also examples of non-keyword reserved words.

← *"Boolean values, variables, operators and expressions", page 72.*

Swapping two values

Here is a typical use of assignment and local variables. Assume two variables *var1* and *var2* of the same type *T*. The following three instructions will swap their values:

```
swap := var1 ; var1 := var2 ; var2 := swap
```

This requires a third variable, *swap*, typically declared in the enclosing routine as a local variable, also of type *T*. The scheme is:

- The first assignment stores the initial value of *var1* into *swap*.
- The second assignment changes the value of *var1* to the initial value of *var2*.
- The third assignment changes the value of *var2* to the value of *swap*, which had recorded the initial value of *var1*.

This completes the desired swap: after execution of these three instructions, *var1* has the value that was initially *var2*'s value, and conversely.

It is clear why we need *swap*: we must have a place to store away the value of one of the other two variables before overriding it. Also note the importance of the *execution order* in this example: the first assignment must execute before the second, since we would not otherwise be able to record the initial value of *var1*. There is, however, more than one correct execution order: reversing the order of the last two instructions is harmless; and we could reverse the roles of *var1* and *var2*.

A variable such as *swap*, used only for a narrow, immediate purpose, is known as a **temporary variable**. A temporary variable is typically declared as a local variable of the enclosing routine.

← Exercise 9-E.5, page 270 asks you to achieve the same result without local variables.

The power of assignment

The symbol for assignment is **:=**. You may read it aloud as "**receives**", for example "*i* receives *i* plus one" for $i := i + 1$.

> Some people say "*becomes*", but "receives" is better: *i* is just *i* and does not "become" $i + 1$, unless this is meant as in the story of the native German speaker getting tired of waiting in a London restaurant: "*Waiter*! *I want to become a potato NOW*!".

The effect is to replace the value of the *target* by that of the *source* expression. The earlier value of the target is lost — and lost forever: no one is keeping any record. Assignment is the counterpart, in high-level programming languages, of a basic operation permitted by computers: replace the content of a given memory cell by a given value. So if you will need a value again, make sure to record it yourself — through another assignment! — into a variable.

A common pattern in assignments is to use the previous value of a variable in the source of an assignment that has the same variable as its target. This appeared in instructions of both of the routines we have seen:

```
    Result := Result + item.time_to_next
    i := i + 1
```

The goal is to update the value of a variable on the basis of its previous value and new information. The scheme is very close to the standard mathematical technique of defining a sequence of values, as in a slightly simplified version of the example cited at the beginning of this chapter:

"Let s_0 be given, and then $s_{i+1} = f(s_i)$ for every $i \geq 0$"

where f is some function (in the Fibonacci sequence example f was a function of two values rather than one). To compute s_n for some $n >= 0$ with a computer you may use the loop

```
from
    Result := "The given initial value $s_0$"
    i := 0
invariant
    "Result = $s_i$"
until
    i = n
variant
    n – i
loop
    i := i + 1
    Result := f (Result)
end
```

This scheme is only applicable if you do not need to retain the successive values, only the last one s_i at each step. Both of our routines used it.

Be sure to remember the difference between the mathematical property $s_{i+1} = f(s_i)$ and the software instruction $x := f(x)$, which reflects the *change* mechanism of software. This mechanism is foreign to mathematics and complicates reasoning about programs. Note in particular the difference between the instruction

```
x := y
```

and the boolean expression

```
x = y
```

as used for example in a Conditional **if** $x = y$ **then** … The boolean expression has the same characteristics as an equality in mathematics; it is **descriptive**, presenting a possible property (true or false) of two values x and y. The Assignment instruction is **prescriptive** (or **imperative**): it tells the computation to change the value of a variable. In addition it is **destructive**, obliterating the previous value of that variable.

A striking example of the difference is the instruction

```
i := i + 1
```

frequently encountered in loops, using an integer variable i. The boolean expression $i = i + 1$, while legal, would be useless since it always has value false: no integer may be equal to the integer that follows it.

Touch of Syntax: Confusing assignment and equality

The first widely used programming language, Fortran (from the 1950s), used the equality symbol **=** for assignment. This was clearly an oversight; subsequent languages such as Algol and its successors introduced **:=** for assignment, reverting **=** to its standard role of equality operator.

→ *Appendix D.*

For unknown reasons, the C language, in the late sixties, brought back **=** for assignment, using **==** for equality. Not only does this convention contradict well-established mathematical properties (for example, $a = b$ in mathematics means the same as $b = a$), but it introduces a frequent source of errors; if instead of **if** $(x == y)$ … you mistakenly write **if** $(x = y)$ …, the result is actually legal in C, but has an unexpected effect: assign the value of y to x; then yield a boolean value (as if the assignment were also a boolean expression), which is **False** if the resulting value of x — the previous value of y — is zero or equivalent, and **True** otherwise! If you use C you must be careful about this source of confusion, which plagues even experienced C developers, and has caused bugs and security holes in important programs.

→ *Appendices A to C.*

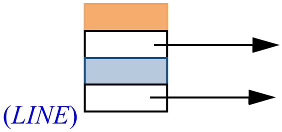
Such recent languages as C++, Java and C# have retained the C convention for assignment and equality, with (in the last two cases) stricter type rules to avoid such bugs.

9.2 ATTRIBUTES

There are two kinds of variables (entities to which we may assign a value). We have now seen the first kind: local variables, including **Result**. The second, to be studied now, is *attributes*. It is not completely new: we have seen it implicitly, under the guise of object *fields*, when learning about object creation. But we can now complete our understanding of this concept, and find its place among entities, features and other creatures of our object-oriented bestiary.

Fields, features, queries, functions, attributes

We saw in the discussion of creation that an object, as it exists at run time in the memory of your computer, consists of a number of *fields*, some of them references, others of basic ("expanded") types:



An object and its fields

Like any other property of the object, these fields must come from the specification of its generating class. Each field indeed comes from a feature of the class, more precisely a query, and even more precisely an attribute.

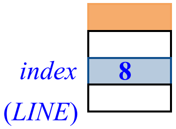
To restart from the beginning, a feature is, as you know, either a command or a query. A query, unlike a command, returns a result. A query can in turn be either a function or an attribute. It is a function if it obtains its result by computing it. For example, class *LINE* had this query:

```
south_end: STATION
        -- End station on South side
    do
        if not is_empty then
            Result := metro_stops.first.station
        end
    end
```

This is a function. On the other hand, we find, in the same class, the following query *without* an algorithm (a **do** … **end** part):

```
index: INTEGER
        -- Index of currently considered station in line
```

This is an attribute. Including it in the class means stipulating that every instance of the class will have a field of the given type — *INTEGER* — containing the current value of the *index* for the station:



An object and its fields

Assigning to an attribute

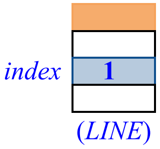
As the comment indicates, *index* in class *LINE* is the index of the "cursor" position; the cursor is an abstract mechanism allowing clients to explore successive stations of a line by going back and forth. One of the commands for manipulating the cursor is *start*, which sets the cursor to the first station (the one known as *south_end*):

```
start
            -- Bring station cursor to first element.
      do
            index := 1
            … Other instructions …
      ensure
            on_first: index = 1
      end
```

A client may call this feature on a particular line, as in

```
Line8.start
```

The effect is to set the value of *index* for the corresponding instance of *LINE*. If that object previously had its *index* field set to 8, as in the preceding figure, the call will reset it to 1, with no change to other fields:



***"Line" object after a** start*

Line8.start is a qualified call to *start*, from a client. As usual, it is also possible to call *start* unqualified from another routine of *LINE*.

Information hiding: modifying fields

Two other procedures of the class also set *index*:

```
forth
        -- Move station cursor to next item.
    require
        not_after: not after
    do
        index := index + 1
    ensure
        moved_right: index = old index + 1
    end

go_ith (i: INTEGER)
        -- Move station cursor to item at position i.
    require
        not_over_left: i >= 0
        not_over_right: i <= count + 1
    do
        index := i
    ensure
        set: index = i
    end
```

All three procedures let clients set the *index* field of any particular *LINE* object, as in

```
Line8.start

Line8.forth

Line8.go_ith (5)                                        [3]
```

Just as importantly, such procedure calls are the **only** way for a client to modify this field. You will not be permitted — try it if you wish, and see the compiler message — to write an assignment

```
Line8.index := 98                                       [4]
```

Warning: *syntactically illegal. For discussion only.*

The syntax may be legal as an "assigner call". See below.

As an assignment this is simply illegal syntax: the target of an assignment must be a variable, and a variable consists of a single identifier, such as *index* in the previous assignment examples. *Line8*.*index* is an expression, not a variable.

The reason for this prohibition is easy to understand. Letting clients directly modify fields of supplier objects would bypass the safeguards of information hiding and good design. Remember the general view, illustrated by an earlier picture reproduced below, of an object as a machine that clients may only manipulate through the operations of its official interface, illustrated as command and query buttons. Performing a direct assignment *your_machine*.*your_field* := *my_value* would be the software equivalent of unscrewing the casing to reveal the innards of the machine, and starting to rewire the connections with a soldering iron. With an electronic device this would void the warranty; with a software machine, it would void the interface and the associated contracts.

← *"Information hiding", page 218; see figure page 28.*

"Line" object as machine

start
forth
go_ith
item
index
count
south_end

Note in particular a key difference between the illegal assignment [4] and the procedure call [3]. The call is bound by the precondition of *go_ith*, stating

```
require
    not_over_left: i >= 0
    not_over_right: i <= count + 1
```

The assignment, if permitted, would ignore that precondition.

Any operation that may access or modify object fields must go through the interface provided by the features of the corresponding classes. When you design a class, it is both your privilege and your responsibility to decide what you let clients do with its instances. For any of the attributes, say *a* of type *T*, you may allow clients to set the corresponding field values directly; then you should provide a procedure — called a *setter* — of the form

```
set_a (x: T)
        -- Set the value of a to x.
    do
        a := x
    ensure
        set: a = x
    end
```

through which clients may use *their_object*.*set_a* (*their_value*) without restriction. Or you may introduce a precondition, as in *go_ith*, which restricts the permitted values. Or you might limit clients to more specific ways of setting the value, as would be the case if *LINE* did not have *go_ith* but provided only *start* and *forth* as operations that affect the *index* field. Finally, you might decide not to give clients any way at all to modify *index*, by not providing them any procedure that assigns to *index*.

> In the first case — where the class design has granted clients full modification privileges, with a procedure such as *set_a*, or *go_ith* in the example — some people find the assignment syntax [4] more attractive than the procedure call [3]. Hence a natural idea: could we not offer [4], not as an assignment but simply as a notational convenience, a shorthand for the procedure call [3]?

> This is indeed possible if you declare the setter procedure, *set_a*, or *go_ith*, as an **assigner command** for the associated query, *a* or *index*. Simply change the declaration of the query to *a*: *T* **assign** *set_a* or *index*: *INTEGER* **assign** *go_ith*. Then *obj*.*a* := *v* is valid but is not an assignment; it is simply a different syntax for the (also valid) call *obj*.*set_a* (*v*). More details in a later discussion.

→ *"Bracket notation and assigner commands", page 384.*

Regardless of the syntax, the static semantic rule is the same: the only way to modify an object from the outside is through a setter procedure. To realize how fundamental this is to the proper engineering of software, consider the following two events in the evolution of a system:

- You decide at some point that — even though this was not included in the original concept for the software — every modification of a certain attribute should be logged, for example by writing a record into a database ("At 7:55 on May 1st, the temperature was changed to 22° C").

- Adding a constraint to the attribute, for example that any setting of the temperature must use a value between -5° C and +30° C. This can take the form of an extra clause in the class invariant.

To achieve the first change, it suffices to add an instruction (updating the database) to the setter procedure. The second change is more delicate since it implies adding a precondition clause to the setter and may affect every client that sets the field; but since every such client goes through the setter it is easy to trace those clients and update them to satisfy the new precondition. If, on the other hand, direct [4]-like field assignments were permitted, you would have a hard time tracing all field-modifying clients; worse, it would be impossible to enforce the new policy — update the database after setting the field, or check that the value is within the new bounds — on *future* clients.

The "callers" view of EiffelStudio gives the list of all client instructions that call a given feature, either explicitly or through the assigner command

We may summarize the discussion through a simple principle:

Touch of Methodology: Attribute Modification Principle

The sole way of setting a foreign object's fields should be through calls (of any syntactic form) to exported setter procedures.

This key rule of modern design and programming methodology is an immediate consequence of information hiding principles. When you program in Eiffel it is automatically enforced. It needs emphasis, however, because of the different attribute export policies of other programming languages, which turn it into a methodology rule for programmers.

→ As discussed below: see "Setters and getters", page 248.

Information hiding: accessing fields

The preceding discussion addresses how to *modify object* fields, corresponding to class attributes. The rules do not prevent you from letting clients *access* object fields such as *index*. With the class as given, a client may use the expression *Line8.index*; try for example

```
Line8.start
Line8.forth
Console.show (Line8.index)
```

which will display the value 2 in the Console window.

In other cases you may wish — for full information hiding — to remove an attribute completely from the clients' reach, for access as well as modification. For example *LINE* has a feature *id_generator*; it uses that feature for its own implementation purposes and does not make it available to clients in any form. It suffices for the class to include a **feature** clause starting with **feature** {*NONE*}; all the features of that clause are kept away from clients. You can indeed see at the end of *LINE*, just before the **invariant**, the clause

```
feature {NONE} -- Initialization
    id_generator: ID_GENERATOR
            -- Internal identification for current line
```

This implies that an expression such as *Line8*.*id_generator* is invalid in any client (try to use it in a class, and see the compiler message). Accordingly, it will not feature in the class documentation as produced by the environment: bring up the Contract View of class *LINE* now; you will not see any mention of *id_generator*. You may only use this feature, unqualified, within class *LINE* itself. For example procedure *extend* uses (again check this for yourself) the assignment

← *"Definitions: Qualified and unqualified call", page 134.*

```
i := id_generator.generated_id
```

> *NONE* is the name of a special class. When studying inheritance we will see its place in the larger order of things.

→ *"Overall inheritance structure", 16.10, page 586.*

9.3 KINDS OF FEATURE

We have now seen all feature categories and are in a position to understand the overall classification.

The client's view

Viewed from the client's perspective, a feature of a class may either:

- Return a result: then it is a **query**.
- Return no result, but be able to modify the target object: then it is a **command**.

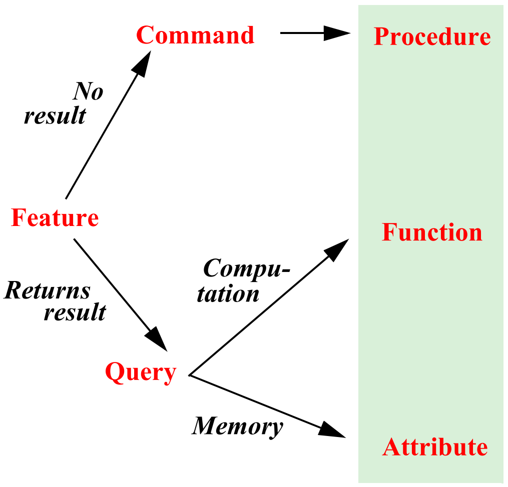
In the first case, there are two possibilities depending on how the class author has chosen to implement the query:

- You may choose to *store*, for every instance of the class, the value of the query in one of the instance's fields. This means implementing the query as an **attribute** of the class. It is then the responsibility of every command of the class to update the value of that field if it needs to — as, for example *forth* changes *index*.

- You may choose instead to *compute* the value of the query whenever requested, using an appropriate algorithm. Then you implement the query as a **function**. A recent example was the function *south_end*.

← *Page 238.*

The following figure represents this classification:



Feature categories (client's view)

"Memory" means that the value is stored, rather than computed. Note that the word "procedure" appears redundant at this stage, being synonymous with "command".

The notion of **query** is particularly important as a common category for attributes and functions. From the client's perspective, it does not matter that a query is implemented by storage or by computation. Although the difference between the two categories appears in the class text, it does **not** appear in the class interface. Bring up indeed the Contract View for *LINE* again; you can see, next to each other, one in the -- Access feature clause and the other in the -- Measurement clause, the interfaces for

```
index: INTEGER
        -- Index of currently considered station in line.
```

and

```
count: INTEGER
        -- Number of stations of this line.
```

They appear similar. But if you now look up these features in the actual class text (not the Contract View) you will see that the declaration of *index* appears as above, since it is an attribute, while the full declaration of *count* reveals it to be a function:

```
count: INTEGER
            -- Number of stations of this line.
    do
        Result := metro_stops.count
    end
```

Nothing in the Contract View suggests this difference. For the client, both feature are just queries.

The policy that treats attributes and functions identically in the Contract View of a class reflects a principle of software development:

Touch of Methodology: The Uniform Access Principle

It must make no logical difference to clients of a class, when they use one of its features, whether the class implements it by storage or by computation.

"Storage" covers attributes and "computation" covers functions. "No logical difference" means no difference of functionality; there might still be a difference of execution *efficiency*, as an attribute implementation takes up space, while a function does not but usually requires longer to execute than a simple field access.

The choice between the two solutions indeed involves space-time tradeoffs, explaining the importance of the Uniform Access Principle: it is very difficult to know ahead of time what solution will be best; during the course of a project you may have to reverse such decisions several times as a result of time and space measurements. The principle shields client software from these changes: the notation *some_object*.*some_query* will remain applicable throughout, so that you may try out various solutions without penalty. If access to attributes and functions used different syntax, you would each time have to update a much larger part of the software than necessary.

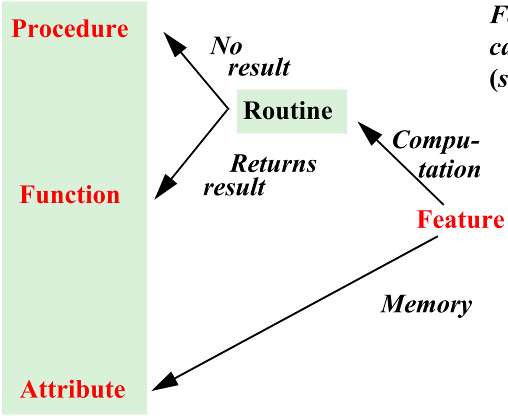
The principle further justifies the information hiding policy discussed:

- It is OK to make an attribute available to clients, as in *Line8*.*index*, especially since we make it available *not as an attribute* but more generally as a **query**: the client has no way to know, from the official interface description of the class, whether it is an attribute or a function.

- It is **not** OK, however, to let clients assign directly to it, as in the illegal *Line8*.*index* := *new_value*, since (among other problems) this would reveal that it is an attribute.

The supplier's view

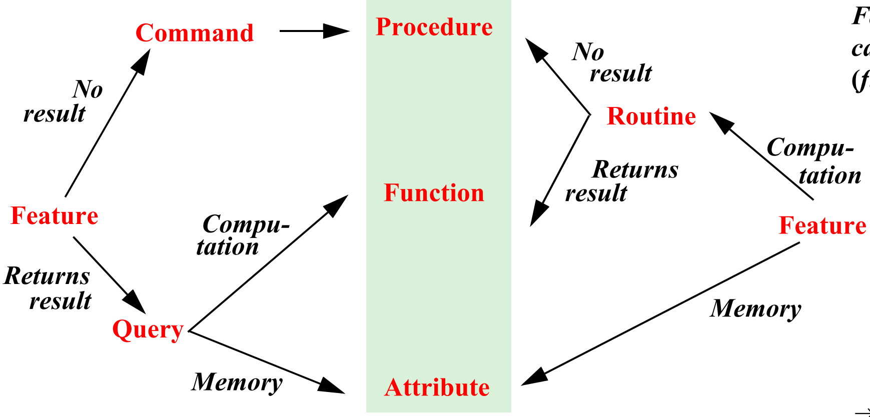
If we take the viewpoint not of the client but of the supplier class, in other words the implementer's internal perspective, we get the following categories:



Feature categories (supplier's view)

The only addition to the previous figure's terminology is the notion of Routine, covering both procedures (the term appears more justified now) and functions.

Putting the two views together, we get the complete picture:



Feature categories (full view)

→ *"Precise feature terminology", page 269* and exercise *9-E.1, page 269*.

You must know the precise definition of all the terms listed on this figure, and their role in building classes and making them usable by clients.

Setters and getters

A procedure such as *set_a* or *go_ith,* which has as its principal effect to set the value of an attribute, is called a *setter procedure* (or setter command).

In some programming languages it is also useful to write a *getter function* whose sole purpose is to return the value of an attribute:

```
current_index: INTEGER
            -- Position of cursor.
      do
            Result := index
      ensure
            same_as_attribute: Result = index
      end
```

Warning: *not necessary in Eiffel.*

Why would we ever need something like this since clients can simply use *index* is the attribute is exported?

Indeed there is no need for getter functions in the framework that we have seen. As we have seen, exporting an attribute such as *index* makes it available to clients in read-only mode: it lets clients *use* the value of *index* (as in *Line8*.*index*), not *change* it as this requires a setter procedure. So exporting a function *current_index* achieves exactly the same effect as exporting the attribute *index* in the first place.

Getter functions only become meaningful in languages such as C++, Java and C# where exporting an attribute means something else: the effect is to allow clients both to access and to change the variable, so that the assignment *Line8*.*index* := 98 [4] becomes valid. This mechanism is risky for the reasons analyzed earlier — it destroys information hiding — and should never be used. Hence the standard advice in methodology-conscious textbooks about these languages: do not export attributes; instead, if you want to let clients access them, shadow each attribute by a getter function, and export the getter function only. The C# language has a notion of "property" that provides standard syntax to achieve this.

→ *"Properties", page 782.*

Requiring programmers to use getter functions achieves information hiding in languages that do not fully enforce this principle. The disadvantage is that programmers discover a language mechanism (here the read-write mode of exporting attributes) together with standard advice not to use it; this casts aspersions on the language design. They can forget the advice and mistakenly break information hiding. If they do apply the advice, writing getter functions can be tedious and makes the code needlessly bigger.

It seems preferable to rely on the language design described previously, with the following property summarizing this discussion:

> ***Touch of Methodology*: Attribute Exporting property**
>
> Exporting an attribute is legitimate and lets clients access (but not modify) the corresponding field values.
>
> The interface of a class does not distinguish (in the absence of arguments) between an exported attribute and an exported function. To client authors, both kinds simply appear as *queries*.
>
> As a consequence of this policy, there is no need for getter functions.

9.4 ENTITIES AND VARIABLES

A bit of terminology cleanup will help finalize our understanding of the fundamental concepts associated with objects and classes. Everything regarding *features* should now be clear, but we have also encountered the terms *entity* and *variable*; let us make sure the concepts and terminology are entirely clear.

Basic definitions

We know what an entity is: an identifier that denotes possible run-time values. We are now in a position to list all possible variants of this notion:

← *In "Entities and objects", 6.2, page 109.*

> ***Definition*: Kinds of entity**
>
> An entity is one of:
>
> E1 •An attribute.
>
> E2 •A local variable of a routine, including the predefined local variable **Result**.
>
> E3 •A formal argument of a routine.
>
> E4 •**Current**, denoting the current object.

So if you were puzzled that *index* from *LINE* was sometimes referred to as a feature and sometimes as an entity, E1 is the explanation: it is both. To be precise, the identifier *index* is an *entity*; it denotes a *feature*. The feature called *index* is, more specifically, a *query*, and even more specifically an *attribute*.

As an entity, *index* is one more thing: a *variable*. Entities are indeed of two kinds:

- Some entities may change value during execution by serving as targets of assignments; they are called **variables**,. This includes local variables (E2) and one kind of attribute (E1), “variable attributes”.
- Others will retain a single value throughout execution and are called constant entities. They include formal arguments (E3), **Current** (E4), and the second kind of attribute, “constant attributes”.

The notion of variable deserves a definition of its own:

Definitions: variable, variable entity

A **variable entity**, or just **variable**, is an entity whose associated value may change during execution.

Variables include *local variables* and *attributes*.

As usual, local variables include **Result**.

Variable and constant attributes

Attributes may be either variable, as in all the examples seen so far, or constant.

Attributes declared in the usual form are variable, for example *index* in

```
index: INTEGER
```

You recognize a **constant attribute** by its declaration including the **=** symbol followed by a value. In *LINE* you may see (in a **feature** {*NONE*} clause towards the end) the declaration

```
First_id: INTEGER = 1000
```

This introduces the constant integer attribute *First_id*. Note the convention:

Touch of Style: Constants

For names of constant attributes, as for predefined objects, start with an upper-case letter, writing the rest in lower case.

This style is also common for strings, as in

```
Map_title: STRING = "Plan of the metro"
```

known as a **manifest string**.

Not being variables, constant attributes of any type may not serve as assignment targets: *First_id* := 2 or *Map_title* := "Something else" are invalid assignments (try them and watch for the compiler messages).

Constant attributes serve to give names to values that your program may need. You *should* systematically use this technique:

> ***Touch of Methodology*:**
> **Symbolic Constant Principle**
>
> When you need any specific values in a program — other than very simple values such as the integers 0 or 1 to start a loop or increment an index — do not use manifest values directly in the corresponding instructions; declare constant attributes with these values, and then use these attributes everywhere else.

So if you need a string for an error message, or a physical constant, do not use it directly in the instructions that need them, as in

```
display ("Could not send message in allotted time")

length := 2.54 * length_in_inches
```

Warning: *Not the recommended style.*

but declare

```
Timeout_message: STRING = "Could not send message in allotted time"

Inches_to_centimeters: REAL = 2.54
```

and write the instructions as

```
display (Timeout_message)

length := Inches_to_centimeters * length_in_inches
```

Two arguments supporting this rule are:

- Readability: the rule encourages you to give to each constant a name explaining its role in the software.

- Facilitating program evolution: the values of such constants may (although not in the second example) change during program evolution; you will only have to update the corresponding declarations. It is common to group all the important manifest constants of a program — for example, error messages such as "Could not send ..." above — in specific classes intended solely for this purpose; this further helps limit the scope of changes.

  Directly using manifest values in instructions would be a particularly bad idea for *strings*, such as this example, since a successful program often requires **internationalized** versions for various countries. In that case the actual strings typically come from external "resource files", making it possible to select the appropriate language version based on user preferences. Even then, it is generally desirable to retain default versions in the program as manifest strings; at run time the internationalization mechanism looks up the resource file for the language-specific version of any particular string and, if it does not find one, retreats to the default.

9.5 REFERENCE ASSIGNMENT

The values we manipulate — in particular object fields, corresponding to attributes of their classes — may be basic values such as integers and booleans, or references. So far we have applied assignment to basic values only; but we also need to assign references. That is in particular how we will build *linked* data structures, such as a list of metro stops where each stop contains references to the associated station and to the next stop on the line.

Building metro stops

Implementing the class *STOP* will require such reference assignments. The class interface included the following feature specifications

← *See the full specification on page 119.*

```
set_station (ms: STATION)
        -- Associate this stop with ms.
    require
        station_exists: ms /= Void
    ensure
        station_set: station = ms

link (s: STOP)
        -- Make s the next stop on the line.
    ensure
        next_set: right = s
```

indicating that the implementations must set the attributes *station* and *right* respectively. To provide these implementations we need assignment. Here are the routine texts (no longer just their interfaces):

```
     set_station (ms: STATION)
               -- Associate this stop with ms.
          require
               station_exists: ms /= Void
          do
               station := ms
          ensure
               station_set: station = ms
          end

     link (s: STOP)
               -- Make s the next stop on the line.
          do
               right := s
          ensure
               next_set: right = s
          end
```

A reference assignment **reattaches** the reference to a new object. It may previously have been void (attached to no object) or attached to another object (or to the same object, in which case the assignment changes nothing). To illustrate these possibilities, consider variables *s1* and *s2* of type *STOP* and two creation instructions

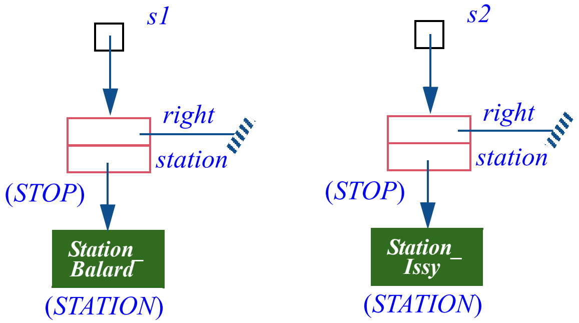
```
create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)
```

both using *set_station* as creation procedure; this is necessary since we had written the class *STOP* as

← *"Creation procedures", 6.5, page 122.*

```
class STOP create
     set_station
feature
     station: STATION
     right: STOP
     set_station (s: STATION) … As above …
     link (s: STOP) … As above …
invariant
          station_exists: station /= Void
end
```

The creation instructions produce two objects:



Creating two stops

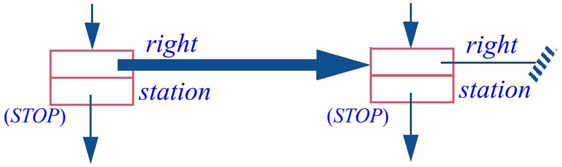
with *station* references attached (thanks to the creation procedure *set_station*) to two pre-existing *STATION* objects. The *right* references are void, since all reference attributes start out void and here *set_station* does nothing about *right*.

Building a metro line

To chain the two stops, you may use the instruction

```
s1.link (s2)
```

which updates the *right* reference of the first object



Chaining two stops

(Rest of structure unchanged from previous figure)

as a consequence of the assignment instruction in procedure *link*:

```
link (s: STOP)
        -- Make s the next stop on the line.
    do
        right := s
    ensure
        right_set: right = s
    end
```

This is an example of a reference assignment, which attaches a reference. Here the reference (the *right* field of the *STOP* object on the left) was initially void, and we assign to it a non-void reference *s2*; the effect is to attach *right* to an object, the second *STOP* object. We can also use reference assignment to make a reference void, for example by adding the following procedure to *STOP*:

```
make_last
        -- Make this stop the last one on the line.
    do
         right := Void
    ensure
        no_right: right = Void
    end
```

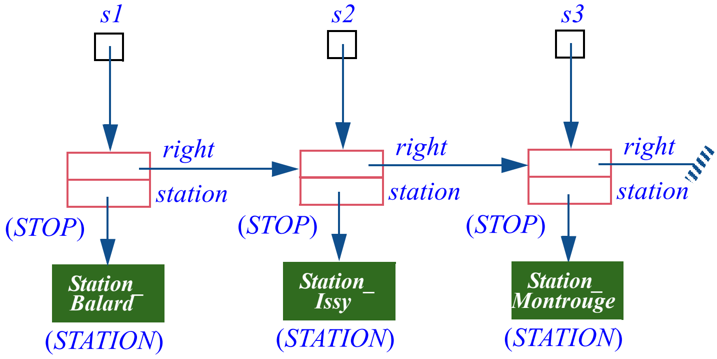
This uses the value **Void**, always denoting a void reference. The following three calls have the same effect (assuming that the value of *v* is void):

```
s1.make_last                s1.link (Void)              s1.link (v)
```

For more illustration of playing with references, here is the previous example again but with three stations rather than two (the additions are highlighted):

```
create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)
create s3.set_station (Station_Montrouge)
s1.link (s2)
s2.link (s3)
s3.make_last
```

The result is the following completion of the earlier figure, showing a metro mini-line:



Creating a small metro line

9.6 PROGRAMMING WITH REFERENCES

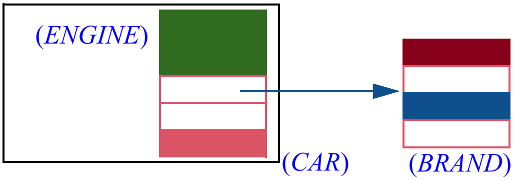
A reference is a value that identifies an object. Using references has several benefits, to be examined now: modeling the "knows about" relationship; supporting linked structures; providing a notion of Void to terminate such structures. We will also discover the darker side: how *dynamic aliasing* makes dealing with references a delicate proposition.

References as a modeling tool

An object may include a reference to another object to represent the concept of "knowing about" that object, which you may compare with the concept of *containing* another object. Contrast for example two uses of the verb "to have" about cars:

- A car *has* an engine.
- A car *has* a brand.

The key difference is sharing: two cars may have the same brand (say they are both Nissans); but no self-respecting car would consent to sharing an engine with another. With object-oriented techniques we may model the first case through a subobject, the second through a reference to another object.



Subobject and reference

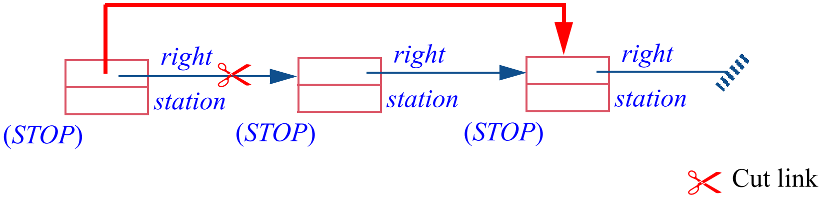
(*Expanded types* help model subobjects.) Such modeling flexibility is important in building programs that model complex systems.

Using references for building linked structures

Another application of references, highlighted in the examples of the previous section, is to represent collections of objects, also known as "containers", in a *linked* implementation made of cells where each cell may contain references to other cells. An example is a *linked list*, such as our metro line, a sequential structure where each cell but the last contains a reference (*right* in the figure on the preceding page) to the next element.

→ Linked lists are studied in detail in "Linked lists", 13.7, page 400.

Linked structures facilitate insertion and deletion operations; for example removing the second element of the mini-line structure involves reattaching a *right* reference. The effect on the earlier figure (reduced to its relevant elements) may be illustrated as follows:



Removing a cell to a linked structure

If this is to be an feature of class *STOP* — representing the operation to remove the next stop on the line — it will appear as the following procedure:

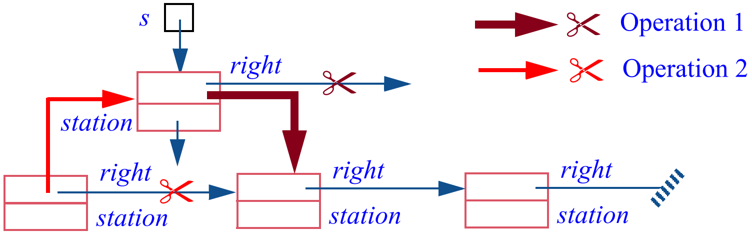
```
remove_right
        -- Remove following stop on line.
    require
        not_last: right /= Void
    do
        right := right.right
    ensure
        skipped_one: right = old right.right
    end
```

→ *For a sketch of a more general version see "Linked lists", page 400.*

An alternative design would be to remove the precondition and change the comment to "Remove following stop on line, if any". Then the routine body should become

```
if right /= Void then right := right.right end
```

Similarly, we may want to insert another stop between the current one and the next one if any:



Adding a cell to a linked structure

Here is the corresponding routine (again meant to be added to class *STOP*):

```
put_right (s: STOP)
        -- Add s as stop on line after current stop, retaining any later stops.
    require
        exists: s /= Void
    do
        s.link (right)    -- Operation 1
        right := s        -- Operation 2
    ensure
        linked_to_new: right = s
        retained_others: right.right = old right
    end
```

→ *For a more general (and more tricky) version see put_right, page 402.*

In this case the routine works whether *right* is initially void (the current stop was the last on the line) or attached. The new cell, *s*, has to be non-void; its previous *right* reference, void or attached, is lost through the application of *link*, but its *station* remains.

← *The text of link appears on page 254.*

As in the algorithm for swapping the values of two variables, the order of the assignments is essential: when we link *s* to its new neighbor (Operation 1 in the figure), this neighbor is represented by *right* which must have its original value, not the value reset by Operation 2.

← *Page 235.*

Procedures *remove_right* and *put_right* illustrate common schemes in manipulating linked structures.

> In most practical cases, the interface will be slightly different: an insertion operation will not take an existing list cell, such as a *STOP* here, as its argument. Instead its argument will be a list element, such as a *STATION* in this example; the operation will first create a list cell containing that element, then insert the cell into the structure. We will study such operations, including a more general insertion routine, in the discussion of linked lists.

→ *"Linked lists", page 400.*

Void references

The third benefit of references is the **Void** value, used in particular to terminate linked structures, as represented by the —⫽ symbols in the last figures.

As you know, this is a mixed blessing: the prospect that a variable *v* might have a void value at some steps of some executions complicates programming since it means we have to check every feature call *v*.*f* (…) for a guarantee that *v* will never be void for any execution of this call.

← *"The trouble with void references", page 112.*

This is the price to pay for the flexibility of describing linked data structures. It comes with methodological advice:

Touch of Methodology: Using void values

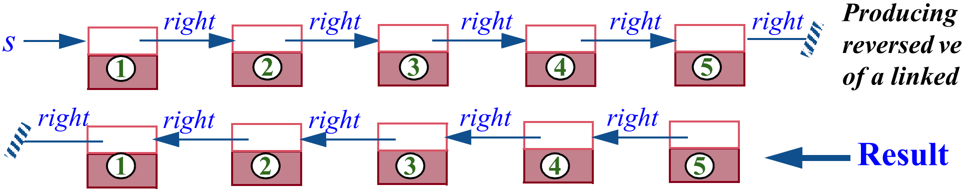
Reserve void references to the termination of linked structures.

This means in particular that when you are dealing with special values of a type, not with a linked structure, void values are generally not the right solution. Say you have a class *ACCOUNT* in a financial program, and you need to represent accounts that have no known properties. **Void** is not the solution; simply use a special object, "Unknown account". This removes the risk of applying a feature of *ACCOUNT* to a void reference, causing a run-time error. Of course you also do not want a feature call that executes but produce an incorrect result, so you must make sure that the features produce a meaningful effect on such special objects.

Reversing a linked structure

Procedures *remove_right* and *put_right* provide good examples of simple reference-manipulating operations on linked structures. Simple, but already not trivial; note in particular the constant need to worry about the possibility of void values.

To acquaint ourselves further with reference algorithms, let us move up one rung in sophistication and devise an algorithm for *reversing* a list. More precisely, to keep things still not too hard, we want to leave the original list unchanged and produce another list that has the same elements in the reverse order:



Producing a reversed version of a linked list

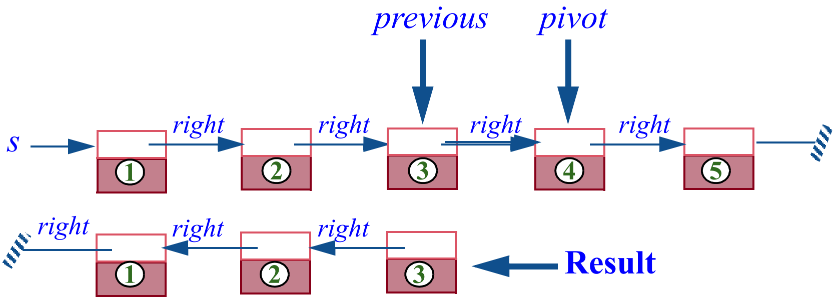
We start from *s*, a reference to a metro stop. Since every metro stop includes a reference to the next one, we can use *s* to access a full metro line by repeatedly applying *right*. We do not want to modify this structure in any way, but produce another one, which will be accessible through the **Result** of our function (bottom part of the figure); it contains the same elements but chained in the reverse order. To illustrate this, the figure shows the information associated with each *STOP* object (a station, itself a reference as illustrated in earlier figures) just as a number, **1** to **5**.

As often, it is a good idea to try to devise the corresponding algorithm for yourself before reading the following solution.

Performance is important in such algorithms. The first element of the new list will be the last element (marked **5** in the figure) of the original one; we can get to it only by traversing the full list. But then to get the second element of the new list, the next-to-last one of the original list, we would need to traverse it once more. This is very bad (it will give a total number of operations proportional to n^2, where n is the number of elements). Instead we want to traverse the list just once.

→ We will learn to be more precise in "Estimating algorithm complexity", 13.3, page 376.

As with any iterative algorithm, the key to designing the algorithm right — or to understanding it if the algorithm already exists — is to get the right **loop invariant**, specifying the properties of any intermediate situation. The figure below, a truncated version of the figure on the previous page, illustrates the situation we will obtain after a typical iteration of the (yet to be written) loop.



Producing a reversed version of a linked list: intermediate state

What we will have done at this stage of the algorithm is to produce a reversed form of *part* of the original list. Two variables — they will be local variables of the routine — tell us how far we have proceeded in that original list:

- *previous* points to the last cell that has been included in the reversed list.
- *pivot* point to the first cell not yet processed. It will be void on the last iteration, when we have processed all cells; indeed *pivot* = **Void** will be our signal that we are done — the loop's exit condition,

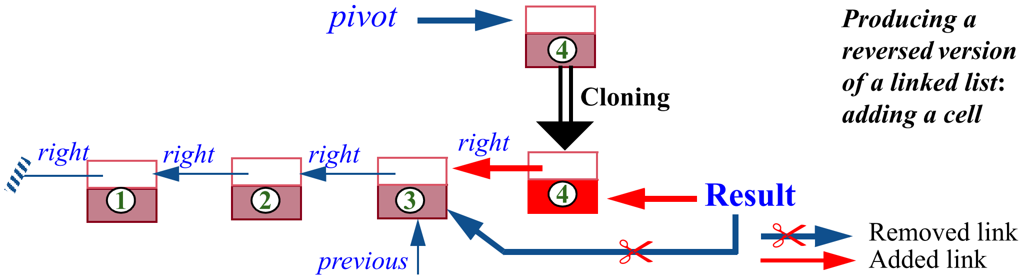
These two properties make up the loop invariant. The scheme is clear: at every iteration of the loop, process the next cell, known through *pivot*, by cloning it, chaining the result to *previous*, and attaching **Result** to it; then update *previous* and *pivot* by attaching each to the cell to the right of the cell it currently represents. This will re-establish the invariant. Here is the routine:

```
reversed (s: STOP): STOP
        -- New stop, the first in a new line that has the same stations
        -- as s but in the reverse order.
        -- (No precondition, will work for void s representing empty list.)
    local
        previous, pivot: STOP
    do
        from
            previous := Void ; pivot := s
        invariant
            -- The list starting at Result contains all cells of the original,
            -- up to and including previous; pivot denotes the next cell if any.
        until
            pivot = Void
        loop
            Result := pivot.cloned ; Result.link (previous)
            previous := pivot ; pivot := pivot.right
        variant
            -- See below.
        end
    end
```

We need the local variable *previous* to retain the previous *pivot* while creating a new cell; note its initalization to **Void**. The function call *a*.*cloned* gives us a new object (as with a creation instruction), duplicated field-by-field from *a*.

Depending on the library version you are using, you might have to use *twin*, the older name for *cloned*.

Here is a picture of what happens in the highlighted step:



Producing a reversed version of a linked list: adding a cell

You should check that the algorithm will always apply the qualified feature calls in the loop body, *pivot*.*cloned* and *pivot*.*right*, to a non-void *pivot*.

→ *"Recursive reversal", 14-E.6, page 502.*

An interesting exercise, when we have studied recursion, will be to rewrite the routine *reversed* so that it uses recursion rather than a loop.

Making lists explicit

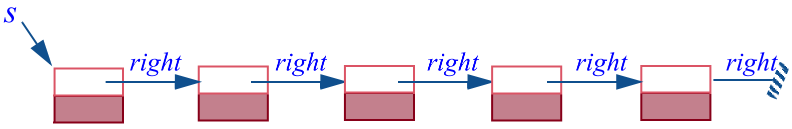
A later chapter will come back to linked lists to cover them more systematically. In particular, we will build a list reversal procedure to reverse an existing list in place, rather than a function such as *reversed* which creates a new list without affecting the original. Clearly such a procedure is more delicate to write, since while traversing a data structure it must modify the references in it, without messing up the remainder of the traversal.

→ *"Linked lists", page 400.*

> No new mechanisms are involved, so it is a good idea to try your hand now at writing such a procedure.

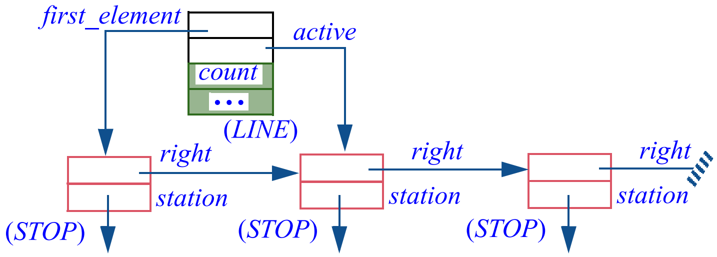
→ *"Reversal procedure", 9-E.6, page 270.*

This will also be the opportunity to provide a more general framework for linked structures than the one assumed, for simplicity, in the last few examples. We have worked on the class *STOP* whose instances each contain a link *right* to the next instance:



Implicit list structure

Access to a stop *s* gives us access, through successive applications of *right*, to the list structure of its successors; but the structure itself remains implicit. One of the consequences is the difficulty of expressing the loop variant of the above version: while the variant is intuitively clear — at step *i* we have produced a reversed version of the first *i* positions of the original — we have no simple way to refer to such global properties of the list since all we have is individual items. In a more systematic approach the list itself is an object:



A linked list

The list object is the instance of *LINE* at the top. It only serves as a "list header", containing no list values (this is the role of instances of *STOP*) but general information about the list, such as *count* (number of elements) and *first_element*, a reference to the first *STOP*, from which you can access all others.

> To implement the notion of list with cursors, which we have often used as it facilitates iteration, it suffices to include another reference in the list header: *active*, indicating the current cursor position.

This is the setup for the list structures of the EiffelBase library, where the list cells (*STOP* instances above) are instances of library classes such as *LINKABLE*, distinct from the list header classes such as *LINKED_LIST*.

It is a good exercise to rewrite the preceding examples — *remove_right*, *put_right* and *reversed* — as routines of the class *LINE*, where they belong more properly than in *STOP*. You should be particularly careful about void values and border cases (empty lines, insertion or removal at either end).

→ *"List of stops as a class", 9-E.8, page 270.*

Where to use reference operations?

Even though we have made our job easier by working on list cells directly rather than both lists and cells, and by avoiding the more delicate variants such as in-place reversal, the preceding examples provide a good idea of what it means to work with references.

This is, clearly, tricky business. You must take into account all possible cases including empty or almost-empty structures, track the state of the cursor if present, and pay special attention to the ever-lurking possibility of void references, as they may not be targets of feature calls. The more general issue, which we will look at next, is "dynamic aliasing", through which references make it harder to reason about programs.

← *As studied in "The trouble with void references", page 112.*

To be a good computer scientist or software engineer you must master these delicate techniques fully; several other presentations of reference-heavy algorithms in this book will help you towards that goal.

In the architecture of a system, such stuff is not for application modules:

Touch of Methodology: Reference Programming Principle

Non-trivial manipulations of references, typically, for inserting and removing items to and from object structures, should appear not in the application-oriented parts of programs but in library classes expressly devised to implement such structures, or (if no library is available to cover a specific need) in specialized clusters of an application program.

The "*application-oriented parts of a program*" are those dealing directly with the program's intent: processing calls (in the software running your cell phone), selling securities, typesetting text… This seldom directly involves tricky reference manipulations of the kind just seen, although they are often useful for the *implementation* of application-oriented concepts. Your text-processing system may use a linked list of paragraphs, but — unless there is something really special about lists of paragraphs as compared to lists of anything else — juggling with references to enter a new paragraph into the list is not a text-processing issue; it is a list issue, and should be handled in classes that deal with object structures in general.

If you must implement such manipulations yourself, the Reference Programming Principle directs you to separate them from the application proper, putting them into special "supporting technology" clusters.

Fortunately, modern development environments provide libraries of components dealing with the basic kinds of object structures; EiffelBase is an example, developed by many people over many years. Others are the Java and C# "collection" libraries, and the Standard Template Library (STL) for C++. A consequence of the above advice is an encouragement to use such libraries:

Touch of Methodology: Fundamental Data Structure Library Principle

For fundamental data structures and algorithms, use components from a basic library, if applicable, rather than developing your own implementations.

If you need a list, or a tree, or a stack, or a queue or any of the fundamental structures (many of them studied in the next chapter), check first whether a library mechanism is available and matches your needs. If so, do not reprogram it: use the library.

This advice extends beyond the specific issue of algorithms manipulating references: why redo when you can reuse?

You will occasionally find that you need a different interface from what a library class provides; then you have to provide your own variant. Just check that your needs are really different; you may be able to use the library by studying it more carefully and perhaps adapting your application software.

If you do have to write your own implementations, the library can still help you. Rather than starting from scratch, you may be able to start from an existing class and modify it. Remember, however, this book's general advice about code duplication. Copy-paste is a terrible idea in software development.

Inheritance often allows you to take an existing class, or several, and build a new one that extends, refines or adapts them, without modifying their own code and without copying that code. This is an interesting form of reuse, specific to the object-oriented approach; different from the form we have seen so far, in which you just become a client of a class and use it through its interface (its API); more delicate too, but powerful and often rewarding.

→ *Chapter 16*.

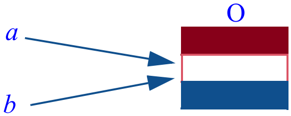
Dynamic aliasing

Let us come back to what makes references tricky. (You may treat this last subsection as supplementary material, not indispensable on first reading.)

In addition to the issue of voidness, references introduce the possibility of having more than one program name to refer to a single object. This is known as **aliasing**, as in the case of a person known through an alias (such as Mark Twain, an alias for Samuel Langhorne Clemens). Reference assignment gives us **dynamic** aliasing, determined by run-time events: assuming the reference *r* may be attached to some object O, the assignment

```
b := a
```

causes aliasing, with *a* and *b* ending up attached to the same object as illustrated below; but we cannot even reliably assume this information, since the assignment could take place in some executions and not in others, depending on the control structure.



Aliasing resulting from a reference assignment

What's wrong with dynamic aliasing? The issue is the added difficulty of reasoning about programs. We can express an operation on the object in terms of *a* only; we can express a property of the object in terms of *b* only. The operation may affect the property, but we might miss this in reading the program since the operation does not even cite *b*.

Such a combination of events disturbingly departs from a common and reassuring mode of reasoning. Consider the following scheme:

```
-- Here you know some property of b : P (b)                    [5]
OP (a)              -- An operation involving a, and not naming b
-- Can you assume that P (b) still holds here?
```

In programming, if *a* and *b* are distinct variables of basic types such as *INTEGER*, the answer to the last question is yes: nothing you do to *a* (say an assignment such as *a* := *a* + 1) can affect *b*.

With references and aliasing, this is no longer the case! Assume that their type is a class *STUDENT* and a procedure *raise* in that class increases the grade point average (query *gpa*) by .01; the precondition of raise should state this property. Now apply the above scheme

```
-- Here you know that b.gpa is less than 4
a.raise
-- What can you assume about b.gpa here?
```

which we can make into a routine with an explicit contract:

```
dubious
        -- Illustrate the perils of aliasing.
    require
        low_gpa: b.gpa < 4
    do
        … Possibly other instructions …
        a.raise
    ensure
        -- What about b.gpa?
    end
```

As before, the example assumes that 4 is the passing grade.

The answer to the final question depends on whether *b* is aliased to *a*. If not, executing the routine leaves *b*.*gpa* and all other properties of *b* untouched. But if *b* was aliased to *a*, the minute change to the GPA may have invalidated the property stated in the precondition, which then is not part of the postcondition. If *b*.*gpa* < 4 is part of the class invariant, whether *dubious* preserves it also depends on aliasing.

The issue is not semantic ambiguity. There is no doubt as to what will happen, depending on whether aliasing holds between *a* and *b*. The difficulty is how we **reason** about programs. In considering the effect of an instruction, especially scheme [5], we have to determine whether aliasing could hold; this prevents us from concluding, when we see that an operation does not name a variable, that it cannot affect any property associated with the variable.

What compounds the difficulty is that it does not know any borders; it extends across the modules of your program, and potentially the entire program. As long as aliasing only results from assignments as above, you can pretty much see what is going on in a small scope, a routine or at worst a class. But any passing of an argument to a routine, as in *r* (*b*) with *r* (*a*: *T*), introduces aliasing between *a* and *b*, just like assignment does in the case of variables. So as soon as you pass any of your references to a routine, which can pass it further, you have the possibility that a call in a remote part of the software will affect one of the objects that you think as being yours only.

This possibility of remote modification is often desired. The problem is to know precisely what is going on. We may note that the issue is largely one of specification. The earlier procedure would probably have been written, with a more specific expression of the actual intent, as

```
dubious
        -- Illustrate the perils of aliasing.
    require
        low_gpa: b.gpa < 4
    do
        … Possibly other instructions …
        a.raise
    ensure
        a.gpa = old a.gpa +Increment
    end
```

where *Increment* is the value added by *raise* (we assumed 0.1). This is the effect on *a*, as expected. What this specification does not say is what should *not* happen, such as an effect on *b*. You can of course add to every postcondition clauses of the form

```
        b = old b
        c = old c
         … and so on for every attribute of the class …
```

but that is not attractive.

The question is how to write practical contracts that not only specify how some properties will change but also list all the properties that do *not* change. The latter are known as **frame properties**, and the general issue as the **frame problem**.

This is still in part a research problem. There is no consensus on how to express frame properties — unlike the partial consensus, modulo differences of notations and scope, that applies to the expression of ordinary contract properties such as preconditions, postconditions and invariants.

A general discussion of the frame problem would extend far beyond this presentation, but you should remember that dynamic aliasing introduces significant difficulty in reasoning about programs, and consequently exercise great care in dealing with references and especially passing them around in your programs. This is one of the main reasons why, as discussed, non-trivial operations on references are best left to specialized libraries and clusters. You now know more precisely what “non-trivial” covers: any manipulations that could cause sneaky aliasing and produce surprising effects.

As a general note, it may be tempting to lay the blame specifically on the notion of reference and suggest we should do away with this mechanism. But the effect would probably be worse, as can be attested by the techniques programmers used in older languages (such as Fortran) which did not have references; they would simply put all their data in large memory structures (*arrays* as studied in a later chapter) and use integer indexes, pointing to positions in these structures, in lieu of references. The effect and the risks are the same, worsened by the need to work at a lower level of abstraction.

Aliasing, in fact, is not just a programming concept or a consequence of using references in modern programming languages. It is a by-product of the human ability to name things, and to give more than one name to one thing. *The Beautiful Daughter of Leda*, *Poor Menelaus's spouse* and *Paris's lover* are all references to the same person, also known as Helen of Troy. Traditional rhetorics had a rich classification for such "tropes" (metaphor, metonymy, synecdoche, allegory…).

As an example of their consequences, imagine that at the cafeteria a friend tells you that "*Judy is worried: her GPA is 3.96 — so close!*". From a conversation at a neighboring table you overhear: "*The jogging partner of my cousin's roommate did very well in the last exam. But because this is already in the last year it can at most raise the overall GPA by four tenths of a point!*". Does the thought occur to you that this might refer to Judy? Dubious, but with aliasing one never knows. Maybe you think you know Judy's jogging partner, and even the partner's roommate, but then there could be *dynamic* aliasing: while the people involved are unlikely to have a new student cousin since you last met them, they could have acquired new roommates or new jogging partners. So maybe Judy is all right after all, but it's hard to know.

Assuming numerical grades, where the passing grade is 4.

For once we cannot just blame programmers for their twisted minds; they both benefit and suffer, like everyone else, from the sophistication of human reasoning patterns.

9.7 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Except in functional languages, programs need to change the values of some of their entities, or *variables*. The primary technique is assignment.
- Assignment applies to both basic values, which it copies, and references, for which it copies only a reference, not the associated object.
- Reference assignment introduces dynamic aliasing: an object may become accessible through several names. This complicates reasoning about programs.
- Variables include attributes, representing fields of objects, and local variables, used only by a particular routine.
- References make it possible to describe and manipulate linked data structures, and perform sophisticated operations such as reversal.

New vocabulary

Assignment **Attribute** **Local variable** **Variable**
Temporary variable **Variable entity**

Precise feature terminology

Attribute **Command** **Feature**
Function **Procedure** **Query** **Routine**

9-E EXERCISES

9-E.1 Vocabulary

Give a precise definition of each of the seven terms in “Precise feature terminology” above.

9-E.2 vocabulary

Give a precise definition of all the terms in the above “New vocabulary” list.

9-E.3 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise “Vocabulary”, 8-E.1, page 225.*

9-E.4 Terminology

1 Is every function an entity?
2 Is every function a query?
3 Can a function be a query?
4 Is **Result** an attribute?
5 Is **Result** a feature?
6 Is **Result** an entity?
7 Is **Result** a variable?
8 Are all variables local?
9 Is every attribute an entity?
10 Is every routine a query?
11 Is every query an entity?
12 Is every attribute a variable?
13 Is every function a variable?
14 Is every entity a variable?
15 Can a query be a variable?
16 Can a function be a variable?
17 Is every variable an entity?

9-E.5 Swapping values

Assume variables *var1* and *var2* of type *INTEGER*, with the ordinary arithmetic operations. Can you write instructions that will swap their values, without using any local variables or any other entity? (The answer is an old programming trick; can you think of any limitation?)

← *As in "Swapping two values", page 235.*

9-E.6 Reversal procedure

In a class *C*, let the attribute *s* denote a reference to the first *STOP* object in a metro line. Write a procedure *reverse* in *C* that reverses the order of the stops in the line, so that *s* will upon completion denote the first stop in the modified line (the last one in the original). The procedure takes no argument. (Making *s* an argument of *reverse* would make things harder, since you cannot assign to a formal routine argument, although you can modify the corresponding object.) Hint: use the function *reversed* for inspiration.

← *Page 261.*

9-E.7 Chaining stops both ways

This exercise asks you to update the class *STOP* of this chapter so that every stop is linked to its left neighbor (if any) as well as its right neighbor. For example *link* should be called *link_right* and complemented by *link_left*.

← *Page 254.*

1 Add *put_left* to complement *put_right* and *remove_left* to complement *remove_right*.

← *Pages 257 and 258.*

2 Update *reversed*.

← *Page 261.*

9-E.8 List of stops as a class

Rewrite the linked-structure-manipulation routines of this chapter (*remove_right*, *put_right*, *reversed*, as well as *reverse* if you have also done the previous exercise) as features of class *LINE* rather than *STOP*.

← *See "Making lists explicit", page 262.*

9-E.9 Language rules

In an assignment *var* := *exp*, *var* must be a variable; it cannot be an expression involving a qualified call, such as *some_object*.*one_of_its_fields*. What is the justification for this rule? (**Hint**: refresh your knowledge of the information hiding principle and class invariants.)

PART II:

How things work

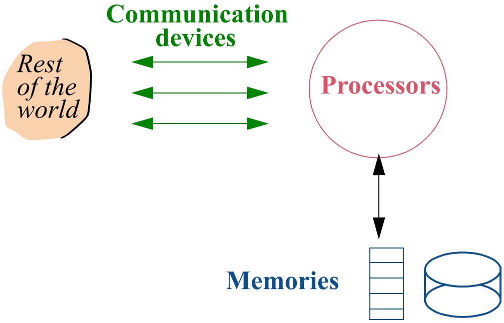
In this second part we take a short respite from software design and implementation techniques to explore some of the supporting technologies:

- *Computers* (going a bit further than in our original overview).
- How to describe the *syntax* of programming languages and other formal notations.
- Varieties of programming languages.
- The *tools* that support software development.

10

Just enough hardware

There would be no software without computers. To understand what it takes to develop good programs, we must have a basic understanding of the underlying machinery — the hardware — on which they will run. In this chapter we take a look at some of the essentials of what you must know about that hardware, detailing some elements of our earlier overall picture:



Components of a computer system

(*Figure from page* 7.)

In particular, we will get a feel for the order of magnitude of hardware phenomena: how much information you can represent through computer data, how fast you can access such data and execute operations on it.

We limit ourselves to properties of direct relevance to programmers and to the topics in the rest of this book. Along with learning to program, you should at some point take a course on a topic such as "Introduction to Computer Architecture", which will go far deeper into the details.

10.1 Encoding data

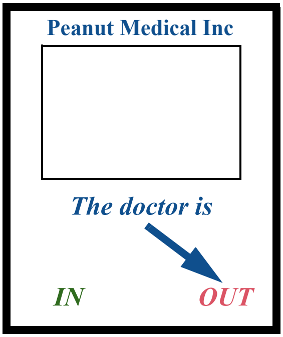
The data that we store in our computers' memories represents information of very diverse nature, from employee records, images and sounds, texts in human languages with formatting information (fonts, layout), to numerical values used in scientific computation — not to forget programs. We need a general way to represent this information and interpret the corresponding data.

The binary number system

Part of what made the computer revolution possible was the discovery of a simple and general way to represent information as data: the binary system.

The basis of the binary system is a set of two values (hence "binary"). These values have no intrinsic meaning, so we might call them Black and White, Tom and Jerry or maybe Isis and Osiris. What matters is that they must be unambiguously different. In fact we call them 0 and 1 (zero and one).

The term *bit* denotes a mathematical variable whose possible values are just these two. The word was made up by engineers in the late 1940s as a contraction of *binary digit*, to indicate that a bit is like a digit of ordinary arithmetic (0, 1, … 9) but with 0 and 1 the only possible values.



A bit (low-tech version)

"Bit" also denotes, by extension, a physical device that has two possible states, and hence can be used to represent a mathematical bit once we agree on which will be one and which zero. A cardboard sign on your door with a little flag that you can move to alternative messages "The doctor is IN" and "The doctor is OUT" is a bit. More relevant to the computer industry are electronic bits (obtained for example by transistors), where the two states correspond to two different voltages, and magnetic bits, for example small areas of magnetic tape or disk, where the states are "magnetized" and "demagnetized".

The reason why the binary system works so well is that today's electronics technology makes it possible to:

- Build such physical bit representations and pack many of them in a small area. To be more precise: pack *very* many of them in a *very* small area. We will see some numbers below.
- "Write" these bits (change their values) and "read" them (obtain their values) quickly. Very quickly.
- Build many such collections of bits, cheaply. Very cheaply.

These properties have ensured the success of the binary system. Some early computers used a decimal system; this seemed more natural since computers were then largely seen as counting machines, and when people count they will continue — computers or not — to use a decimal system for a long time, if only because we have ten fingers, not two or eight or sixteen. (The word digit itself comes from the Latin for "finger".) But for automatic computers built with the devices of electronics, the binary system long ago displaced its competitors.

To what extent is this relevant to programmers? More than you might think. True, we work on source programs expressed in a pleasant programming language, where the connection to information is clear, so that we write numbers, for example, in the usual decimal notation: 10, or –1, or 3.1415926524. But as soon as we consider how data is represented in memory, in particular where it is stored, we must remember that the binary numbering system is the one that is natural for computers even if strange to people at first.

This justifies taking a look at some of the properties of the binary system and its associates, although this is not a substitute for the more detailed knowledge you will gain from courses on logic and digital design.

Binary basics

Unless all the information you ever deal with is the result of a single toss of a coin, two possibilities isn't much. The basic combinations, out of which you can encode finite data of any size, are:

- The **byte**, or sequence of *eight* bits.
- The **word**, which on newer computers increasingly means a sequence of eight bytes, or *sixty-four* bits. In the past two decades "word" usually meant four bytes, or *thirty-two* bits (hence "32-bit architecture" and "64-bit architecture").

  Early on, the definition of "word" was not so standardized; computers used many different word lengths. You can still encounter outliers, but they are rare. Bytes have always been 8 bits and are also called *octets*.

How many possible values can such a sequence of bits represent? One bit has two possible values, 0 and 1. With two bits there are four possibilities:

0	**0**
0	**1**
1	**0**
1	**1**

More generally, for a sequence of n bits for any integer $n > 0$ there are 2^n (two to the power n) possibilities.

Basic representations and addresses

For the basic units:

- A byte, with eight bits, has 256 (2^8) possible values.
- A collection of 32 bits has 2^{32} possible values; that number, given in the table below, is on the order of four billion.

If, for example, we want to store characters making up a text, a possibility is to use one byte for each character. 256 possibilities might seem a luxury, but in fact it is just about what we need once we have included the ten digits, special symbols on your keyboard (such as ~, !, @ etc.), the 26 lower-case and 26 upper-case letters of the Roman alphabet, and the most common accented letters of Western languages (é, Ä and so on). The standard assignment of each possible 8-bit configuration to represent each one of these characters is known as extended ASCII. The original ASCII used only 7 bits (128 possibilities) and had no support for accented letters.

"American Standard Code for Information Interchange". You may see the ASCII byte code assignments at www.asciitable.com.

> Extended ASCII has several variants, but the most popular, known as ISO 8859-1, covers the characters used by the most widespread European languages.

For languages such as Cyrillic with other character sets, or ideograms as in Chinese, extended ASCII is not sufficient; the standard there is *Unicode*, which uses up to four bytes for a character, supporting a large set of possibilities that cover the most important written languages.

To declare character entities, you can use the type *CHARACTER_8* for extended ASCII or *CHARACTER_32* for Unicode. The more common solution is to use the type *CHARACTER*, which means one or the other depending on a configurable setting; this is what we will do in examples involving characters.

For numeric information, the common practice is to use a word to represent an integer variable. The mathematical set of integers is infinite, but in the memory of a computer we have room for only a finite set of values; using a 32-bit word, we can represent about two billion negative values and two billion positive ones, which is often enough. With 64 bits these limits are squared. The type for such data, in our programs, will be written *INTEGER*; in the same spirit as for characters, this can be set to mean either *INTEGER_32* or *INTEGER_64*. Types *INTEGER_8* and *INTEGER_16* are also available. If you are dealing with integer values that can only be non-negative, it is better not to use *INTEGER* and its variants but *NATURAL*, which covers integers from zero up and has the corresponding variants, from *NATURAL_8* to *NATURAL_64*.

The table appearing next on page 278 gives exact values.

A word can also serve to represent non-integer numerical values, mathematically corresponding to rationals, such as $3/2$, and other reals, such as π. Such values are particularly useful in "scientific computation", the use of computers for solving problems with a strong numerical component in physics, biology, engineering or even finance. The corresponding types in our programs are *REAL_32*, *REAL_64* as well as plain *REAL* which, again with the same convention, stands for one of the previous two. Unlike with integers, 2^{32} possible values often does not give enough precision; the 64-bit variant is usually the one you need for serious numerical computation.

The starting position at which a data element appears in memory is called its address. Examples of data types such as *CHARACTER_8*, *INTEGER_32* and *REAL_64* indicate that data elements may be of different sizes (in these cases one byte, four bytes and eight bytes). To provide a uniform way of denoting addresses, the convention is always to count in bytes, and to start at zero (rather than one). So if the memory starts with a thousand values of type *INTEGER_64* on a computer with 8-byte words, the first element that follows them will be at address 8000.

Powers of two

If only because of the property just seen (n bits can have 2^n values), the powers of 2 are important to the binary system.

The table on the next page lists the first ten values and other important ones in this sequence. You need to remember the first ten values, the order of magnitude of the others listed, and the "common" abbreviations (kilo etc.).

From cherries to bytes

In the ordinary, decimal way of counting things, abbreviations like "kilo" represent powers of ten, more precisely the powers of 10^3 which serve as natural milestones: a kilogram of cherries at the market is one thousand grams (10^3), one million dollars (10^6) will hardly buy you anything decent in Southern California, one billion dollars from the taxpayer (10^9) might prolong the life of a failing bank by a few hours.

n	2^n	Approximation by power of 10	Common name (abbreviation)	Official name (abbreviation)
0	0			
1	1			
2	4			
3	8			
4	16			
5	32			
6	64			
7	128			
8	256			
9	512			
10	1024	10^3 (thousand)	Kilo (K)	Kibi (Ki)
16	65536			
20	1,048,576	10^6 (million)	Mega (M)	Mebi (Mi)
30	1,073,741,824	10^9 (billion)	Giga (G)	Gibi (Gi)
32	4,294,967,296	4×10^9 (4 billion)		
40	1,099,511,627,776	10^{12} (trillion)	Tera (T)	Tebi (Ti)
50	1,125,899,906,842,624	10^{15}	Peta (P)	Pebi (Pi)
64	18,446,744,073,709,551,616	1.8×10^{19}		

This also applies to computer-related measurements other than memory:

- A transmission line functioning at 1 Mbps (Megabit per second) can transmit one million bits each second.

- A CPU with a speed of 1GHz (one Gigahertz) can execute one billion basic processor instructions per second. “Hertz”, number of events per second, is a frequency measure borrowed from physics.

While memory sizes and addresses are expressed in *bytes* (abbreviation **B**), transmission speeds are usually given in *bits* per second or **bps**, where the abbreviation for “bit” is **b**. So a “56K modem” — if functioning at its highest rate, which it usually does not — would transmit 56,000 bits each second.

To express memory size, computer engineers prefer to use the powers of two. This is where the mess begins; to be precise it began when someone (in whose honor no statues have been erected, as his or her name is lost to history) made the clever observation that the tenth power of two, 2^{10}, is 1024, slightly over 10^3 — a thousand — and the truly brilliant decision, as a consequence, to reuse decimal abbreviations shown in the highlighted entries of the table: kilo for a near-thousand (2^{10}), mega for a near-million (2^{20}, about 10^6), and giga for a near-billion, (2^{30}, about 10^9), although the approximation is less and less "near" as we go on; see the exact values in the table.

Having a binary interpretation of "thousand" and its multiples along with the traditional decimal one can be quite confusing, especially since the two are sometimes used together: the standard capacity of a "floppy disk", an older memory medium, is specified as 1.44 megabytes (MB), but means 1440 (1.44 times one decimal thousand) times 1024 (one binary thousand) bytes!

To end this confusion, which has led to lawsuits accusing manufacturers of misrepresenting memory capacities, the relevant standards organization prescribes retaining the older names — kilo, mega, giga … — in their traditional decimal meanings only, and using new names, shown in the last column of the table, for their binary neighbors; kibi, mebi, gibi… These names have not, however, gained widespread use.

The organization is the BIPM (Bureau International des Poids et Mesures), which defines the international system of units, or SI.

This is putting it politely. A more frank assessment as of 2009 is that no one uses them. While a Google search for "gigabyte" leads to countless ordinary uses, the first few hundred links for "gibibyte" are all to discussions of the term itself.

To avoid confusion, remember that binary interpretations are only used for memory measurements. For anything else, the usual decimal meanings apply. If the ad for that 1-GHz laptop, which executes a billion operations per second, also says it has 1 GB of memory, you will actually be getting more than a billion bytes; about 73 million more.

In most practical cases the difference does not matter: between friends, what's a few million?

Computing with numbers

The good side of the standard representation of integers on a computer is that they are exact: the integers that your programs can manipulate directly represent integers as we know them from mathematics. The bad side is that they are partial: they only cover a finite subset of mathematical integers. For a 64-bit computer the highest and lowest representable integers are in the neighborhood of 2^{63} and $–2^{63}$.

The exact values of these extremes depend on the computer's *number system*, in particular how it encodes negative integers. A common system is *two's complement*, where the binary representation of $–n$ is that of n with every binary digit reversed (0 for 1 and 1 for 0). Then the extremes, for M bits, are $2^{M–1}$-1 and $–2^{M–1}$.

For applicable integers, not only is the representation exact: arithmetic operations give the same results as their mathematical counterparts — as long as that result fits. Writing $a + b$ in your program, for integers a and b, will give the correct result, except if it would be over the maximum or under the minimum. Such a case is known as an arithmetic overflow.

Like the representation of integers, the representation of real numbers — the type is generally called either *REAL*, as in Eiffel, or float — is finite and hence partial; unlike for integers, it is not exact for most values. This would be impossible since there is an infinity of mathematical reals between any two distinct ones, for example in the interval [0, 1]. In fact the rational numbers (values of fractions of which the numerator and denominator are both integers) already have that property, even though they are a much smaller subset of the reals; so even the rationals cannot all be represented exactly. This is a typical example of the distinction between information and data.

← *"Definitions: Data, information", page 8.*

The standard representation of reals uses three parts stored in a word: a bit s representing the sign; an exponent, representing an integer n; and a fraction f, representing a real number whose leftmost digit is not zero (the fraction is "normalized"). This represents the number $f \times 2^n$, of sign s.

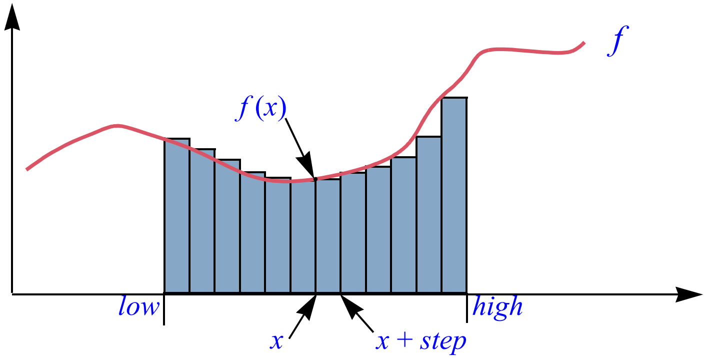
These properties affect your programs in three important ways:

- When you specify a real value x explicitly in the source text, or the program reads x at run time from a file, a sensor or another source, the value actually used is one of the representable floating-point values, very close to x but not guaranteed to be identical.
- Arithmetic operations — addition, multiplication, division, exponentiation — may fail even when their result is mathematically well-defined. **Overflow**, mentioned earlier for integers, can also occur here: if x and y are two mathematical reals both representable in the computer's number system (and assumed positive for these examples), $x + y$ or $x \times y$ may be too large to be representable. Another example, specific to real numbers, is the division x / y, where y is not zero, so that the operation is mathematically defined, but very small, making the result too large to fit in the number system. Real numbers also introduce the risk of **underflow**: with y very large, x / y may be too small in absolute value to be represented,
- Even in the absence of overflow and underflow, arithmetic operations can cause errors. If x' and y' are the representable values of x and y, the programming language notation $x + y$ does not denote the mathematical sum of x and y; it may not even represent the exact value of the mathematical sum $x' + y'$, which is not guaranteed to be representable. Any algorithm that deals with floating-point numbers must take this risk into account, lest it produce grossly erroneous results.

This is an important concern in "scientific computations", applied not only to science and engineering but also, for example, to financial modeling, with heavy use of real numbers and numerical algorithms. The error in each individual operation is generally acceptable, since it is so small, affecting for example the least significant digit. The risk arises in computations performing millions or billions of elementary operations: small errors may accumulate to the point where they seriously affect the validity of the results.

Numerical programming requires careful techniques to avoid numerical disasters. Here is a simple example. In a later chapter we will encounter the example of an integration routine that approximates the integral $\int_{low}^{high} f(x)\, dx$ of a function f over a given interval $[low..high]$ by the sum $\sum_{i=0}^{n-1} f(low + i \times step) \times step$ of the areas of rectangles of fixed width *step*, with $n = (high - low) / step$:

→ *"Agents for numerical programming", 17.4, page 634.*



Integration by finite approximation

We can implement this through a loop (reduced to its simplest elements, see the full context in the later discussion), using the local variable *x* of type *REAL*:

```
from x := low until x >= high loop
    Result := Result + f.item ([x])        -- f.item ([x]) is the value of f at x.
    x := x + step                                                              [1]
end
```

Warning: *initial version, numerically bad.*

In principle this does the job, but note how the highlighted instruction updates *x* on each iteration by adding *step*. This introduces a small error which, accumulated over a very large number of iterations, could cause a significant drift of the value of *x* and a seriously wrong result for the overall computation.

Some programmers instinctively use forms such as **[1]** because of the expectation that additions will be faster than multiplications, but this is not necessarily true and the effect on numerical precision is damaging. A direct implementation of the above Σ formula, with an integer local variable *i*, uses multiplication:

```
from x := low until x >= high loop
     Result := Result + f.item ([x])-- f.item ([x]) is the value of f at x.
     i := i + 1 ; x := low + (i * step)                  --                  [2]
end
```

Recomputing *x* from scratch each time removes the drift: for each value, we get at worst the error of a single addition and multiplication.

There is a general principle here:

Touch of Methodology: Computing with real numbers

In software that deals with computer representations of real numbers, be aware of the approximations involved, and devise the algorithms so that they will avoid *accumulation* of approximation errors.

A shorter form of this advice is “study numerical analysis” — the part of applied mathematics that deals with computing with actual numerical values (as opposed to symbolic computation), taking into account the properties and limitations of number representation and operations on actual computers.

In the treacherous land of numerical programming there is one beacon of comfort: standardization. Preventing mathematically correct algorithms from numerically misbehaving used to be a machine-specific task, as each computer architecture had its own number system. This situation has largely come to an end with the wide adoption of the IEEE Standard for Floating-Point Arithmetic. The standard defines a single framework for computer number systems, with both 32-bit and 64-bit variants. Most of today’s computer architectures implement it; so when you need to check that an algorithm does not cause unacceptable numerical errors you can at least expect to do this work only once.

→ Reference under “Further reading”, 10.5, page 291.

10.2 MORE ON MEMORY

Memory is where we put and access the data. At the most elementary level it holds basic data elements such as characters and integers, but to our programs it will be the place where we create and find objects. Let us see what memory can do for us.

Persistence

The diagram of computer organization shows two symbols for memories, ▤ and ⛁, to emphasize that some kinds are transient and others persistent, supporting data with different requirements:

← *Page 273.*

- Transient data is created and manipulated by a program's execution, but is not guaranteed to survive that execution. With some memory technologies, powering off the memory unit will result in loss of data.
- Persistent data remains forever unless expressly deleted; switching power off has no effect on this property.

Why have transient data at all? It might be simpler to make all data persistent by default, and delete what we do not need any more. The answer is technological and economic. Memories that processors can access at an appropriate speed for their data-processing operations are transient and expensive; persistent memories are cheaper, so that we can (thankfully) use them to store large amounts of data — representing text, images, music, flight tables, personal information … — which we have to access more slowly.

Words like "appropriate" or "slower" speed, "large" or smaller amounts of data, "cheap" or not, should be put in context. Here are some rough estimates in current technology (at the time of writing):

- A computer suitable for software development, possibly a laptop, might have a transient memory capable of holding a few GB (gigabytes) at a cost of a few hundred dollars or euros for the memory. The time to access a character might be around 50 nanoseconds, meaning 20 million accesses per second. (A nanosecond or ns is 10^{-9} seconds.)
- The computer might have a **disk** (persistent memory) that is a hundred times as big (a few hundred GB), costs half as much ($100), and has an average access time on the order of 5 milliseconds — two hundred accesses per second.

The ratio of access times is remarkable, and directly relevant to the programmer. Programs that manipulate large amounts of data cannot ignore the issue of their distribution between transient and persistent memory; they must keep transfer times under control, for fear of damaging execution speed.

→ See below a transposition to the human level in "Registers and the memory hierarchy", page 287.

Transient memory

Processor operations, as noted, will access data in transient memory. This key component of any computation has several names:

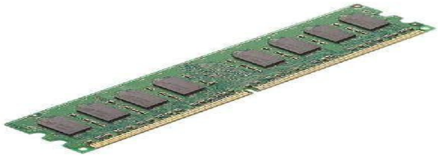
- *Main* memory.
- *Primary* memory.
- RAM, for *Random Access Memory*.

  The term has a historical origin: initially, non-primary memory was implemented with technologies such as magnetic tape or, even earlier, *punched tape* (paper tape with holes to represent bits, where the presence of a hole can mean 1 and its absence 0; in such cases data access is *sequential* in the sense that elements are accessed one after the other, in the order of their appearance on the tape. Starting from the beginning, the time to access an element is proportional to its address (since you must first go over all the ones before it). In contrast, main memory is "random", meaning that it takes the same time to access any element, regardless of its address. Many non-primary memories, such as the disks presented below, are now random as well, but "RAM" has stuck.

  → *See also the discussion of sequential and random-access structures in chapter 13, including the figure "Sequential and random access", page 380.*

- *Core* memory, or just "core". This term points back to an older technology, the little magnetic elements or "ferrite cores" of many years ago, but you will still hear that a certain set of data is "in core". Just understand "core" as meaning central, as in "core competence".

The photograph below shows a main memory "chip" containing 2 GB; its "DDR2-800" technology ensures 5 nanoseconds cycle time and 800 million transfers per second, with a peak transfer rate of 6.4 gigabytes per second.



A memory chip

Varieties of persistent memory

Persistent memory really consists of two kinds:

- Some elements are intended to remain attached to a computer during its operation. They are called **secondary memory** to emphasize that they really serve as extension of the primary memory. On the plus side (in addition to the benefit of persistence), the access cost per gigabyte is lower, making it possible to offer much more space, starting in the hundreds of gigabytes; the drawbacks are that access is slower and processors cannot directly get to the data: any transfer must go through primary memory.

- Others are meant to be connected to a computer only episodically, so that data can be copied onto them; then they can be removed, and later on connected again to the same computer or to another, to read the data back. They serve as devices for data "backup" (long-term preservation and storage) and interchange. They are called **removable memory**, or removable storage devices. ("*Storage*" is just a synonym for memory.)

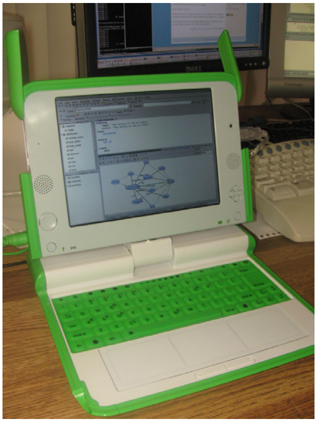
The most common form of secondary memory is the disk. A more correct term is "disk device" since "the disk" on your computer is actually a pile of magnetized disks, all rotating at a speed of 4,000 to 12,000 runs per minute, with reading heads that can move back and forth over disk surfaces to access the data, a bit being represented by the magnetized or demagnetized state of a tiny area. If power is switched off, the heads obviously will not work but magnetization is preserved, making disks suitable for persistent data.

A disk

The device shown above has two disks, although we see only one, and can store 8 gigabytes. (If this sounds underwhelming that is because it is a 1999 model — I was not going to rip apart my newest disk drive just to take a picture. At the time of writing you can get a disk of several hundred gigabytes for something like $50, and terabyte disks are available in consumer shops.) It turns at a rate of 5400 RPM (rotations per minute) with an access rate of 9 ms (milliseconds) and a maximum transfer rate of 33 MB (megabytes) per second. The access time and transfer rates are only an approximation; one of the characteristics of disk access is that a *latency* time is necessary to get the head to the right position; after that, you can access even a large amount of data much faster if it is all contiguous. When writing programs that make heavy use of disk data, you may have to take this property into consideration to optimize performance.

While disk drives remain the dominant kind of persistent memory, they are increasingly challenged by "solid-state drives" (SSDs) using *flash memory.* Unlike disks, SSDs are purely electronic (not electro-mechanical) and do not include any movable parts; they are not subject to the "disk crashes" of traditional disk technology, which without warning will lose all your data. A disadvantage of flash memory is that it supports only a set number of rewrites, although a number of techniques are available to cope with this limitation. Around late 2008, SSDs became an attractive alternative to disks, even for laptops, thanks to the fast increase in capacity and decrease in price.

The MIT Media Lab's XO laptop, introduced in 2007 as part of the "One Laptop Per Child" (OLPC) project, was one of the first computers destined for wide availability and using flash memory rather than a disk.



"One Laptop Per Child", running EiffelStudio

Following in the tradition of paper tapes mentioned earlier, some memory devices are removable. Among the most popular are USB memory sticks, so called because they connect to the standardized "Universal Serial Bus" of a computer and use flash memory technology; capacities of 2 to 16 GB are now common. USB disks — normal disks as described earlier, but connecting through a USB port rather than internal wiring and hence removable — start around 100 GB.

Memory stick and USB disk

An earlier device connection technology still supported by many laptops is mostly notable for its acronym: "PCMCIA".

For "Personal Computer Memory Card International Association" although you may prefer the unofficial version: "People Can't Memorize Computer Industry Acronyms".

Registers and the memory hierarchy

Computer operations such as addition generally require their operands, or at least some of them, to be in special locations called *registers*. Most architectures offer at most a few dozen registers. To perform an operation on operands stored in ordinary memory, for example *a* and *b* in the assignment

```
a := a + b
```

you may need the following sequence of hardware instructions: transfer the values of *a* and *b* from ordinary memory to registers; apply the operation (here an addition); transfer the result to ordinary memory, here back to the word dedicated to *a*.

The basic memory hierarchy has, as a result, three levels:

- Registers, very few in number, but accessible at the speed of the CPU clock.

- Core memory, typically a few gigabytes today, slower.

- Disk or equivalent, into the hundreds of gigabytes or terabytes, again slower.

The orders of magnitude of typical access times at the time of writing are: 0.5 nanoseconds; 50 nanoseconds; 5 milliseconds. While all this may appear very fast, the ratios are significant. Consider in particular that the ratio between the last two, disk and memory is around 100,000. Transposing to the human level, imagine a worker with:

- Operations (the processor instructions) to process materials, if available for processing (meaning, in the registers), in 0.01 seconds.

- Space (the main memory) to hold a large but limited amount of materials available within 1 meter (taking one second to grasp any item), but stored away at the end of any working session.

- Space (the disks) to store permanently an essentially unlimited amount of materials, but a hundred kilometers (and one day of transport) away!

For actual hardware the times should be divided by about twenty million but the ratios remain the same. The memory policy — what remains in main memory and what has to be fetched from disk — obviously has a major effect on performance.

Virtual memory

In practice the distinction between main memory and disk is blurred by the availability of *virtual memory*, a facility that operating systems provide to let you pretend that you have more core memory than you paid for.

Virtual memory allows you to use an address space much bigger than the available core memory through behind-the-scenes management of the physical placement of your data, keeping some of it in core and the rest on disk. For this purpose, it splits the address space into units called *pages*, each typically a few kilobytes. Actual data access requires that the data be in core; if it is not, the access causes a *page fault*: the virtual memory system loads the corresponding page from disk, an operation called *page-in*. This also requires, if no room is left in core memory, moving one or more pages to disk: a *page-out*.

The reason this policy can work efficiently is that programs typically use, over extended periods of their execution, a small subset, the *working set*, of their potential address space. Only under extreme circumstances does execution enter a degraded mode, known as *thrashing*, where page-ins and page-outs come to dominate execution.

Virtual memory also facilitates sharing a computer between many different programs: each program sees a single, continuous (and large) address space, which in reality is mapped onto a set of pages, interspersed in core and disk memories with the address spaces of other programs.

All this, of course, makes it more difficult to estimate the performance of programs, since you do not really know when a seemingly innocuous memory access might suddenly cause a page fault; it is as if once in a while our metaphorical worker, while reaching out for a piece of material, suddenly realized that none remains and he has to travel the hundred kilometers to the depot. In most practical cases, however, the working set properties of programs enable you to ignore this concern, and pretend that you really have (say) 20 GB of memory even if you only installed 4 GB.

10.3 COMPUTER INSTRUCTIONS

In ordinary situations the instructions of concern to programmers are those of higher-level programming languages. It is useful, however, to have an idea of the actual instructions into which they will be translated, the only ones that computers can actually execute.

A typical computer instruction is stored into a word, or occasionally several words. It contains an instruction code, determining the type of the instructions, and zero or more arguments, which can be addresses or other values. Here for example is an instruction for the 32-bit PowerPC instruction set architecture; the instruction occupies a word, with bits indexed from 0:

0						6					11					16					21	22									31	Bit position
0	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	Bit value
Instruction code						Result					Operand 1					Operand 2						Secondary code (266)										Role

The instruction code is the combination of a primary code **31** (binary **11111**) in bits 0 to 5 and a secondary code **266** (binary **100001010**) in bits 22 to 31. The result (bits 6 to 10) will go into register 5 (binary **101**); the operands (bits 11 to 15 and 16 to 20) are in registers 3 and 4 (binary **11** and **100**).

The bit-by-bit representation, as used here, is inconvenient for practical manipulations of binary numbers. The customary notations are hexadecimal (base 16, collecting bits by groups of 4) and octal (base 8, groups of 3). The above binary number **01111100101000110010000100001010** is **7CA3210A** in hexadecimal (where **A** to **F** are the hexadecimal digits for decimal 10 to 15).

Computers offer instructions of three main kinds:

- Computation: arithmetic operations such as addition (as in this example), subtraction, multiplication, division, usually in both integer and floating-point variants; comparisons; bitwise operations (like boolean operations, but applied to entire words, for example by "**and**"-ing the bits at matching positions in two words). Usually such operations take their operands in registers.

← *"Boolean operations", 5.1, page 72.*

- Load and store: transfer values from memory to a register, or the other way around; exchange values to and from hardware devices.
- Program flow: divert execution to a specific location, either conditionally or unconditionally. We saw examples in the discussion of control structures: branch unconditionally, branch on equal.

← *"Conditional and unconditional branching", page 182.*

Each instruction has a numerical code, such as **31** in the above example. A more convenient way of referring to instructions is through mnemonic codes; the addition instruction for PowerPC has the code **add**. More generally, the final form of machine-code programs, expressed in binary, is not suitable for human communication. **Assembly languages** provide a human-readable form of such programs. In PowerPC assembly language, the example instruction reads

```
add r5, r3, r4
```

An assembly language resembles programming languages in its use of symbolic names: register names such as r5, instruction codes such as **add**, as well as identifiers used to denote addresses and constants. The translation from assembly language to binary machine code is the responsibility of a system program called an *assembler*, a kind of elementary compiler. But assembly language remains in direct correspondence with machine code, unlike programming languages which offer advanced constructs at a much higher level of abstraction. Also note that an assembly language is applicable to just one type of computer architecture — Intel "x86", PowerPC or any other — whereas modern programming languages are portable (platform-independent). Rather than a programming language in the full sense of the term, an assembly language is just a tool removing the most tedious aspects of writing machine-level programs. Correspondingly, assemblers are much simpler programs than compilers.

The observation to remember here is, once again, the crudeness of the basic computer instructions: how remote they are from any useful task for which we wish to use computers. This gap justifies the need for programming languages and explains the difficulty of programming.

10.4 MOORE'S "LAW" AND THE EVOLUTION OF COMPUTERS

In considering the influence of hardware performance on programming, it is impossible to take a timeless view. The extraordinary characteristic of information technology has been the constant progress of hardware power. A 1965 article by Gordon E. Moore, co-founder of Intel Corporation, described this phenomenon in a particularly vivid way. The most common formulation of what came to be known as *Moore's law* is that the number of components crammed into integrated circuits at a constant cost doubles every 18 months (although Moore himself mentioned two years). There are a number of variants, which all point to exponential progress. They do not reflect an actual "law" of nature such as those of Newton, Maxwell or Einstein, but an observation about the industry's progress over several decades — an observation that has proved prescient and remains remarkably applicable. No other area of technology has ever experienced even remotely comparable growth. (Cars today are not 1000 times faster than cars twenty years ago; US cars, specifically, get about the same gas mileage in 2009 as in 1909.)

While the original Moore's law characterizes component integration and its immediate effect is on processing speed, variants of the phenomenon have governed the evolution of many aspects of computing power — memory size and access speed, disks, costs of various devices — at different speeds of evolution. I remember being amazed (little more than a decade ago) when the price of a megabyte of disk storage went below one dollar. Today few people would accept paying a buck for a *gigabyte*.

The basic Moore's law cannot be sustained indefinitely, as the packaging of ever more components into a fixed space raises issues of heat generation and approaches physical limits of signal transmission speed. This has resulted in a situation where (as some computer architects put it) "*the number of people declaring the demise of Moore's law doubles every eighteen months*". In fact the end is not coming just now, but the industry consensus is that the only viable long-term solution is to continue increasing **concurrency**: speeding up computation not just by making the processors faster but by using several processors in parallel. "Multicore" and "manycore" architectures are already pervasive. The problem, however, is that no satisfactory solution has emerged for programming these concurrent architectures. Not one more word on this topic (or, in this chapter, any other, as this is the last section): it is fodder for some other book.

But see the article by John Markoff in "Further reading" and, for a taste of current research, se.ethz.ch/research/scoop.

10.5 FURTHER READING

IEEE Standard for Floating Point Arithmetic (754-2008), available at ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935.

> "Specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments".

John Markoff: *Faster Chips Are Leaving Programmers in Their Dust*, in *New York Times*, December 2007, available at tinyurl.com/5cstbq.

Full URL: www.nytimes.com/2007/12/17/technology/17chip.html?scp=1&sq=john%20markoff%20intel%20concurrency&st=cse

> A newspaper article, not a scientific publication; presents a clear description of the need to rely on concurrent architectures to sustain Moore's law and the difficulty of concurrent programming. John Markoff has for many years covered Silicon Valley for the NY Times and is a well-known figure in the industry.

John L. Hennessy and David Patterson; *Computer Architecture*, Fourth Edition: A Quantitative Approach, Morgan Kauffmann, 2006.

> Classic textbook on software architecture, updated to reflect the latest trends, in particular the move to parallel architectures.

Patterson (2007)

10.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Internal representations of data on a computer use the binary system.

- The basic unit of data is the bit, with two possible values, 0 and 1. Bits are grouped into bytes, of 8 bits, and words, generally of 8 or 4 bytes (64-bit and 32-bit architectures). Addresses are measured in bytes. An integer or real number is commonly stored in a word; more compact variants (one or two bytes) also exist. A character occupies a byte (extended ASCII) or, with increasing prevalence, two or four bytes (Unicode).

- Measurements of quantities other than memory, for example speed, always use decimal units, with standard prefixes of the international system of measures such as kilo (thousands), mega (millions), giga (billions), tera (10^{12}) and peta (10^{15}). For memory size and addresses the common practice, departing from official standards, is to use the same prefixes for neighboring powers of two, starting with 2^{10} (1024, close to $10^3 = 1000$).

- Integer representation on a computer is exact, but only on a finite interval.

- Real number representation is approximate. Arithmetic operations usually cause an approximation error; the implementation of numerical algorithms must avoid accumulating these errors.

- The memory hierarchy includes registers, core memory and persistent devices such as disks or flash memory. Processor operations apply to operands in registers. Register access is the fastest (less than a nanosecond), but only a small number of registers are available. Core memory is (today) typically on the order of gigabytes, with access time about 100 times slower, and is transient (turning off the computer results in a loss of values). External memories — disks, flash — are again orders of magnitude slower than core memory, but support higher capacities (hundreds of gigabytes to terabytes and more) and provide persistence (values are retained).

- Machine code provides low-level instructions: simple operations on values in registers; transfer between memory levels; and transfer of control.

- The computer industry has benefited from an exponential increase in computer power, expressed as “Moore’s law”. Sustaining this development increasingly requires multiple processors and concurrent programming.

New vocabulary

Address
Bit
Byte
Core
Disk
Flash memory
Kilo
Giga
Hexadecimal
Mega
Multicore (and manycore)
Moore’s Law
Octal
Persistent
Primary memory
RAM
Read
Register
Removable memory
Secondary memory
Storage
Transient
Word
Write

10-E EXERCISES

10-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

10-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise “Concept map”, 9-E.3, page 269.

10-E.3 Measurements

How many bytes exactly is:

- One kilobyte
- One megabyte
- One megaword (1 word = 4 bytes)
- One gigabyte?

10-E.4 Your new laptop

The computer catalog advertises a laptop with 1.3 GB of memory:

- How many bytes, exactly, does the memory contain?
- How many bits, exactly, does the memory contain?
- Assume you use the entire memory to represent a single variable. How many possible values can that variable have? You are not asked to write down the exact number (*hint*: do not try unless you own a paper factory) but the best approximation you can of the form 10^n for some n.
- If you *did* want to write the number on paper, 100 digits per line and 60 lines per page, how many pages would you need?

10-E.5 Size and transmission speed

You must transmit 128MB of data using a 128-Mb modem working at full capacity. How many seconds will it take?

10-E.6 Octal arithmetic

On the Web you will find converters between decimal, octal and (next exercise) hexadecimal arithmetic. Do not use them to do these exercises, but do use them to check your results.

Octal arithmetic uses base 8, with digits 0 to 7.

- What is the octal representation of the decimal number 300000?
- What decimal number does the octal number 74223 represent?
- Compute the sum of these two numbers using octal arithmetic (decimal addition rules transposed to octal). Give the result in both octal and decimal.

10-E.7 Hexadecimal arithmetic

Hexadecimal arithmetic uses base 16, with digits 0 to 9 and A to F.

- What is the hexadecimal representation of the decimal number 300000?
- What decimal number does the hexadecimal number A42D3 represent?
- Compute the sum of these two numbers using hexadecimal arithmetic (decimal addition rules transposed to hexa). Give the result in both hexadecimal and decimal.

11

Describing syntax

With the study of control structures and assignment we have started to encounter language constructs with an elaborate syntax structure, which programmers can nest within one another. Other syntactically interesting concepts will follow.

To reason about such constructs, we need to define their syntax. The presentation of control structures relied on informal English descriptions: "A Conditional consists of the keyword **if** followed by...". Such a style, useful for introductory explanations, cannot give us as a general specification technique: it is too verbose and at the same time not precise enough. We need the reverse: conciseness and mathematical rigor.

← *For example "Syntax: Conditional", page 180.*

BNF (Backus-Naur Form) satisfies these requirements and is the main focus of this chapter. Other topics include: related techniques for describing *abstract* syntax; a sketch of how to develop a *parser* from a syntax description; some historical background; and an introduction to the theory of *finite automata*.

11.1 The role of BNF

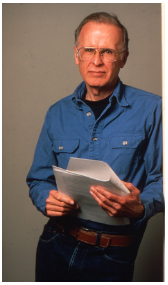
We saw that the full description of a language includes several levels: lexical, syntactic and semantic. BNF only addresses the specification of syntax.

← *Chapter 3..*

Before proceeding, make sure the basic syntax concepts introduced in that earlier discussion are fresh in your mind: construct, terminal, nonterminal, specimen, syntax tree.

Touch of history: The original BNF

The history of programming languages starts in the nineteen-fifties. The first language to achieve widespread recognition was Fortran (originally FORTRAN, for FORmula TRANslator), intended for scientific computation and designed by a team led by John Backus at IBM in 1954, with the compiler shipping in 1956. This success sparked the design of many new programming languages.



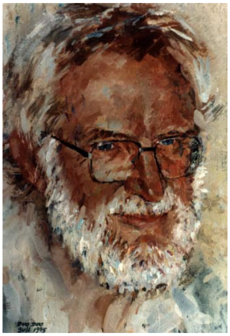
Backus

Soon American and European groups joined forces to design an international standard language which became known in 1958 as Algol 58. (The name stands for *ALGO*rithmic *L*anguage, and was ALGOL in upper case.) Its most influential version was the next one, **Algol 60**.

The preparation of the Algol 60 specification uncovered the need for better ways of describing syntax than the largely informal techniques used thus far. John Backus, by then a member of the Algol committee, proposed a notation for describing the language, which became known as Backus Normal Form, the original BNF.

A 1964 letter to the *Communications of the ACM* from Donald Knuth (a professor of computer science at Stanford) suggested acknowledging the contributions of another committee member, Peter Naur from Denmark, by retaining the acronym but making it stand for "Backus-*Naur* Form".

Many variants of BNF have been proposed since then. In the specification of his Pascal programming language (a descendant of Algol, first published in 1960) Niklaus Wirth devised a graphical variant which has also been widely used.



Naur (1995), portrait by Duo Duo Zhuang

Languages and their grammars

For our purposes a language is simply a set of "phrases", where each phrase is a finite sequence of **tokens** from a certain "vocabulary". For example in the Eiffel language a phrase is a class text, such as

```
class A end
```

the simplest one we can produce, made of just three tokens (two keywords and an identifier). The phrases encountered in practice — texts of useful classes — have many more tokens.

Not every sequence of tokens from the vocabulary of a language is a phrase of that language: **end** *A* **class class** is not a class text. To define the syntax of a language is to specify which token sequences are phrases, and which are not. Such a specification is called a **grammar**:

Grammar

A grammar for a language is a finite set of rules for producing, from the language's vocabulary, sequences of tokens such that:

1 Any sequence obtained by a finite number of applications of the rules is a phrase of the language.

2 Any phrase of the language can be obtained by a finite number of applications of the rules.

← This is a more detailed version of the original definition on page 40.

So by applying the rules we get all of the desired language (clause 2 of this definition) and only the desired language (clause 1).

Most languages of interest are potentially infinite; for example there is an infinite set of possible Eiffel class texts. But that theoretical possibility does not cause any practical problem, first because in our lifetime we will only deal with a finite set of programs, but more importantly because every *phrase* — here every class text — is itself a **finite sequence** of terminals. The sequence might be very long, but it cannot be infinite.

With a finite set of rules and an infinite language, we would have to keep applying the rules forever to produce all possible phrases, for example all Eiffel class texts. That again need not bother us. We do not need all classes; we only need those of interest to us — once we know that the rules are capable in principle of describing every possible class.

BNF is a notation for defining grammars. It is an example of a **metalanguage**: a *language* serving to describe other *languages*, such as programming languages.

> BNF and the other techniques of this chapter apply not only to programming languages but to all *formal languages*: artificial notations with a rigorously defined structure. HTML, the format of Web page texts, and XML, a general-purpose format for structured data, are examples of formal languages that are not programming languages in the usual sense. In fact the original research was directed at understanding *natural* languages — whose complexity and irregularity exceed, however, the modeling power of BNF.

→ See at the end of this chapter: "Touch of History: Classes of languages and grammars", page 318.

BNF basics

To describe grammars we will use a form of BNF called BNF-E, which serves in particular for the standard description of Eiffel. There are many other variants, such as Extended BNF (EBNF) defined by the International Standards Organization. "BNF" in the rest of this discussion means any BNF variant; any property specific to BNF-E is signaled as such. The differences are matters of style rather than substance.

BNF enables us to define a grammar for a language. *A* grammar, not *the* grammar, since different grammars may yield the same language.

→ We'll see an example in "Recursive grammars", page 307.

A BNF grammar consists of the following parts, each a finite set:

- A finite set of **delimiters**; as we have seen these are the basic, fixed tokens of the language's vocabulary, such as keywords (**class**, **if** …) and special symbols (period, colon, …).

← "Tokens and the lexical structure", page 43.

- A finite set of **constructs** representing structures of the language. Examples include Class, representing class texts, and Conditional, representing conditional instructions. The BNF-E convention is to start construct names with an upper-case letter and write them in Green. As you will remember, a particular instance of a construct is called a **specimen** of the construct; for example any conditional instruction is a specimen of Conditional.

← *Page 40.*

- A finite set of **productions**, each associated with a particular construct and specifying the form of its specimens. A production for Conditional, for example, defines the form of any conditional instruction: first the keyword **if**, then a specimen of Boolean_expression and so on.

Each production defines the syntax of specimens of a particular construct, in terms of other constructs and delimiters. Here for example is the production for Conditional:

Conditional ≜ **if** Then_part_list [Else_part] **end**

This says that any specimen of Conditional — any conditional instruction — consists of the keyword **if**, a delimiter, followed by a specimen of the construct Then_part_list, followed optionally (the brackets signal an optional component) by a specimen of the construct Else_part, followed by the keyword **end**. The constructs Then_part_list and Else_part have their own productions.

Every production defines a single construct, here Conditional, appearing to the left of the symbol ≜, read "**is defined as**"; the BNF expression on the right specifies the structure of the construct's specimen. This use of productions enables us to distinguish between the two kinds of construct:

- A construct defined by a production of the grammar is a **nonterminal**.

- Other constructs are **terminals**, for example (in a grammar for Eiffel) Identifier, whose specimens are identifiers such as *PREVIEW*, and Integer, whose specimens are natural integers such as 34. The grammar does not define terminal constructs, so we must look elsewhere for their syntax; it is described at the **lexical** level.

← *"Levels of language description", page 44.*

The notions of terminal and nonterminal construct are not new; we saw them earlier in relation to abstract syntax trees, where terminals represent leaves and nonterminal represent internal nodes.

← *"Abstract syntax trees", page 41.*

The reason for treating certain constructs as terminals and defining their properties outside of BNF is pragmatic: these constructs have a simple structure for which the full power of BNF (aimed at the description of potentially nested and complex structures such as those of classes and instructions) would be overkill. An identifier, for example, is simply a sequence of one or more characters, of which the first has to be a letter and any subsequent ones are letters, digits or underscores. This can be expressed easily by lexical techniques studied in a later section of this chapter.

→ "The lexical level and regular automata", 11.6, page 311.

Viewed from the syntax level (the BNF grammar), specimens of terminal constructs are just lexical tokens, like delimiters. Unlike delimiters, each of which defines a fixed single token such as a keyword or special symbol, most terminal constructs, such as Identifier and Integer, have an infinite set of possible specimens; this makes no difference to the BNF grammar, which does not concern itself with the contents of tokens but treats them as atomic units. The role of the grammar is to define, for every nonterminal construct, the form of the construct's specimens, as a combination of terminals and delimiters.

← Tokens are the basic lexical units, as recalled above: "Languages and their grammars", page 296.

Of particular interest for any particular language will be the nonterminal describing the top-level structures, such as classes in Eiffel. Such a nonterminal, Class in this example, is called the **top construct** of the language. The *phrases* of the language — here class texts — are the specimens of the top construct.

Distinguishing language from metalanguage

The production for Conditional illustrates that a BNF grammar includes symbols of three distinct kinds:

- **Metalanguage** symbols: those of BNF itself, serving to express the productions. In the Conditional example they are $\triangleq$ and the brackets [ ] signaling an optional part.
- **Language** elements, directly belonging to the language being described, for example — if the language is Eiffel — the keywords **if** and **end** in a production for Conditional and delimiters such as **:=** in a production for Assignment.
- Names of **constructs**, both terminal and nonterminal; they belong to the metalanguage where they denote elements from the programming language. Terminals, as noted, denote tokens. Nonterminals denote syntactic structures; for example every specimen of Conditional is a syntactic structure, itself containing substructures such as a specimen of Then_part_list.

Because of this mix of symbols from language and metalanguage, we must be careful to avoid confusion. Typographical conventions help:

- Metalanguage elements (the symbols of BNF-E itself) appear in black.
- Names of constructs, such as Conditional (nonterminal) and Identifier (terminal), appear in green.

- Special symbols appear — like all programming language elements in this book — in **blue**. But this is not quite enough for symbols like brackets which would be easy to mistake for a metalanguage symbol. So they will be enclosed in straight quotes: for example "**:**" denotes a colon as it will appear in the Eiffel text; and a production for any Eiffel construct that uses an opening bracket will denote it as "**[**" to avoid any confusion with the metalanguage bracket introducing an optional part.

- For delimiters of the other kind, keywords such as **if** and **then**, we do not need quotes because keywords are always written in **boldface blue**, avoiding any confusion. So we just let the keywords stand for themselves.

The term *specimen*, which may have surprised you the first time around, is similarly intended to avoid confusion. A specimen of a construct is a language structure that satisfies the properties of the construct, for example a particular conditional instruction. The word "instance" would capture this notion, but it is already used to denote run-time objects corresponding to a class. An instance of a class is not the same thing as an instance of the construct Class! (One is a run-time object, the other a program text.) Using "specimen" for constructs avoids the ambiguity.

← *"Grammar, constructs and specimens", page 39.*

11.2 PRODUCTIONS

A production defines the syntax of specimens of one construct. It is of the form

Construct ≜ *Definition*

where the left-side Construct states the construct being defined, and *Definition* specifies the syntax, in terms of constructs — terminals and nonterminals — and delimiters. Depending on the form of the *Definition* there are three kinds of production: Concatenation, Choice and Repetition.

Concatenation

A **Concatenation** production lists zero or more constructs in a certain order, some possibly enclosed in brackets [...] and then said to be **optional**. Our first production, Conditional, was an example:

Conditional ≜ **if** Then_part_list [Else_part] **end**

Such a production specifies that every specimen of the construct on the left of the ≜ consists of a sequence ("concatenation") of specimens of each of the constructs listed on the right, in the order given, except that the specimens of any of the optional constructs may be missing. In the example, every specimen of Conditional consists of the concatenation of: the keyword **if** (a terminal); a specimen of Then_part_list; optionally, a specimen of Else_part; and the keyword **end**.

> "Concatenation" simply means the linking of two or more elements as in a chain — *catena* in Latin. The word is often used in programming: to *concatenate* two character strings is to join them into a single string. Its use for BNF is a bit pretentious, as we could talk of "*sequence* productions". But in the programming language we also have *sequences* of instructions, our first control structure. Again to avoid confusion between language and metalanguage, we use "Sequence" for the Eiffel construct and "Concatenation" for BNF productions. In a similar way the Choice productions evoke conditionals, and the Repetition productions evoke loops, but the terminology is distinct to avoid confusion. The analogies are, however, significant, and will be explored further below.

→ *"Turning a grammar into a parser", 11.5, page 311.*

Choice

A **Choice** production lists one or more constructs, separated by vertical bars |. An example is the production defining instructions:

```
Instruction ≜ Conditional | Loop | Compound |
              Assignment  | Call
```

A Choice production specifies that every specimen of the construct on the left of the ≜ consists of exactly one specimen of one of the constructs on the right. (Unlike for Concatenation productions, the order of their listing is irrelevant.) In the example, a specimen of Instruction is a specimen of either Conditional, or Loop etc. In ordinary language, we would say "An instruction is one of: a conditional, a loop, a compound, an assignment, a call".

→ *This is only an example, omitting a few of the kinds of instructions available in Eiffel.*

> We may indeed from now on say "An X", for some construct name X, as an abbreviation for "A specimen of X"; for example: "A Conditional".

Repetition

Finally, a **Repetition** production lists two constructs on the right of the ≜ : one a nonterminal to be repeated; the other, usually a terminal, serving as separator. We may for example specify compound instructions (instruction sequences) as

← *"Sequence (compound instruction)", 7.4, page 147.*

```
Compound ≜ {Instruction ";" ...}*
```

meaning: a specimen of Compound is made of a succession of zero or more specimens of Instruction, each separated from the next, if any, by a semicolon. According to this rule, possible specimens of Compound are of the forms:

- Nothing at all (Repetition of zero Instruction specimens)
- *inst1*
- *inst1* ; *inst2*
- *inst1* ; *inst2* ; *inst3*
- etc.

where *inst1*, *inst2*, *inst3* are instructions.

> We saw that in Eiffel the semicolon is optional. Although this property can be expressed through the grammar, it is more convenient to use the above production and add a non-BNF tolerance rule stating that a missing semicolon is harmless.

← *“Compound: syntax”, page 149*.

The asterisk — a well-established symbol from the mathematical theory of formal languages — means “zero or more”; the three dots suggest repetition; the braces { } are just for grouping.

With this production, the repetition may be empty; Eiffel syntax indeed allows for an empty Compound. This is convenient for such cases as

```
if some_condition then                                    [S1]

else
    instruction_1
    instruction_2
end
```

where the empty **then** part is legal since syntactically it is a just an empty Compound. (In terms of programming style this is not a tidy structure and if it is to persist you should clean it up, for example to

```
if not some_condition then                                [S2]
    instruction_1
    instruction_2
end
```

but the first form can be useful when you are moving instructions around, in the process of updating your program, and a Compound like the **then** part of [S1], previously containing instructions, temporarily finds itself empty.)

For some constructs an empty repetition is not desirable. Then you will use a variant of the Repetition that instead of the asterisk “∗” uses a “+”, also a standard symbol from mathematical language theory, meaning “one or more”. Here for example is the production for Then_part_list, given with the other constructs related to conditional instructions (which we can now see in full since all types of production have been introduced):

Conditional ≜ **if** Then_part_list [Else_part] **end**
Then_part_list ≜ {Then_part **elseif** …}$^+$
Then_part ≜ Boolean_expression **then** Compound
Else_part ≜ **else** Compound

The Repetition production for Then_part_list indicates that a specimen of this construct — a more complete name would be “Then and possibly Elseif part list” — is of one of the forms

- *cond1* **then** *inst1*
  -- One specimen of Then_part

- *cond1* **then** *inst1* **elseif** *cond2* **then** *inst2*
  -- Two specimens of Then_part

- *cond1* **then** *inst1* **elseif** *cond2* **then** *inst2* **elseif** *cond3* **then** *inst3*
  -- Three specimens of Then_part

and so on, for boolean expressions *cond1*, … and instructions *inst1*, … Note that the Then_part_list is not optional in the Concatenation production for Conditional, so there will always be at least one Then_part, of the form *some_condition* **then** *some_compound*; if there are two or more, they will be separated by **elseif** as shown.

Rules on grammars

In BNF — any variant — an obvious rule on productions is that every component appearing on the right side (the *Definition* of a construct) must be one of: a delimiter; a terminal construct; a nonterminal construct.

This corresponds to the three sets involved in a BNF grammar: delimiters, terminals, nonterminals. To write a grammar in practice it suffices to list the productions, which define these three sets through simple conventions:

← *“BNF basics”, page 297.*

1 **Delimiters** are self-describing, with the conventions defined: keywords **stand out**, special symbols appear "in quotes".

← *"Distinguishing language from meta-language", page 299.*

2 Any other identifier appearing in a production denotes a construct.

3 If such a construct appears on the left side of at least one production, it is a **nonterminal** (since the production defines a structure for its specimens, in terms of other elements).

4 Otherwise the construct is a **terminal**. In that case its definition must appear in the lexical part of the language specification.

Case 3 assumes that a given nonterminal may appear on the left of more than one production. This is permitted by most BNF variants other than BNF-E, with the convention that two separate productions for the same construct

- A ≜ *Def1*
- A ≜ *Def2*

are to be interpreted as a single production involving a Choice:

A ≜ *Def1* | *Def2*

Alternatively or in addition, such BNF variants allow mixing the various production mechanisms — Concatenation, Choice, Repetition — in a single production, as in

A ≜ B | C [D] {E ";" …}*

Warning: Not in BNF-E. See next.

BNF-E disallows such mixing of production styles:

***Touch of Methodology*:**
BNF-E rule

- Every nonterminal must appear on the left side of exactly one production, called its **defining production**.
- Every production must be of one kind: Concatenation, Choice or Repetition, following the rules defined above.

So for the example given you must use three productions:

A ≜ B | Concat
Concat ≜ C [D] Repet
Repet ≜ {E ";" …}*

The correct form.

Along with a few notational conventions, this rule is the specificity of BNF-E among BNF variants.

In writing language definitions, I have found that while the rule leads to introducing more nonterminals — such as Concat and Repet here, and Then_part_list in the earlier example — it yields more understandable language descriptions.

> It also permits a better assessment of **language size**. If you can stuff different mechanisms into a single production, you might give the impression of a small language whereas the language is actually quite complex. Since you cannot do that with BNF-E, the number of productions is a good indicator of actual syntactical complexity, since the extra nonterminals do represent real concepts.

11.3 USING BNF

We have now seen all of BNF. The following pragmatic observations will help you apply the techniques effectively.

Applications of BNF

BNF descriptions enable you to:

- Understand the syntax of existing languages, in particular (but not only) programming languages.
- Define the syntax of languages you need to design.
- Write syntax analyzers, or *parsers*.

The second application is not as far-fetched as it sounds. Although you may not have to design a general-purpose programming language — a competitor to C, Java or Eiffel — as part of your first job, programmers have to define "*little languages*" all the time. Whenever you write a program that will process some data, the format of the data is a language, and if that format is not trivial BNF is often the right way to define it. Exercises in this chapter ask you to write the BNF specifications of such examples.

The third application, parsing, is useful for writing **compilers** and other tools that process programs and, more generally, structured texts. One of the first tasks of such a tool is to reconstruct the structure of the text — in the form of an **abstract syntax tree** — from the external appearance of the text. This is what the parser does, as we will see in the next chapter. Any parser needs a formal description of the syntax of the language it is supposed to parse; it will get that description from a BNF grammar.

→ *"Compiler tasks", page 336.*

Language generated by a grammar

We may see a BNF grammar in two complementary ways, following from the two clauses in the definition of the notion of grammar:

← "Grammar", page 296.

- It is a *recognition* mechanism, making it possible to determine whether a certain sequence of terminal specimens and delimiters is a phrase of the language, and if so to reconstruct its syntactic structure. This is the view of interest when you are, for example, writing a parser.

- The grammar is also a *generation* mechanism: by applying its rules you may produce, one after the other, all the phrases of the language.

The second view is less often useful in practice but important all the same. Let us explore it a bit further. To produce all the possible specimens of any nonterminal — in particular the top symbol — it suffices to look up the production defining it (remember that in BNF-E there is only one):

P1 For a Concatenation, produce all possible sequences of specimens of the constructs listed, plus those where optional ones are ignored.

P2 For a Repetition, produce all sequences of zero or more (one or more in the case of "+") specimens of the construct listed, with the given separator in-between.

P3 If at any of the previous steps you encounter a nonterminal, apply the same process to it so as to produce its own specimens.

P4 For a Choice, apply the previous steps to all the constructs listed, and collect all the specimens that you have produced from any of them.

These four phrase-generation mechanisms, carried out as long as at least one of them is applicable, will eventually yield all the phrases of the language. The process is generally non-terminating since, as we have seen, most languages of practical interest are infinite.

In carrying out this process for a nonterminal A whose production uses B, you may have to apply the same rules — in steps P3 and P4 — to other constructs.

Recursive grammars

The last observation may cause some alarm. What if, applying the process to A, we have to apply it to another construct B and in so doing we encounter A again?

The production for Compound provides a good example. It reads:

```
Compound ≜ {Instruction ";" ...}*
```

involving the construct Instruction; but that construct is itself defined as

```
Instruction ≜ Conditional | Compound | ...Other choices ...
```

involving Compound, as well as Conditional whose definition also uses Compound. If we try to understand Compound by looking for its specimens according to the rules given above, we will need to determine the specimens of Instruction; but this will require us, by the same rules, to look for specimens of Compound. This seems like circular, meaningless reasoning.

Such a definition, appearing to define a concept in terms of itself (directly or, as here, indirectly) is said to be **recursive**. Recursion — the use of recursive definitions — pops its head in almost all areas of programming and we will have an entire chapter devoted to it. But since there is almost no useful grammar without recursion we should already convince ourselves that such grammars can actually make perfect sense.

→ *Chapter 14.*

We can study this on a small example. Consider a mini-language with three keywords **heads**, **tails** and **stop**, no other terminals, and Game as its top construct defined by the following grammar involving one Choice and two Concatenation productions:

```
      Game ≜ Head_start | Tail_start | stop
Head_start ≜ heads Game
Tail_start ≜ tails Game
```

This is similar to the situation with Compound, Instruction and Conditional: three nonterminal constructs defined in terms of each other.

Quiz time!

Can you devise examples of specimens of Game in the language defined by the above grammar? What are the specimens of Head_start and Tail_start? (Do the quiz before turning the page, since the answer appears overleaf.)

Because of the recursion the grammar might seem meaningless. But let us take a pragmatic view by asking if we can use the grammar, through the construction process described above, to **generate** specimens. Notice that in the production for Game one of the branches, **stop**, is a terminal. So we can generate a first phrase (a first specimen of Game):

← *P1 to P4, page 306.*

- **stop**

But now since both Head_start and Tail_start are defined in terms of Game, we can use the information just gained about Game to get a specimen of each of these constructs: the productions tell us that **heads stop** is a specimen of the first, and **tails stop** of the second. Now we bring this information back into the production for Game, which has these two constructs among its choices, to get two new specimens of Game:

- **heads stop**
- **tails stop**

Applying the same process again, by using these results in the productions for Head_start and Tail_start then coming back to Game, gives us four more specimens:

- **heads heads stop**
- **heads tails stop**
- **tails heads stop**
- **tails tails stop**

And so on. We see the pattern: a specimen of Game is any sequence of zero or more occurrences of **heads** and **tails** (arbitrarily intermixed), followed by **stop**. More precisely: from what we have seen it is easy to *prove* that any such sequence is a specimen; a slightly more delicate question is whether, conversely, these are the only possible specimens.

→ *Exercise 11-E.4, page 320.*

The very simple language defined by this grammar for the top construct Game might represent all the possible coin-tossing sequences by a player who gives up at some point, crying "**stop**!". The non-recursive grammar

$$\begin{array}{rcl} \text{Game1} & \triangleq & \text{Throw\_sequence } \mathbf{stop} \\ \text{Throw\_sequence} & \triangleq & \{\text{Throw} \ \ldots\}^{+} \\ \text{Throw} & \triangleq & \mathbf{heads} \mid \mathbf{tails} \end{array}$$

generates the same language through three productions, one each of Concatenation, Repetition (with empty separator) and Choice.

This example illustrates the earlier remark that it makes no sense to talk of *the* grammar for a language, since any non-trivial language, even one as simple as this example, can be generated by many possible grammars. The other way around, of course, a grammar defines just one language.

← *"BNF basics", page 297.*

In applying the production rules to generate the language, we have used a strategy that favored terminals (here delimiters **stop**, **heads** and **tails**, but other terminals would play the same role) over nonterminals. By choosing a different policy, we would get into an infinite cycle without ever producing a phrase: start by choosing the first possibility, Head_start for Game; for Head_start, we get **heads** Game; for Game, choose again Head_start; and so on forever. This produces an infinitely growing sequence of **heads** keywords without ever generating a phrase.

To avoid such situations, a language generation process needs strategies, or *heuristics*; a possible one is, for a Choice, always to start (at step P4) with a production using terminals only, if there is one, and otherwise with a production that starts with a token.

→ On the notion of heuristics see also "Interpretation vs compilation", page 542.

Even with such heuristics the process will not yield anything if a grammar is *entirely* recursive. To get the process started it needs at least some choices involving tokens. Grammars such as

$$\text{A} \triangleq \text{A}$$

or

$$\begin{aligned}\text{A} &\triangleq \text{B}\\ \text{B} &\triangleq \text{A}\end{aligned}$$

are useless. These issues will be discussed further in the chapter on recursion.

→ Chapter 14.

A more delicate case is a grammar that does have tokens but is *left-recursive*, as in

$$\begin{aligned}\text{Instruction} &\triangleq \text{Compound} \mid \text{Assignment}\\ \text{Compound} &\triangleq \text{Instruction}\ \text{"}\mathbf{;}\text{"}\ \text{Instruction}\end{aligned}$$

where for simplicity we take Assignment as a terminal (meaning we assume it is defined somewhere else, with no reference to the other nonterminals listed). This grammar is meaningful, since it permits instructions of the form

- *assignment_1*
- *assignment_1* **;** *assignment_2*

and so on. To obtain a constructive view of such recursive definitions, we need a general theory of recursive definitions, sketched later.

→ "Making sense of recursion", 14.7, page 473.

One form of grammar that avoids these complications and makes it possible to write simple parsers is known as **LL (1)**, characterized by the property that the first terminal starting a phrase is enough to choose between variants of any nonterminal. Eiffel is close to LL (1): restricting ourselves to instructions, we know that we have a Conditional if the first token is **if**, a Loop if it is **from** and so on.

11.4 DESCRIBING ABSTRACT SYNTAX

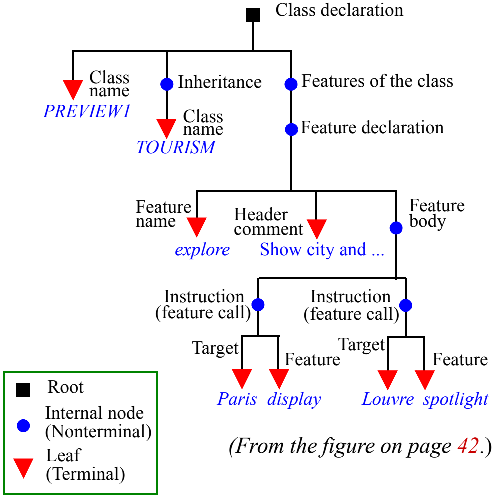
The syntax that we have studied so far in this chapter is the **concrete** syntax of a program: it describes the full structure of program texts, including keywords — **if**, **do**, **class** … — and other delimiters that serve a purely syntactic role: they avoid syntactic ambiguity but do not carry any semantics of their own.

Earlier we had encountered **abstract** syntax, which discards these elements and retains the structurally meaningful ones only.

← *“Abstract syntax trees”, page 41.*

We saw how to describe the resulting syntactic structures through **abstract syntax trees** (ASTs for short), such as the one representing our example *PREVIEW1* class, reproduced on the right.

As noted then, it is easy to deduce a notion of *concrete* syntax tree, which returns all the symbols of the input text. Some compilers indeed construct such a tree, but this is usually not necessary: in subsequent phases of compiling, such as semantic analysis, code generation and optimization, the syntactic markers do not play any role; all we need is a representation of the program’s structure, exactly what an AST provides.



(From the figure on page 42.)

If our goal is to describe the abstract syntax of a language, without going through concrete syntax, we do not need a new formalism. It suffices to use BNF, omitting all tokens that are not constructs of the lexical grammar, in particular keywords. For the last production on the preceding page, specifying Compound, the right side would now be just Instruction Compound.

Such a grammar is not useful for applications such as parsing and compiling input texts — which obviously require a concrete grammar similar to those discussed so far — but can help capture the structural properties of texts, unaffected by details of their textual appearance.

11.5 TURNING A GRAMMAR INTO A PARSER

One of the applications of BNF is, as noted, to guide the construction of compilers, starting with the *parsing* phase. Compilers are usually **syntax-driven**: the parser constructs an AST, and subsequent compilation phases continue to work on this data structure, repeatedly adding semantic information (this is known as *decorating* the tree).

A detailed discussion of how to build such a syntax-driven compiler, or just a parser, is beyond our scope here, but to get an idea of possible techniques you may wish to look at the EiffelParse library. EiffelParse is not the most efficient parsing mechanism available (for a widely used Eiffel parsing tool based on more traditional techniques, look up the "Gobo" library), but it provides a clear, practical illustration of how to apply the object-oriented principles of this book in full to parsing and compiling.

docs.eiffel.com/book/solutions/eiffel-parse-tutorial

The idea behind EiffelParse is to turn a BNF-E grammar directly into a set of classes. For each construct of the grammar, you will write a small class that inherits from one of the EiffelParse classes *AGGREGATE* (for concatenation productions), *CHOICE* and *REPETITION*. In the *AGGREGATE* case, for example, the class will simply list the various components of the production's right side, each associated with a class similarly describing a construct. You have to be a bit careful about left recursion, but otherwise the classes are a mirror image of the BNF-E productions; a translator, YOOC, developed by Christine Mingins, can produce the classes directly from the grammar.

To parse an input text, it then suffices to call the EiffelParse procedure *parse* on the construct of interest. This produces an abstract syntax tree. You can then add *semantic* processing of any kind through procedures of the syntax classes. The process shows the power and elegance of object-oriented modeling for language processing.

11.6 THE LEXICAL LEVEL AND REGULAR AUTOMATA

For terminal constructs such as Identifier and Integer, the BNF grammar does not provide a production, leaving the specification instead to the lexical level. For that reason the terminal constructs are also called **lexical constructs**. Their specification appears in a "lexical grammar" complementing the BNF grammar.

Lexical constructs in BNF

At the syntax level, covered by BNF, tokens (terminals and delimiters) are atoms with no further structure of interest. At the lexical level, we become interested in their internal makeup. For example (using Eiffel conventions):

- An Identifier is a sequence of one or more characters, of which the first is a letter (upper or lower case) and any subsequent one is a letter, digit or underscore "_".
- An Integer is a sequence of one or more decimal digits (0 to 9), which may also contain underscore characters to separate groups of digits for readability in long numbers: 123_456_789.
- An Integer_constant is an Integer optionally preceded by a sign, + or –.

It is in fact possible — as an exercise invites you to check for yourself — to specify such terminal constructs in BNF. That is not, however, the usual practice. For such simple constructs, language definitions generally take advantage of specific *lexical* techniques, which we will now study. This avoids loading the grammar with productions for basic structures that can be described more simply, and reserves the BNF grammar for specifying the higher-level structures of the language, in particular those permitting nesting.

→ "BNF for lexical grammars", 11-E.3, page 319.

Correspondingly, compilers do not use the parser to decode specimens of terminals, but a simpler tool known as a *lexical analyzer*.

Regular grammars

To define the structure of lexical constructs such as the examples above, we may use a **regular grammar**, a toned-down version of BNF.

The nonterminals of such a grammar are constructs such as Identifier and Integer_constant, which will be used as terminals in the BNF. To express their structure the regular grammar has its own more elementary terminals, usually character categories such as

```
       Letter ≜ 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
                'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' |'z'
Decimal_digit ≜ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
   Underscore ≜ '_'
```

Letter and Decimal_digit have a simpler definition, see below.

each expressed as a Choice between **single characters**, shown in single quotes. Such constructs are *really* terminal (atomic): we cannot decompose them any further.

It is common to provide a special notation for consecutive characters, so that we may rewrite the production for Letter as

```
       Letter ≜ 'a' ..'z' | 'A' ..'Z'
```

using this opportunity to add the upper-case variant. We may similarly define Decimal_digit as '0'..'9'

A regular grammar may have the same kinds of production as in BNF, but with slightly different conventions and significant restrictions:

- You may use a **Choice** as just seen, possibly with character intervals.

- You may define a lexical construct by **Concatenation**, but this does not assume breaks (spaces, new lines etc.) between the concatenated elements. If you define a construct as A B, any of its specimens will be made of a specimen of A followed by a specimen of B with nothing in-between. If the language needs a notion of break you may define it explicitly in the regular grammar as one of the lexical constructs.

- A **Repetition** will take a simpler form: just A* or A^+ where A is a previously defined construct. These two forms denote "zero or more specimens of A" and "one or more specimens of A", with no notion of delimiter and again no intervening break.

- **No recursion** is permitted in the grammar, either directly (the definition of A uses A) or indirectly (the definition of A uses B while the definition of B uses A, or a similar scheme with any number of intermediaries). A simple way to enforce this prohibition is to make the *order* of the rules significant — in BNF it is not — and add the rule that the definition of a construct may only refer to constructs whose definitions appear *before* it.

Unlike BNF-E, a regular grammar allows you to mix the different kinds of production (since the formalism is much more restricted). Parenthesizing removes any ambiguity, as in Letter (Letter | Digit | Underscore)* which indicates the concatenation of Letter and a repetition, itself based on a choice.

← *"Touch of Methodology: BNF-E rule", page 304.*

With such a regular grammar we can give a precise definition of the lexical notions of identifier and integer constant:

Identifier	≜	Letter (Letter | Digit | Underscore)*
Integer_constant	≜	Decimal_digit^+

The expressions permitted by the rules just defined are called **regular expressions**. A language that can be described by a construct of a regular grammar is a **regular language**. We may note the following property:

Theorem: Canonical form of a regular language

Any regular language can be described by a regular grammar whose production right sides do not include any nonterminal.

Proof: this is a simple consequence of the prohibition of recursive definitions. Starting from a regular grammar, order the productions as discussed above, so that any nonterminal appearing on the right side of any production has been defined by a previous production. Then for every production in that order, if the right side has a nonterminal *N*, replace it by the right side for *N*. Since the same process has already been applied to *N*, you get terminals only.

For example with

$$
\begin{array}{lll}
A & \triangleq & T1 \mid T2 \mid T3^* \\
B & \triangleq & T4^+ \mid A \\
C & \triangleq & A\ B
\end{array}
$$

this process yields the alternative — not necessarily clearer — definition

$$
\begin{array}{llll}
A & \triangleq & T1 \mid T2 \mid T3^* & \text{-- No change} \\
B & \triangleq & T4^+ \mid T1 \mid T2 \mid T3^* & \text{-- Obtained by replacing } A \\
C & \triangleq & (T1 \mid T2 \mid T3^*)\ (T4^+ \mid T1 \mid T2 \mid T3^*) &
\end{array}
$$

which generates the same language. Another way of stating the theorem is that any regular language can be described by a *single* regular expression; in the example it is the right side for C in the last production.

→ Exercise "Single-production regular grammar", 11-E.5, page 320.

The theorem illuminates the principal restriction of regular languages: they do not support recursive nesting. We saw that programming languages such as Eiffel have Conditional instructions that may contain other instructions of the same kind, or of different kinds such as Loop which in turn can contain conditionals, allowing nesting up to any desired depth. With BNF we can describe this through recursively defined productions (keeping the description finite); with regular grammars we cannot.

Regular grammars are well suited, however, for defining the usually simple tokens that make up the elementary fabric of programs. To express that a certain kind of token has specimens made up (say) of one or more character of a certain kind, followed by any of three specified characters, followed optionally by characters of another kind, regular expressions are just the ticket.

Finite automata

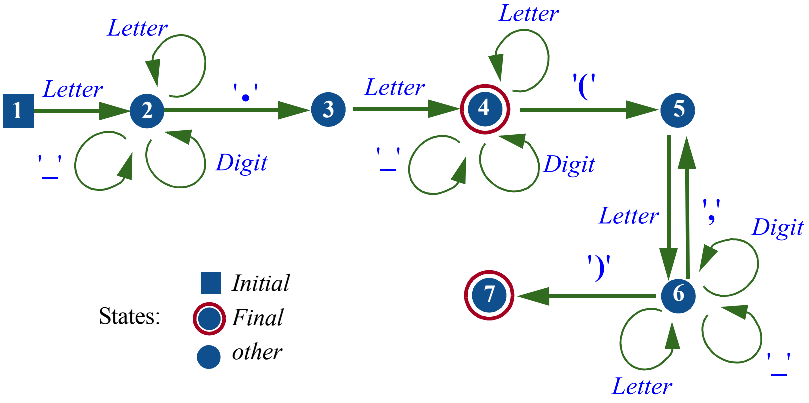
Behind regular expressions stands the mathematical theory of *finite automata*. Let us take a glimpse at it, if only for the visual illustration that it provides. (There is much more to the theory than appears in this brief introduction.)

One "automaton", two or more "automata" (this was our English lesson for today).

A finite automaton is a graph with nodes representing states and edges labeled by elements of some basic set, here characters or character categories.

The following example corresponds to the structure of qualified feature calls in Eiffel, possibly with arguments, as in *Line8.extend* (*new_station*):

Without spaces, for simplicity.



Finite automaton for recognizing feature calls

As the name suggests, we can view a finite automaton as a machine to process input strings. The automaton will start from the state marked *initial* and then, at each step, follow an edge, if any, labeled by the next symbol from the input. This is called a **transition**. For the input *x9.f_g*(*a*,*a*), the above automaton starts in state **1**; the input symbol *x* causes a transition to state **2**; then *9* causes a transition from state **2** to itself; the period takes us to state **3**, then *f* to **4**, where it stays for _ and *g*; the argument list then causes successive transitions to **5**, **6**, **5** again for the comma, **6** again for the second *a*, and **7**, marked as a *final* state.

The **language recognized by a finite automaton** is the set of strings which — like *Line8.extend* (*new_station*) and *x9.f_g*(*a*,*a*) in this example — will take the automaton, starting from the initial state, through a sequence of transitions ending in a final state. A string does *not* belong to that language if, in an attempt to apply this process to the string:

- Either the automaton reaches a state that has no transition matching the next input symbol (as with the input string *a.b.c*, where *a.b* takes the automaton to state **4**, from which there is no edge labeled with a period).

Actually legal Eiffel, but not included in this mini-language of feature calls where we only accept single-dot calls.

- Or, having consumed all symbols from the input, it reaches a non-final state (as with the input string *a.b* (, taking us to state **5** which is not final).

A basic theorem states that any language described by a regular grammar is recognized by a finite automaton, and conversely. Without proving the theorem, we may illustrate it by noting that the language recognized by the above automaton is also the language generated by the last construct of the following regular grammar:

```
       Identifier ≜ Letter (Letter | Digit | Underscore)*
 Another_argument ≜ "," Identifier
    Argument_list ≜ "(" Identifier Another_argument* ")"
     Feature_call ≜ Identifier "." Identifier [Argument_list]
```

The feature calls recognized by this grammar are only a subset of those possible in Eiffel, where expressions, like instructions, can be nested: a feature argument can be an expression, as in $x.f\ (y.h\ (z.i))$, which the above lexical grammar and the associated finite automaton cannot handle since they limit any argument to a single identifier. As soon as you venture beyond tokens, you need the full power of BNF. (Note also how the BNF-E convention for repetitions, with its provision for a separator, is more convenient: to define the equivalent of the above Argument_list it needs a single production, with the right side {Identifier "," ...}$^{+}$.)

> The above automaton is *deterministic* in the following sense: it has a single initial state, and from any state there is at most one edge for any given character; as a result, the recognition process illustrated above can take, at any step, at most one transition. *Nondeterministic* finite automata are not subject to these constraints. It turns out, however, that they recognize the same class of languages.

Finite automata provide the basis for *lexical analyzers*, the part of compilers that takes care of recognizing tokens. It is indeed not hard to see how to define a finite automaton from a regular expression and, from that definition, build a program that will recognize tokens through the process just illustrated. This is the scheme behind all lexical analyzers.

Context-free properties

The theory of formal languages distinguishes a number of levels including, from simplest to most sophisticated:

- **Regular languages**, which can be described by a regular grammar.
- **Context-free languages**, which can be described by a grammar made of production rules with possible recursion, as in BNF.
- **Context-sensitive languages**, for which such production rules are no longer sufficient.

As an example of why context-free languages are not enough, consider the *type rules* that govern many programming languages. Eiffel requires that whenever you use an entity *x*, in an expression such as *some_function* (*x*) or an instruction such as *x*.*some_procedure*, there must exist, in an enclosing program unit — the enclosing routine, or the enclosing class — a declaration of the form

```
x: SOME_TYPE
```

specifying *x* as a formal argument or local variable of the routine, or a query of the class; *SOME_TYPE* must be a type suitable for an argument to *some_function* or for the target of *some_procedure*. Otherwise your program is invalid and compilers must reject it. But this is different from an error such as

```
if c then a + b end
```

which violates the BNF grammar (since the relevant production specifies an Instruction after **then**, whereas *a* + *b* is an Expression).

There are many more examples of construct specimens that satisfy the grammar but are not acceptable, such as an assignment *a* := *b* where the type of *b* does not conform to the type of *a*.

Context-free grammars and BNF cannot capture such type rules; a grammar to handle them would have to be context-sensitive. Where BNF rules define a nonterminal A as a sequence γ of terminals and nonterminals, a context-sensitive grammar has rules allowing replacement of α A β by $\alpha\ \gamma\ \beta$; here α and β are the "context" around A.

In practice, no context-sensitive grammar formalism exists that matches the simplicity and practicality of BNF. Because programming languages need type rules and other context-sensitive properties, what compiler writers do in practice is to:

- Rely on a regular grammar to describe the language's lexical properties and build a lexical analyzer.
- Rely on BNF or equivalent to handle the context-free aspects of the language — the overall, usually nested, structure of programs — and on associated techniques for parsing that structure.
- Enforce all other checks — the context-sensitive aspects, such as type rules — through additional mechanisms, either based on formalisms for the description of context-sensitive aspects (such as "attribute grammars") or programmed in an *ad hoc* way in the compiler.

Touch of History: Classes of languages and grammars

The classification of languages into regular (*Type 3*), context-free (*Type 2*), context-sensitive (*Type 1*) and unrestricted (*Type 0*, recognizable by any general automaton or “Turing machine”), comes from articles published in 1956 and 1959 by Noam Chomsky, then as now a professor at MIT, and Marco Schützenberger from the University of Paris. Chomsky, also famous as a political activist, was interested in the structures of *human* languages, for which his work started a whole new school of linguistics; but it proved seminal as well for the understanding of programming languages and other artificial notations.

Chomsky (2005)

11.7 FURTHER READING

Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers*: *Principles*, *Techniques and Tools*, Addison-Wesley, 2006.

The newest edition of a famous compiler textbook, going back several decades and still a standard reference.

Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen: *Modern Compiler Design*; Wiley, 2000.

A good description of current compiler technology.

Steven S. Muchnick: *Advanced Compiler Design and Implementation*; Morgan Kaufmann, 1997.

Another recent text, up to date on many important compiler techniques.

Sethi (2008)

11.8 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A formal language, such as a programming language, is a set of *phrases* built from a basic vocabulary according to precise rules.
- Most interesting formal languages are infinite.
- BNF is a formalism for describing a formal language from a finite set of rules called “productions”.
- Each production of a BNF grammar describes the structure of a certain construct, or “nonterminal”, from other nonterminals as well as atomic constructs or “terminals”.
- A production defines a construct by one of: concatenation of other constructs, possibly with optional components; choice between other constructs; or repetition of another construct.

Lam (2008)

- Compilers and other language analysis tools use grammars for decoding, or “parsing”, the structure of input texts.
- BNF can also describe abstract syntax which (unlike “concrete” syntax) discards keywords and other elements that do not directly carry a semantic value of their own.
- For the elementary components of input texts, such as identifiers and constants, BNF is usually overkill; simpler descriptions can be obtained through *regular* grammars, where productions may not be recursive and as a result do not support nesting. Regular expressions are closely associated with mathematical devices known as finite automata.
- BNF covers the class of “context-free” languages but does not capture “context-sensitive” aspects such as type rules.

New vocabulary

(Also remember, from the presentation of basic syntax concepts: Abstract syntax tree (AST), Construct, Grammar, Lexical, Nonterminal, Specimen, Syntax, Terminal.)

← *3.4, page 39 and 3.6, page 41.*

BNF
Choice production
Concatenation production
Defining production
Grammar
Lexical construct
Lexical grammar
Metalanguage
Phrase
Production
Recursive grammar
Repetition production
Top construct
Vocabulary

11-E Exercises

11-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

11-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise “Concept map”, 10-E.2, page 293.*

11-E.3 BNF for lexical grammars

Write a BNF grammar that fully describes the Eiffel forms of Identifier, Integer and Integer_constant.

← *“Lexical constructs in BNF”, page 311.*

11-E.4 Language defined by a recursive grammar

Consider the language defined by the grammar with top construct Game:

← *In "Recursive grammars", page 307.*

1 Prove that any specimen of Game1 in the non-recursive grammar, that is to say any sequence of one or more **heads** or **tails** followed by a single **stop**, is a specimen of Game.

2 Conversely, is any specimen of Game a specimen of Game1? Prove your answer.

11-E.5 Single-production regular grammar

Devise a single regular expression that describes the entire language generated by the Game construct.

← *In "Recursive grammars", page 307.*

12

Programming languages and tools

Over the past four decades software tools have profoundly changed how people from all industries design their products, from cars to pharmaceutical drugs, newspapers, bridges and buildings — the list goes on. This is known as Computer-Aided Design (CAD, complemented by CAM, Computer-Aided Manufacturing). Software construction is design too; disproving the old saying about the cobbler's children, software engineers have developed CAD tools for themselves, from fairly simple programs such as text editors to entire tool suites known as integrated development environments or IDEs.

Software engineering tools are an integral part of the professional culture of any software engineer, who should know what kinds of tools are available to support the software development process and what they can and cannot do. This chapter gives an overview of the most important concepts.

We start with conceptual tools, programming languages, and continue with software tools — computer programs that help build other programs.

Programming languages are the programmer's primary means of expression. They have a long history, and a number of distinctive programming language styles have emerged. We will discover a bit of the history of the style used in this book, object-oriented programming, and the basic concepts of another, functional programming.

Software tools offer a rich set of possibilities, starting with the tools most directly related to programming languages: compilers, a familiar figure from previous chapters, but also their sister tools, interpreters; we will particularly examine how compilation and interpretation complement each other. Other tool categories reviewed in this chapter are: program preparation tools such as editor and graphical CASE (Computer-Aided Software Engineering) tools; debuggers, static analyzers and testing tools, which allow us to assess and improve program reliability; configuration management tools, which help us keep track of the many components of a software system; browsing and documentation tools, which help us understand software at various levels of abstraction and navigate through its complexity; metric tools, which give us a quantitative views of a software system. We end with Integrated Development Environments, which collect many tools to present the developer with a unified framework; the last section sketches one IDE, EiffelStudio, and its approach to compilation (Melting Ice Technology).

The discussion in this chapter is more general and less detailed than in its predecessors. Compilation and interpretation, in particular, are highly technical topics deserving courses and textbooks of their own; here you will only find a survey. Take this chapter as a leisurely walk, a warm-up for the next climbs starting with recursion in the next chapter.

12.1 PROGRAMMING LANGUAGE STYLES

Programming languages play a central role in software development. The core of this book uses a programming language, Eiffel, but downplays its role, since it focuses on programming concepts rather than the specifics of a particular language. Appendices describe the specifics of four important languages: Java, C#, C++ and, briefly, C. Here we will review some general criteria to classify programming languages, learn a little more about the object-oriented family of languages (which includes all the languages cited above except C), and get a glimpse of another family with its own unmistakable traits.

Appendices A to D.

Classification criteria

Programming languages can be classified along several criteria. Here are the most important:

- Application. Some programming languages are *general-purpose*; others address a specific application area, such as Web site development, business data processing or real-time. The term *domain-specific language* or DSL refers to this second category.

  Such a characterization by application area is always subject to revision, since successful languages frequently outgrow their original targets. Two examples are Fortran, the first widely used programming language, which was intended for mathematical computation (“FORmula TRANslation”) but became a general-purpose language; and Java, devised for programming set-top boxes, then presented as a tool for writing Internet “applets” and soon thereafter generalized to many more areas.

Appendix A.

- Program scope. Some languages are intended for developments that may become large-scale (large code size, many developers, development and operation over many years); others target smaller developments and prototypes that may be discarded after fulfilling an immediate need or testing a conjecture. *Scripting languages* are generally of the latter kind. There is of course no guarantee that a program that starts small will remain small, and as a consequence successful languages of the second category often end up being applied to large developments.
- Verifiability. Some languages are designed to ensure that compilers and other tools (as discussed in the rest of this chapter) can find out potential flaws — or, inversely, guarantee specific properties of the execution — through program analysis, *before* any execution; usually this puts a higher burden on the programmer, since verification may require adding information (for example, extensive type declarations) to the program. Other languages favor ease of expression and lighten the requirements on programmers, with the consequence that errors may not become visible until run time.
- Abstraction level. Some languages rely on direct use of machine-level concepts; others provide a more abstract model of computation.
- Lifecycle role. Some languages address implementation only; others can also help, beyond programming in a restricted sense, for system modeling, analysis and design.
- Imperativeness vs descriptiveness. In *imperative* languages, programs are made of commands that modify the program state. Other languages are *descriptive* in the spirit of mathematics, leading to programs that specify properties of the intended results but not the precise sequence of steps for obtaining these results.

→ Some descriptive languages are "applicative". See the more precise definition of "imperative" on the next page.

- Architectural style. This defines the main criterion for decomposing systems into modules. The two main possibilities are to organize the modules around units of the software's *functions*, or around the *types of objects* it manipulates. The corresponding language styles are called *procedural* ("functional language" means something else, as we will see) and *object-oriented*.

Almost every combination of these various choices is possible (and many are represented in practice). The style of the programs in this book, illustrated by Eiffel but also broadly representative of Java and C#, is: general-purpose (although specializable to application areas through libraries); suitable for large developments; designed for thorough compile-time verification; at a high level of abstraction (while providing access to machine-level mechanisms, particularly through libraries); wide lifecycle role (this is particularly true of Eiffel, which is routinely used for specification and design); imperative; object-oriented.

→ See appendices A about Java and B about C#.

The widely used C language, as another example, is imperative but not object-oriented; it remains at a rather low level of abstraction to ensure direct, almost machine-level execution control. C++ adds an object-oriented layer to C.

→ Appendix C describes the C++ language , and appendix D its C subset.

The rest of this programming language discussion covers two important complements to the rest of this book:

- An introduction to a style of programming languages belonging to the "descriptive" category: *functional* languages, which radically depart from today's dominant practices.
- Some background on object-oriented programming languages.

Functional programming and functional languages

An earlier chapter — please read that short discussion again! — emphasized a fundamental distinction between the concepts of programming and mathematics:

← "Math is static, software is dynamic", page 227.

- Programs work on a *state* (think of it as a more abstract notion of computer memory) which they repeatedly change through such instructions as assignment, which updates the value of a variable in a state, and more generally through commands. State changes are also known as *side effects*.
- Mathematical discourse is purely descriptive; it does not change anything, but talks about values and relations between these values.

Not everyone is happy with this distinction. The programming style known as *functional programming* seeks to bring programming as close as possible to mathematical reasoning, often by rejecting the notion of state; the basic concept is that of a function, in the mathematical sense of a mechanism that defines how to obtain results from arguments without any notion of side effect. Expected benefits include making programs simpler and being able to reason clearly and rigorously about them through standard mathematical techniques.

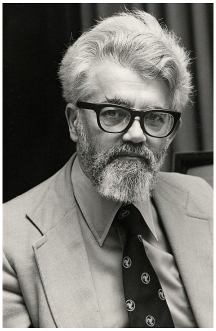
The following terminology is useful:

Definitions: imperative, applicative

A programming style that relies on state changes is *imperative*. A synonym for non-imperative is *applicative*.

Most programming languages, including Eiffel, support an imperative style (tempered, in the Eiffel case, by a strict distinction between commands and queries, where queries are expected to be applicative). Functional languages are essentially applicative; "*essentially*" because many of them provide a few imperative escape routes for operations that are by nature imperative, such as database manipulation and I/O.

The first functional language (retaining some imperative characteristics) was Lisp, a breakthrough development by John McCarthy, introduced as early as 1959.



McCarthy

The central concept of Lisp is the list, written in parentheses, as in

```
(A B C)
```

Note that Lisp uses the term "list" in a special sense, different from the usual computer science concept of list as used in other chapters. Lisp's lists are in fact closer to the *binary trees* which we will study in the data structure chapter.

Lists are data structures (in fact the only ones in Lisp, general enough to cover a wide variety of applications); but they also serve as program structures: if A denotes a function, the example list above denotes the application of that function to the argument list given by the rest of the elements, what in more ordinary notation would be written A (B, C). The power, simplicity and elegance of the ideas seduced many people, and much of the early work in Artificial Intelligence was carried out in Lisp. The language continues to be actively used and developed, in particular through one of its descendants, Scheme, which retains the essential concepts.

Even without any further detail of Lisp mechanisms, the basic idea immediately suggests a possibility that all functional languages offer: the ability to use *higher-level functionals* — functions that take functions as argument or return functions as a result. Consider

```
((F A B) (G C) D)
```

This is a list of three elements, of which the first is itself a list with three elements, and the second a list of two elements. F may be a function of two arguments, which returns as its result a function of two arguments; call H the function that results from applying F to arguments A and B. If G is a function of one argument, call E the result of applying G to C. Then the expression as a whole denotes (H E D), the result of applying function H to arguments E and D. This is a remarkably powerful and general mechanism.

In a later chapter we will see a mechanism, *agents*, that achieves similar aims in an object-oriented framework, and the basic ideas of the underlying theory, *lambda calculus*, which also serves as the theoretical foundation for Lisp and much of functional programming.

→ Chapter 17 introduces agents and lambda calculus.

To get a closer look at the functional programming style let us move from Lisp to a more recent design, Haskell, which has become one of the most popular functional languages. We will only consider one example; it illustrates functional programming ideas well and is also an excellent preparation for our forthcoming study of *recursive* reasoning.

→ Chapter 14 covers recursion.

Remember the algorithm to reverse a linked structure. It involved delicate pointer manipulations. Isn't there a way to obtain the result — after all, the notion of reversal is conceptually simple — without all these details? In functional programming there is, if you are willing to ignore performance issues (or trust that a very smart compiler will solve them from you).

← *"Reversing a linked structure", page 259..*

A list in Haskell is written in square brackets [...]. The following definition specifies a function that reverses a list:

```
reverse :: [T] –> [T]
reversed [ ]                  = [ ]                                      [1]
reversed (first:remainder)    = reversed remainder   ++   [first]        [2]
```

This is all you need to write. The first line is a type declaration, stating that reverse is a function that for any list of values of type T (written [T]) will return a list of elements of the same type; this applies to any type T. The definition of the function, in the next two lines, is by cases:

→ *Such a definition applying to an arbitrary type is known as* **generic**. *We will study this concept in "Static typing and genericity", 13.1, page 363.*

- Case [1] covers the case of the empty list [ ]: the result is also empty.
- Case [2] covers the case of a non-empty list. Such a list can always be written first:remainder, a notation that represents a list starting with a first element first (which must exist since the list is not empty) and continuing with a list remainder containing the remaining elements; remainder could be empty. In this case the result is the reverse of remainder, obtained by applying the function itself (this is a case of *recursive call* as studied in detail in the corresponding chapter) and appending the element first to it, through the operator ++. Note that function application in Haskell is written without parentheses, so that reversed remainder means the application of the function reversed to the argument remainder; since function application binds more tightly than the ++ operator, the right side of [2] would be written (reversed (remainder)) ++ [first] in ordinary mathematical notation.

Simon Peyton Jones

Phil Wadler

(Two of the principal designers of Haskell; 2008 photographs.)

In contrast with the imperative style, such definitions are descriptive; they specify the properties of the result rather than a precise sequence of computational steps that produce it. The above definition is in fact close to how you might explain the notion of "reverse of a list", without necessarily thinking of a computer implementation: if the list is empty, it is its own reverse; if not, you may obtain reverse as the reverse of its tail part (the list without its first element) followed by its first element.

The definition is applicative, making no reference to state changes and indeed using no notion of state in the sense of a set of variables to which the program may assign new values.

Although (as you may guess) there is much more to functional programming in Haskell, Lisp and other functional languages, this small example conveys the simplicity and elegance of functional programming concepts.

Why then hasn't everyone switched to functional programming? This is a controversial issue — proponents of functional programming think the world *should* switch — but we may note three significant problems:

- Performance. The simplicity of solutions such as the one above may result in considerable time or space overheads. It is instructive to note that documentation of functional programming languages typically starts with examples in the above style and then recommends using more complicated variants to achieve better efficiency.
- Scalability. For structuring large systems, the notion of class present in object-oriented languages is more effective than the notion of function. It should be noted, however, that many functional languages have added some object-oriented constructs.
- Statelessness. While a purely applicative language may considerably simplify algorithms of the kind illustrated above, working on possibly complex data structures, some aspects of computing fundamentally require the notion of state; think for example of input and output (which rely on changes in the program's environment), or real-time systems. Functional languages, Haskell in particular, have added special mechanisms to handle such stateful (imperative) aspects of programming, but they are not as simple as the basic functional model.

In the eyes of many software development practitioners, imperative object technology — used in this book and implemented with some variants by many of today's most popular programming languages — provides a better answer to the critical challenges of software development. But not everyone is of this opinion, and in any case it is important to understand the concepts and applications of functional programming.

> It is possible to emulate part of the functional programming style in an imperative object-oriented language by using recursive functions and avoiding side effects. An exercise in the recursion chapter asks you to write the equivalent of the Haskell reversed function using plain imperative O-O mechanisms.

Object-oriented languages

Before we leave the topic of programming languages, let us take a few moments to understand the background of the language style that underlies this book and has also, over the past two decades, become dominant in the software industry (at least the part of the industry that is not just concerned about maintaining older, "legacy" code): object-oriented languages.

It is easy to trace the precise origin and originators of the technology: the place was Oslo (Norway), the time was the early- and mid-sixties, and the creators were two computer scientists, Ole-Johan Dahl and Kristen Nygaard, from the University of Oslo and the more industry-oriented Norwegian Computing Center.

Nygaard

Together they designed a language for *discrete-event simulation*: applications that use the computer to model processes of interest, such as the running of a factory, not through mathematical analysis but by executing sequences of computer operations that emulate the actual events. The first version, Simula 1, was specifically tailored to this application area. The next one, Simula 67, the topic of their 1967 paper in *Communications of the ACM*, was a general-purpose language, and introduced the key object-oriented ideas. It is indeed amazing, more than 40 years later, to see how much was there: classes, objects, dynamic allocation with garbage collection, inheritance (single only), polymorphism, dynamic binding.

Dahl

The mainstream academic community remained strangely uninterested. Only a few years later would some theory become available to understand object technology: information hiding (Parnas's 1972 work) and *abstract data types*, presented in a short 1974 paper by Barbara Liskov and Stephen Zilles, then given a firm mathematical basis by John Guttag's 1975 PhD thesis.

Liskov (2007)

(with Don Knuth)

Simula's origins as a simulation language are telling. At the heart of the O-O method lies the idea that programs are not just for talking to your computer; they are modeling tools. In his talks, Nygaard often used the slogan "*To program is to understand*" and was proud that the first ever Simula report was entitled "A Language for Programming *and Modeling*". Discrete-event simulation, with its emphasis on describing external processes, is the ideal target area to develop such a view; but the evolution from the specialized Simula 1 to the general-purpose Simula 67 showed the general applicability of that view. Much of programming, aside from strictly technical implementation issues, concerns itself with understanding systems of various kinds — a company's accounting process, a digital camera, a document preparation system… — and devising suitable models of them. O-O concepts are successful because they effectively support this modeling process: we describe the system through object types (*ACCOUNT*, *IMAGE*, *PARAGRAPH*), organize them in inheritance hierarchies (an *INTERNAL_ACCOUNT* is a special case of an *ACCOUNT*), apply information hiding to make sure we can develop them separately, and specify them through contracts. These ideas (except for the last one) were present in Simula; its creators clearly understood the potential.

So did an enthusiastic community of Simula users, the majority of which was in Europe. There was, however, no industrial power behind Simula, and to many people the whole approach seemed exotic. For many years, Simula remained the best-kept secret of the software industry.

Kay (2007)

But the ideas were gaining ground. At the university of Utah, a powerhouse of graphics research, Alan Kay combined ideas from Simula and from Lisp for his PhD thesis. When he moved to the Xerox Palo Alto Research Center in California (PARC), the birthplace of many modern ideas in hardware and software, he produced the first versions of the Smalltalk language and programming environment, starting with Smalltalk-71; the other two key members of the group were Dan Ingalls and Adele Goldberg.

Smalltalk is a dynamically typed language: you will not find in it any of the type declarations used throughout this book and, correspondingly, any of the various protections that such declarations afford when the compiler detects a type inconsistency. In fact, the first versions of Smalltalk did not have a compiler; they were purely interpreted (the distinction is explained in detail later in this chapter). Smalltalk was notable not only for its application of object-oriented ideas at the language level, but also for a wonderful graphical development environment, ahead of anything that existed at the time. What dynamic typing lacked for guaranteeing reliability, it made up for by providing developers with a remarkable degree of freedom in trying things out and even experimenting with the environment itself.

Goldberg (2002)

With successive versions, in particular Smalltalk-76 and Smalltalk-80, interest in the language grew; but the devotees were still, like those of Simula, a small club. Then in 1981 *Byte* magazine, then the battleship of the rapidly growing community of personal computer enthusiasts, decided to publish a special issue on Smalltalk, even though it was not available on the machines its readers typically used. The August 1981 issue (today a collectors' item), edited by Adele Goldberg, opened the floodgates, allowing a new generation to discover object-oriented programming; when in 1986 the ACM organized a conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA, since then a yearly fixture), expecting a hundred attendees, a thousand people showed up. New languages started to appear; both Brad Cox, who had used Smalltalk at ITT (then a large telecommunications company), and Bjarne Stroustrup from AT&T, who had used Simula for his PhD at Cambridge, had the idea of extending C, by then a lingua franca of programming, with object-oriented concepts. The results were Objective-C and C++. The Eiffel language also dates back to that period.

In just a few years object-oriented languages overcame industry's initial suspicions and some of their own limitations (affecting in particular performance) to become significant players at first, and the dominant ones soon thereafter. New languages appeared, notably Java in 1995 and C# four years later.

Ever since the first OOPSLA, critics have been predicting the "end of objects". There has never been any sign of such a phenomenon. Objects continue to thrive; as one observer put it, there is no other game in town.

12.2 COMPILATION VS INTERPRETATION

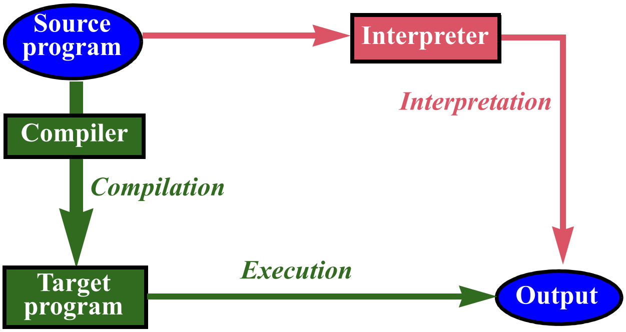
The rest of this chapter explores software tools supporting software development.

We write our programs not in the form that computers can directly execute, machine code, but in a higher-level notation meaningful to humans — a programming language. A programming language can be viewed as machine code too: machine code for a fictitious brand of computer, more abstract than actual processors. We talk of an abstract machine, or virtual machine.

The purpose of compiling is to enable execution by the actual machine of code written for the abstract machine. Compiling is, however, only one of two basic techniques available for this purpose.

Basic schemes

Instead of compiling a program, we may interpret it. The following figure — ignoring the role of input data — illustrates the difference.



Compilation & interpretation (without input data)

Both compilers and interpreters are programs, whose inputs are arbitrary programs written in a programming language. A compiler (see the green path, down then right) translates its input program into a target form which, being in machine code, can directly be executed by the computer, leading to output results. Processing the same program, an interpreter (red path, right then down) does not produce another program but directly computes the program's output.

The interpreter must be able to determine the effect of executing every kind of construct in the programming language. As an example of how interpreters do this, consider interpreting an assignment $x := x + 1$. The interpreter must keep a table of all variables and their associated values. It evaluates the expression $x + 1$ by looking up the current value of x in the table and adding 1 to it. Then it carries out the assignment by replacing the value in the entry for x by the newly computed value.

→ The table will typically be a "hash table", as studied in "Hash tables", 13.9, page 411 .

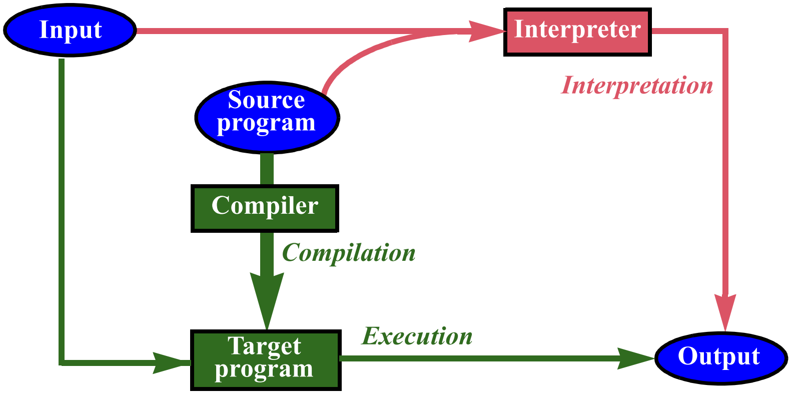
A compiler would generate machine code that produces the same effect, using machine instructions for the operations and memory addresses (rather than a high-level data structure such as a table) for the data.

> Complementary exercises ask you to write both an interpreter and a compiler for a small language, applying these ideas.

→ *16-E.3, page 616 to 16-E.6 (part of a later chapter as they benefit from recursion and inheritance).*

In the view of a programming language as the machine code for an abstract machine, the interpreter is a program that simulates execution of this machine; the machine's memory is abstract too, represented by data structures such as the interpreter's variable-value table.

The full picture of compiling vs interpreting includes input data:



Compilation & Interpretation

This highlights another difference: while the input of a compiler is just a source program, an interpreter needs two sources of input, the program and its data. A later discussion will express this observation mathematically.

→ *"Currying in practice", page 645.*

The basic distinction between compilers and interpreters — whether to process information as it is, or transform it first into a more convenient form — is an important concept of computer science and has applications beyond program processing, as we will see in studying advanced algorithms.

→ *"Interpretation vs compilation", page 542 and "Invest then enjoy", page 694.*

Compilation and interpretation have complementary strengths. Several criteria are involved. On run-time performance compilers win:

- With compilation, the output is machine code, which the computer executes directly. In addition, compilers can perform optimizations to tune the generated code for performance.
- Interpreted code requires further processing by a program, the interpreter, typically slower by one or more orders of magnitude than direct execution by the hardware.

The advantage changes sides if we switch to the perspective of development speed and convenience. The compiler stands between you and the realization of your latest idea: before seeing the result of a change, you must wait for the compiler to compile your program — and also to link it as discussed below. With interpretation you can start executing immediately.

In modern development environments this disadvantage of compilation may not be critical, thanks to *incremental compilation* techniques which, after a change, only recompile the parts of the program that have changed or are directly affected by the change. At the end of this chapter we will see how EiffelStudio applies such techniques.

→ *"The melting ice technology", page 357.*

The last criterion, bringing the advantage back to compilation, is program reliability. Compilers do not just translate source code into another representation; in the process they also perform various validity analyses, such as type checks in a statically typed language. This roots out many errors which, in a purely interpreted solution, would only manifest themselves at run time.

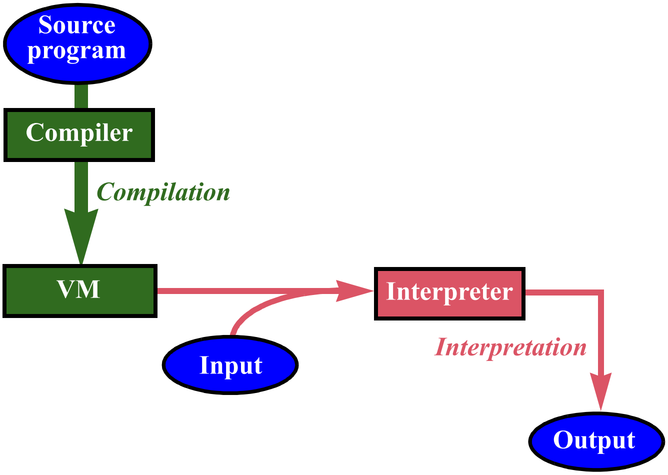
In principle an interpreter can also perform some of these checks before executing the program. It is then no longer a pure interpreter, but already a mix of interpretation and compilation.

Combining compilation and interpretation

The pure compilation and pure interpretation schemes are extremes; many practical solutions use a combination. This is the case with the EiffelStudio compilation process studied later in this chapter.

We may note that a 100%-interpreted scheme makes little sense: every time the interpreter executes an instruction, for example as part of a loop, it would have to go back to its source code — the actual sequence of characters — and parse it again. Any realistic solution avoids such a waste of resources; practical interpreters start instead with a step that turns the input into a form more directly suitable for interpretation, for example an abstract syntax tree. In the process they may also, as just noted, perform some consistency controls such as type checks. Even when you read that a language implementation uses an interpreter this is usually the meaning.

Combining interpretation and compilation goes further than this basic idea. In the general compilation scheme the output does not have to be machine code but may be subject to further processing:



Compilation plus interpretation

This mixed strategy, involving compilation to an intermediate virtual machine — "VM" in the figure —, can be tuned to reconcile the advantages of compilation and interpretation. Through a careful design of the virtual machine it is possible to get:

- Portability, since the VM code can be independent of physical processors.
- A reasonable level of efficiency, if the code is chosen to be low-level enough for fast interpretation.

Virtual machines, bytecode and jitting

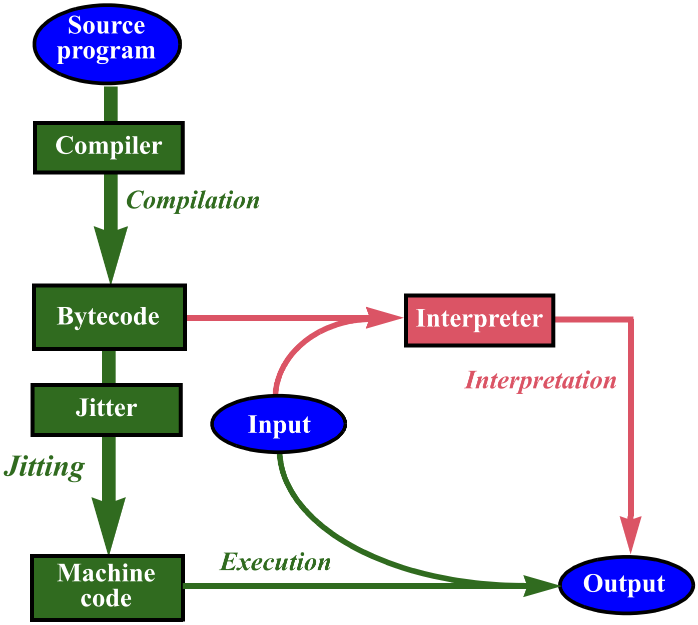
Both Java and .NET implementations commonly rely on a mixed solution as just described, where the intermediate code is known as a bytecode. The term emphasizes that such virtual machines use compact instructions, similar to those of actual processors, where each instruction contains an instruction code — typically fitting in a byte — followed by 0, 1 or 2 arguments.

> An alternative to bytecode would be to use, as the target of the compilation, a set of data structures, for example an abstract syntax tree to represent the program structure and a hash table to store properties of variables. With such a structure it is easier to write both the compiler and the interpreter, but bytecode gives better run-time efficiency, both in space (as the code is tighter than data structures) and in time.

This technique, relying on compilation to a bytecode-based virtual machine then interpretation, was already used by Pascal implementations in the seventies. It came to renewed prominence with the spread of the Internet since it lends itself to distributing code for local execution by Web clients: the provider of a small program, or *applet*, can compile it into bytecode and deliver it in that form; in addition to the compactness of bytecode the portability benefit is particularly interesting here since the alternative would be to generate binaries for every possible target platform. To run the applet, users only need a bytecode interpreter. They do not even have to know that the interpreter exists if it is embodied in their Web browser. Because the approach raises potential security risks — rogue applets could invade your system — it requires the use of trusted interpreters; this is the case with established Web browsers, which strictly limit the scope of the operations applets can perform.

> Program delivery through applets has achieved some success but has not become the primary means of software distribution as some had predicted at the time Java was introduced. Apart from security concerns, the main reason is the loss of efficiency inherent in any solution involving interpretation. Most successful applets are small programs intended to run on a Web page, often with a strong visual component, for which the performance penalty is tolerable.

To improve run-time efficiency without reintroducing the full overhead of compilation, some implementations of the bytecode-virtual-machine scheme add a technique known as Just-In-Time compiling or JIT. The idea is to produce machine code for some modules on the fly, the first time they get called during execution. This is known as "jitting" and the compiler from bytecode to machine code is a "jitter" (I will spare you the "jitter bug" jokes). We can picture it as a refinement of the previous figures; the addition is at the bottom left:



Compilation plus interpretation and jitting

Usually, as the figure suggests, the option remains of interpreting the bytecode rather than jitting it.

Jitting normally takes place, if at all, on the first use of a particular module (a feature, or the entire class); since this will only occur for code that is actually used during execution, the technique often saves space over a traditional compiler which would generate code for the entire program. It also saves some compilation time but, more importantly, spreads compilation overhead over execution (as opposed to executing a traditional compilation step before execution starts). The disadvantage is that execution can be penalized by on-the-fly jitting; in particular, execution time is less predictable.

It seems at first sight unnecessary with this approach to perform type checks and other consistency controls at the jitting stage: who would want to start execution and suddenly, as the corresponding module gets jitted, discover a type error? This would take us back to the problems of dynamically typed languages. Rather, we may feel entitled to expect that the first compilation step has performed all the necessary checks, so that any code passed to the jitter is type-safe.

Unfortunately these reassuring assumptions are unrealistic in a distributed setting, because security concerns pop back up again. If you download bytecode from a site, how do you know it passed the type checks? In general you do not. But then type violations do not just cause reliability violations, such as crashes; attackers may be able to turn them into security violations.

> From a security engineer's viewpoint, a security violation is worse than a crash: with a crash everything stops; with a security violation execution seems to continue, or to terminate normally, but systems may have been compromised or confidential information leaked.

As a consequence, practical jitting solutions perform type checking anyway. The performance penalty can remain reasonable since the type system of a bytecode-based virtual machine is generally simpler than those of high-level programming languages.

The compilation strategy of EiffelStudio also involves a bytecode, but as we shall see below it relies on a different way of combining interpretation and compilation.

→ *"The melting ice technology", page 357.*

12.3 THE ESSENTIALS OF A COMPILER

Today's compilers (and interpreters) are sophisticated tools, resulting from a half-century of research and development. The principal task of a compiler is to generate some target code from a source programming language, but as we have seen it is not the only one: on the way, compilers check program validity.

Compiler tasks

The details of compilers vary considerably, but the general tasks tend to be common to all variants. We look at them in the rough order in which the compiler must apply them when processing a source text.

Lexical analysis turns the text into a sequence of tokens representing identifiers, keywords and symbols. We have seen the basic techniques used for this task: finite automata and regular grammars.

← "The lexical level and regular automata", 11.6, page 311.

Parsing reconstructs the syntax structure of the program.

Validity checking includes type checking and other consistency verifications. Eiffel, for example, has some 90 "validity rules" such as "in an attachment (assignment or argument passing), the type of the source must conform to the type of the target" (a type rule, setting the limits on polymorphism) and "a class B may not list a class A among its parents if B lists an ancestor of A among is own parents" (to prohibit cycles in inheritance).

Actual Eiffel validity constraints are expressed in a more formal "if and only if" style. See the ISO/ECMA Eiffel standard at www.ecma-international.org/publications/standards/Ecma-367.htm.

Semantic analysis includes processing the result of the parsing step — data structures described next, such as an abstract syntax tree and a symbol table — to discover important semantic information useful for the next steps.

Code generation produces target code from source code. There may be more than one code generation step, as compilers may use intermediate representations before generating the final code. From Eiffel source code, for example, the EiffelStudio compiler generates bytecode, which is available for interpretation (as part of the Melting Ice Technology discussed later) but also serves as intermediate code from which the compiler can generate a final target.

Optimization improves the code generation process to ensure the production of more efficient code. Optimization can occur in conjunction with several of the preceding tasks, such as semantic analysis or code generation. Examples of optimization include:

- Register allocation — optimizing for execution time. Mathematically $3 * b + a$ has the same value as $a + 3 * b$, but one of these forms may compute faster than the other by using the processor's registers more effectively. Optimization will ensure that code generation produces the fastest variant.

← "Registers and the memory hierarchy", page 287.

- Dead code removal — optimizing for execution space. If the optimizer determines that some part of the program will never be used during execution, it can remove the corresponding generated code or, better yet, avoid generating it in the first place.

> A program that includes elements that will never be executed does not necessarily reflect programmer sloppiness. If your software relies on libraries of reusable components, a simple compilation strategy might compile the entire library, but at any particular stage of its evolution the program uses only a subset of that library. In EiffelStudio, where most programs rely on general-purpose libraries such as EiffelBase, dead code removal often halves the size of the generated code.

Fundamental data structures

In general the lexical analysis and parsing tasks are closely integrated: the parser calls the "lexer" (short for lexical analyzer) to get successive tokens. The main output is an abstract syntax tree, representing the structure of the program stripped of any purely textual property such as keywords. Another fundamental structure is the symbol table; it records the names used in the program — class names, feature names, local variables, other entities — and, for each of them, the associated properties. For example the entry for a local variable will record the type of the variable and the routine to which it belongs. Further properties, useful in semantic analysis and optimization, might include the lists of instructions which use the variable's value and of instructions that may modify that value. *Hash tables*, studied in the chapter on data structures, are often a good implementation for symbol tables.

→ *"Hash tables", 13.9, page 411 .*

In the typical organization of a modern compiler the combined lexing and parsing task will produce a raw AST and a symbol table initialized with basic, purely syntactic information. The role of the remaining tasks is then to enrich, or *decorate*, these data structures with ever deeper semantic information.

← *"Describing abstract syntax", 11.4, page 310.*

Passes

The traditional description of the compiling process emphasized the notion of *pass*. A compiler pass is a traversal of the entire program, intended to perform specific operations on its constituents. Historical circumstances made this notion important: the program representations appropriate for each step — initially the source text, then the parse tree and so on — would generally not fit in the limited memories of earlier computers; they had to be kept in external storage as files on disk or, earlier yet, tape. Compilation consisted of successive passes, each processing the previous file and producing the next one. The dominant performance goal was to minimize the number of passes.

This concern even had an influence on language design. Pascal, for example, was explicitly designed with strict limits on forward references (such as calls to a routine occurring before the routine's declaration) to permit one-pass compilation.

Today the situation is different and the notion of pass less clear. In the most common scheme, the compiler first applies a single pass — the only one clearly recognizable as such — combining lexical analysis and parsing to produce an abstract syntax tree and a symbol table. The rest of the compilation processes and decorates these data structures.

The compiler as verification tool

To appreciate the full role of compilers, we should remind ourselves of an observation already encountered several times: the compiler is — in addition to a program translation tool — a program verification tool. For modern typed languages with their elaborate type systems, the type properties of a program provide considerable semantic information. The validity rules of Eiffel are an example, describing a rich set of consistency properties all meant to improve the reliability of the software you write. By performing validity checking, the compiler enforces these rules, catching violations at compile time.

Software engineering research has shown the considerable benefit of detecting errors early; in particular it costs much more to correct an error if it is found dynamically, during execution, rather than statically. A compile-time error, in this view, is good news.

→ "Touch of Methodology: Zen and the Art of Reacting to Compiler Messages", page 367.

Loading and linking

Machine-code programs need memory addresses: an assignment *x* := *exp* will put a value in the memory address associated with x; a conditional instruction **if** *c* **then** *a* ... will branch to the address of subsequent code if *c* evaluates to false; a routine call *r* (...) or *x*.*r* (...) will transfer execution to the address of the code for *r* and, on termination of that code, must branch back to the address of the code following the call.

← Using the scheme given in "The goto instruction", page 183.

The exact addresses are beyond the control of the compiler. On a typical platform, many programs run concurrently. A special program of the operating system, the **loader**, is responsible for starting other programs. Whenever it launches a program, the loader must find memory space and adjudicate it to the program's code and data. This scheme prevents the compiler from including the final addresses in the generated code:

- When processing software elements *within a module*, for example the routines of a given class, the compiler only controls their *relative* addresses in the memory area assigned to the module. For example when it processes an unqualified call *r* (...) appearing in a routine of the same class *C* as *r*, the compiler knows the offset of the code for *r* within the area assigned to *C*; but it does not know the corresponding *absolute* addresses, which will only be set at loading time and may vary from one execution to the next.

- For elements *in a different module*, addresses will be relative to the start address of that module. If the entire program is compiled together this case falls within the preceding one, as the compiler could define the layout for all the modules. But often it is desirable to allow **separate compilation**, where modules can be compiled individually before being combined into a single target program.

The first problem has two possible solutions. Some operating systems use a *relocating loader* which will, before execution, add a constant value (the memory area's start address) to every relative address. The more common solution is to use a hardware platform designed to handle the notion of relative address directly: all instructions interpret their address arguments relative to the program's memory start address.

The second problem requires the use of another operating system program: a *linker*, which combines several target modules into a single module. Any one of the linker's input modules may include *unresolved* references, which the linker will replace whenever possible by addresses obtained from other modules. The process can be iterative: if not all references are to targets in the modules being linked, the linker's output can still have unresolved references, to be filled later by a new linking phase.

Linking is a conceptually simple operation but can have a significant performance impact because it may have to traverse the entire set of modules being linked, making linking time potentially proportional to total program size. This goes against the trend towards incremental compilation, itself resulting from a desire to provide fast turnaround after a program change.

The runtime

Today's ambitious programming languages require not only sophisticated compile-time tools (including a sophisticated compiler) but also sophisticated execution-time support: when a program runs, it needs dynamic memory allocation (for instructions such as **create** *x*), automatic garbage collection to reclaim objects that are no longer accessible, exception handling, support for input and output. The hardware generally does not provide these mechanisms directly. Effective memory management, in particular, relies on complex algorithms and data structures.

← *"Memory management and garbage collection", 6.7, page 128.*

Since these needs are common to all programs written in a given language, it would make no sense for the compiler to generate the corresponding code separately for each. Instead, the generated code will include, where the program needs one of these facilities, calls to routines from a library known as the run-time system, the run-time library or just "*the runtime*". The code must then be linked with the runtime before execution.

Or even several languages, as in the .NET framework which provides a multi-language execution infrastructure.

Another way of stating this role of the runtime is to relate it to the notion of virtual machine. While typical machine code and the visible properties of actual hardware — other than speed and size — have not changed much in a half-century, today's programming languages expect more advanced virtual machines. We have seen that it is possible to build such a virtual machine with its own instruction code such as bytecode, and target the compiler to that code, which will then be interpreted; but the performance overhead may be unacceptable. Here we encounter a different approach: generate the machine code of the actual hardware, but provide the more advanced mechanisms through a runtime. In this case the virtual machine is the combination of the hardware and the runtime.

Bytecode-based virtual machines also include a set of mechanisms supporting the execution of bytecodes; so the notion of runtime applies to them too.

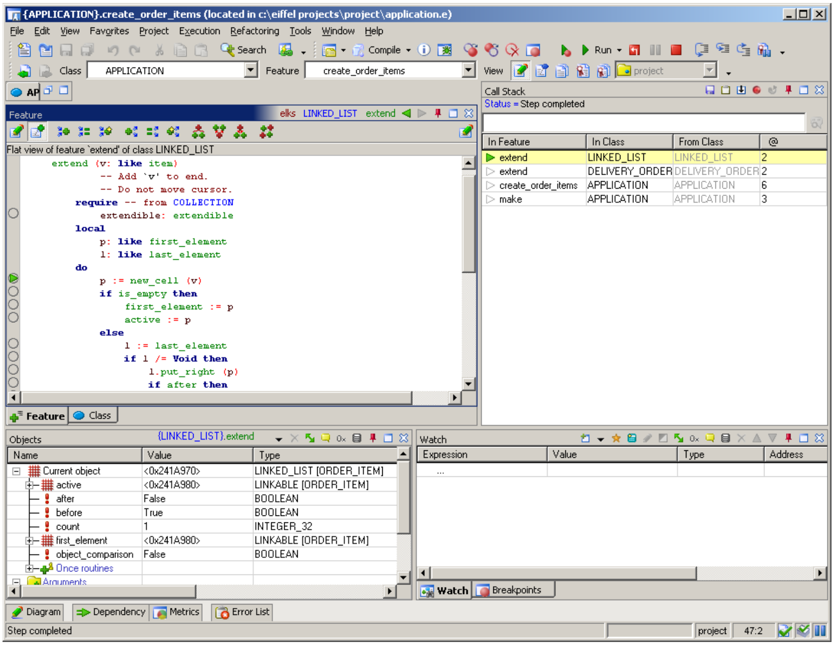
Although typically a smaller program, the runtime is as necessary as the compiler for the processing of programs in a modern object-oriented language.

Debuggers and execution tools

Once your program has been compiled and linked, you will want to exercise it. The final version will typically be a self-running executable, but that's only when you have completed the development. Until then, you will want to try out the execution under controlled conditions, enabling you for example to explore the execution context (precise location in the program text, object contents) in case of an execution failure. To do this you need a debugger. The term is actually too specific — too pessimistic — since such tools are useful even when you are not specifically looking for a fault ("bug"); modern debuggers are control tools enabling you to monitor what the execution of programs.

A typical debugger will provide such facilities as: define, in the source code, "breakpoints" where execution will stop; start execution; interrupt execution; resume execution; terminate execution. When execution has stopped — which can arise from three possible causes: reaching a breakpoint, triggering a failure, or responding to a user interrupt — the debugger will let you examine the code that led to the current state, explore the object structure by looking up object contents and following references, evaluate expressions dynamically, and perform other inspections on the program and its data. In some cases, such as the EiffelStudio debugger, you can even simulate backward execution of the program, a useful facility if you encounter a failure and want to understand what happened before. The figure on the facing page shows a typical state of the EiffelStudio debugger.

The availability of a good debugger is not an excuse for sloppy programming (resting on the hope that any problems will be found at debug time). Execution control can only tell you about a few cases out of a myriad of


{APPLICATION}.create_order_items (located in c:\eiffel projects\project\application.e)
File Edit View Favorites Project Execution Refactoring Tools Window Help
Search
Compile
Run
Class APPLICATION
Feature create_order_items
View
project
AP
Feature
elks LINKED_LIST extend
Flat view of feature `extend' of class LINKED_LIST
Feature
Class
Call Stack
Status = Step completed
Objects
{LINKED_LIST}.extend
Watch
Diagram
Dependency
Metrics
Error List
Watch
Breakpoints
Step completed
project
47:2

```
    extend (v: like item)
            -- Add `v' to end.
            -- Do not move cursor.
        require -- from COLLECTION
            extendible: extendible
        local
            p: like first_element
            l: like last_element
        do
            p := new_cell (v)
            if is_empty then
                first_element := p
                active := p
            else
                l := last_element
                if l /= Void then
                    l.put_right (p)
                    if after then
```

In Feature	In Class	From Class	@
extend	LINKED_LIST	LINKED_LIST	2
extend	DELIVERY_ORDER	DELIVERY_ORDER	2
create_order_items	APPLICATION	APPLICATION	6
make	APPLICATION	APPLICATION	3

Name	Value	Type
Current object	<0x241A970>	LINKED_LIST [ORDER_ITEM]
active	<0x241A980>	LINKABLE [ORDER_ITEM]
after	False	BOOLEAN
before	True	BOOLEAN
count	1	INTEGER_32
first_element	<0x241A980>	LINKABLE [ORDER_ITEM]
object_comparison	False	BOOLEAN
Once routines		
Arguments		

Expression	Value	Type	Address
...			

A debugger session with EiffelStudio

possible executions. More generally, dynamic verification techniques such as debugging are not a substitute for static verification and validation; it is always better to avoid problems than to cure them. But debuggers let you experiment with your program and get a concrete feel for what happens at run time.

→ On static verification and validation see next section and "Static techniques", page 732.

12.4 VERIFICATION AND VALIDATION

Debuggers typically support program checking performed by the program's developers themselves. Before being released, programs must generally be submitted to more systematic verification and validation ("V&V"), often by different people for more objectivity. (Verification refers to checks of internal consistency; validation, to checks of adequacy to intended purposes.)

The chapter on software engineering has more on V&V. For the moment we simply note that the corresponding tools belong to two major categories:

→ "Verification and validation", 19.7, page 727.

- *Static analyzers* rely on the program text. An example is the enforcement of type rules and other validity constraints by a compiler, but static analysis tools go further, all the way to possible *proofs* of program correctness.
- *Dynamic* techniques must execute the program, especially by *testing* it against expected results.

12.5 TEXT, PROGRAM AND DESIGN EDITORS

To enter program modules and other software elements (such as design documents and other documentation), you may use text editors — the programs that enable us to type and format documents.

When these documents are programs, two possibilities present themselves: a generic text editor, which you happen to use for texts in a particular programming language; or a specialized program editor, which knows about that language and is able to interact with other programming tools such as the compiler. There are pros and cons to each approach:

- Good generic editors offer many sophisticated features, for example ways to perform complex changes, or a command language to process the document in specific ways. But they are not specifically tailored to programming and miss some facilities specific to programs; for example, they typically do not know that program texts will be processed by a compiler and executed under the control of a debugger.
- A specialized program editor can take advantage of specific knowledge about the language; for example it can parse your program texts as you type them, show the location of errors directly in the text, offer a button to start a compilation from within the editor, and provide other direct connections to the development tools. But in its language-independent text editing capabilities it may be less advanced than a general-purpose text editor.

Such technical considerations are not the sole determinant. Text editors — such as Vi and Emacs, both notable for the enthusiasm they generate in their respective user communities — can be addictive; people used to the conventions of such a general tool may resent being asked to change their practices just because they are typing a program rather than plain text.

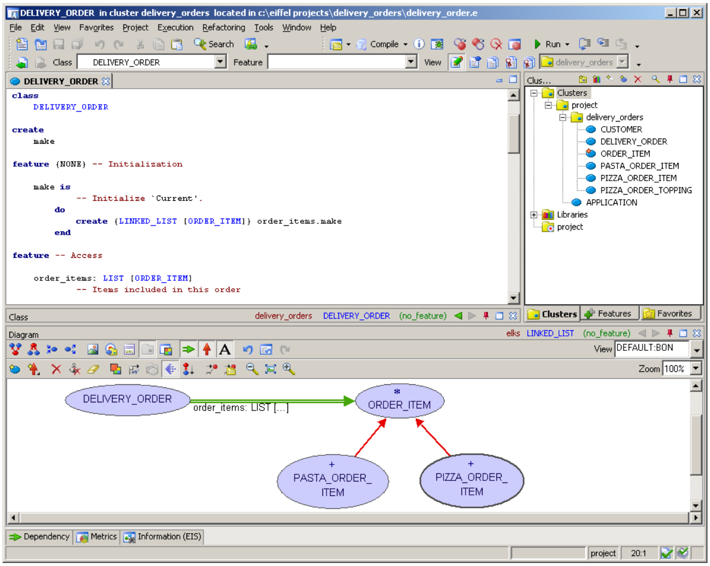
Another reason why program editors have not displaced generic editors even for entering programs is that today's generic editors can often be parameterized to support the syntax of a particular programming language (or other precisely defined notations such as HTML). Making such an editor support a new language simply means providing it with a grammar description in BNF or equivalent. The generic editor can then provide some of the benefits of a program editor; examples are syntax coloring to distinguish keywords and various kinds of syntax elements, and automatic completion (you type the beginning of a construct specimen, for example **if**, and for the rest a template appears with keywords pre-filled, such as <Condition> **then** <Instructions> **else** <Instructions> **end** showing where to enter the missing elements).

These observations indicate that a development environment should offer a specialized editor that supports the language or languages of interest, but not impose its use; it should also accept texts prepared with other tools.

> As an example, EiffelStudio offers a built-in editor for class texts and would have an easier job if it could assume that this is the only way for users to enter and type classes: it could then easily keep track of changes, facilitating the automatic recompilation process described below. But it must accept that users will also rely on other text editors; in that case the compiler must analyze files and their time stamps (times of last modification) to know what to recompile.

Besides purely textual tools, it may be convenient to use graphical representations of program texts, such as the diagrams used in this book to describe software architectures as sets of classes with client and inheritance links. The notation we use is called BON (Business Object Notation); another one, more widely used and more complex, is UML (Unified Modeling Language). Graphical tools support these notations: they let you enter a diagram interactively, then will automatically generate the program text, or at least the overall structure. They are often called CASE tools (for Computer-Aided Software Engineering — a term that literally covers all the tools of this chapter but is generally used in this more restrictive sense). A well-known example supporting UML is Rose from Rational Software. EiffelStudio includes a Diagram Tool providing graphical displays of classes and clusters:

→ E.g. “An inheritance hierarchy”, page 554. See also the diagram on page 565.



DELIVERY_ORDER in cluster delivery_orders located in c:\eiffel projects\delivery_orders\delivery_order.e

```
class
    DELIVERY_ORDER

create
    make

feature {NONE} -- Initialization

    make is
            -- Initialize `Current'.
        do
            create {LINKED_LIST [ORDER_ITEM]} order_items.make
        end

feature -- Access

    order_items: LIST [ORDER_ITEM]
            -- Items included in this order
```

Text view (top) and Diagram Tool of EiffelStudio

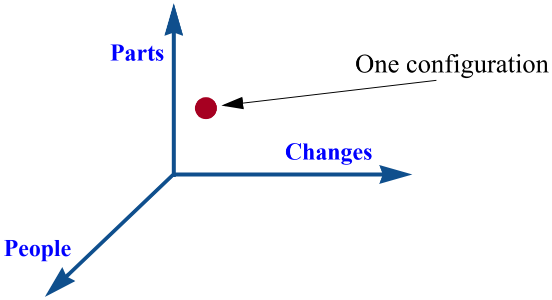
Another example, from Traffic, appears on page 555.

An important requirement on such tools is **round-trip engineering**: the guarantee that transforming from graphics to text and backwards, in either order. will yield back the original. Graphical views are, for many people, a clear way to describe overall structures; but when it comes to precise semantic properties nothing beats text. Round-trip engineering guarantees the consistency of these views: the tool should enable you to start from the text or the pictures as you please, then work on the pictures through the graphical interface and have the relevant classes immediately regenerated, or modify the text and have the pictures automatically updated. This is the principle behind EiffelStudio's Diagram Tool.

12.6 CONFIGURATION MANAGEMENT

Software almost always changes over time. Software systems often have many parts. Software is commonly developed by many people.

Put these *three* characteristics together and you have a serious administration problem.



The three dimensions of software configuration management

Put any *two* of these characteristics together and you still have a problem. In fact, any *one* of them already calls for configuration management.

Varieties of configuration management

The particular issue that gives configuration management its name is the task of ensuring the consistency of combinations, or "configurations", of the parts making up a system. Consider just the "Parts" and "Changes" dimensions. In the case of programs the parts are modules, such as the features, classes and clusters of object-oriented programming. Each module goes through successive versions, according to its own schedule and constraints; the process is more counterpoint than harmony, more Bach fugue than military march. But the system as a whole also needs to progress through its own version history: every so often you must put the pieces together, in their current state, and produce a release. This is called a **build** in the trade jargon, and is where disaster threatens.

It is ever so easy to use version 3.1 of module A and version 2.5 of module B, whereas the A version has only been certified to work with version 2.4 of B. Many software catastrophes on record can be traced to such seemingly trivial management mishaps, which cease to be trivial as system size and project duration grow.

There is more to configuration management. The issues can be tricky even for the development of a single module. Typical questions are:

- *When* was the module last modified?
- *Who* modified it between September and December of last year?
- When was version *n released*?
- *What* changed between version *n* and version *n* + 1?
- What was the *reason* for this particular change?
- Does this bug *still* exist in the current version?
- If not, when was it *fixed*?
- Can we *revert* to the version of module *M* as of March 15 of this year?

This aspect of configuration management is often called **version control**.

The most widespread configuration management tools address the two questions just described: automatic build and version control. We review these two facets of configuration management. The next section will discuss total project repository platforms, which extend configuration management by providing general project infrastructure.

Build tools: from Make to automatic dependency analysis

The inspiration for build tools is the Unix Make command developed by Stuart Feldman in 1977. The idea is to automate the reconstruction of a system on the basis of a description of module dependencies, known as a "make file" or "makefile". A makefile is a list of entries of the form

Feldman (2006)

```
target: source1 source2 …
    command1
    …
```

This states that the target is dependent on *source1*, *source2 …*; usually the target and sources are files. Whenever any of the sources have changed, the corresponding *target* is no longer up to date; the listed command or commands are in charge of updating it. Executing

The tool's name is often written "make", all lower case, as this is how you must type it to execute the command-line version.

```
make target
```

will reconstruct *target* from its sources by applying the corresponding command; if any of these sources is itself the target of a dependency, it will first be reconstructed in the same way. In this process the order of entries in the makefile does not matter: the **make** processor deduces from the dependencies the order in which to apply the commands. The language of makefiles is indeed — unlike most programming languages, which specify execution order precisely — of a descriptive style.

The notion of dependency enforced by Make also involves update time. The operating system records the time stamp (last modification time) of every file; Make will only apply the commands associated with a dependency if one or more of the sources have a more recent time stamp than the target.

A typical makefile, for building a program written in C, is

```
program: main.o module1.o module2.o
    cc main.o module1.o module2.o
%.c: %.o
    cc $<
```

The C practice is to store source modules in files called *name*.**c**; the compiler, command **cc**, will generate the object code into a file *name*.**o**. Command **cc** doubles up as a linker: applied to one or more **.o** files, it produces a new object module with cross references removed. The first entry says that our program depends on three object modules, as listed (*main* is the main program module); generating the program requires applying **cc** to them. To describe how to get the **.o** modules, we could use three entries of the form

```
main.o: main.c
    cc main.c
```

and similarly for *module1* and *module2*. The second rule in the makefile subsumes them into a general rule applicable to any **.o** file; the symbol % is a placeholder, so the first line says that any *name*.**o** depends on *name*.**c** for the same name; in the command of the second line $< denotes whatever was matched in the second part of the first line. (At this point you could be forgiven for remarking that the notation is not the most limpid possible, but before jumping to conclusions remember that since 1977 the original Make and its avatars have helped millions of programmers build their programs right.) The command **make** *program* will do the expected: first compile the three modules to produce their respective **.o** files per the first entry; then link them (again through **cc**) to produce *program*. Note again that Make automatically determines the order of these operations from an analysis of the dependencies.

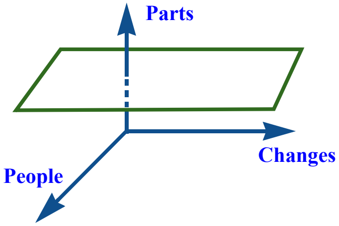
The concepts are applicable beyond programming: for example you could define a makefile for documentation generation, with dependencies from source files in a document composition format (Microsoft Word, Open Office, TeX, FrameMaker…) to targets in HTML or PDF, and commands to regenerate these representations.

Make single-handedly established the discipline of build management and is a success story of software engineering. With hindsight its main limitation is the need to describe dependencies explicitly. When modules change, their dependencies change too; you must in each case make sure to update the makefile. Another way of stating that observation is to note that a makefile is software, and must be designed and maintained like the rest of the software. Some simplifications are possible, thanks in particular to built-in generic rules such as the second rule above (specifying how to compile and link C programs); but the process remains tedious. Errors can result in incorrect builds and faulty systems.

A more modern approach is to equip the program with enough information to allow computing the dependencies automatically. This is for example what EiffelStudio does: there is no equivalent of makefiles for Eiffel code because the compiler can determine, when a feature or class changes, what other classes depend on it and hence will have to be recompiled.

Version control

Version control tools help you keep track of successive versions of an individual module: in our three-dimensional picture, it corresponds to a horizontal plane as shown, or in the case of a single developer to a horizontal line. The parts (modules in the case of programs) undergo successive changes. We cannot just let developers edit them at their discretion then recompile the entire system; havoc would follow. Instead we must record every change — who, when, what, why — and, if needed, compare successive versions or revert to an earlier version.



(From the figure on page 344.)

There are many version control tools, open-source and commercial; some are ambitious integrated frameworks, although the most successful seem to have been simpler tools that focus on the essential issues and are easy to integrate into a company's existing software development process. Most notable has been a line of four- then three-letter-acronym tools starting with SCCS by Mark Rochkind (Source Code Control System, 1972. coming out of Bell Labs like Make) and continuing with RCS by Walter Tichy (1982, R for Revision), CVS (1986, C for Concurrent, based on RCS) and more recently SVN (longer name: Subversion), a reimplementation of CVS.

Tichy (2008)

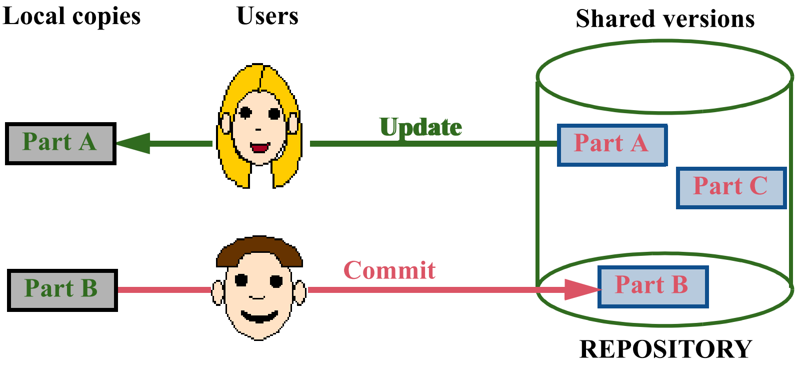
The setup with such a system includes:

- A *repository,* which contains the official successive versions of each part under version control.

  Conceptually the repository is a database — like the customer database of your phone provider, or the account database of your bank — although version control systems do not generally use database technology.

- Local copies of the parts, which users (for example software developers) can keep and modify for their own purposes.

Checking in and out



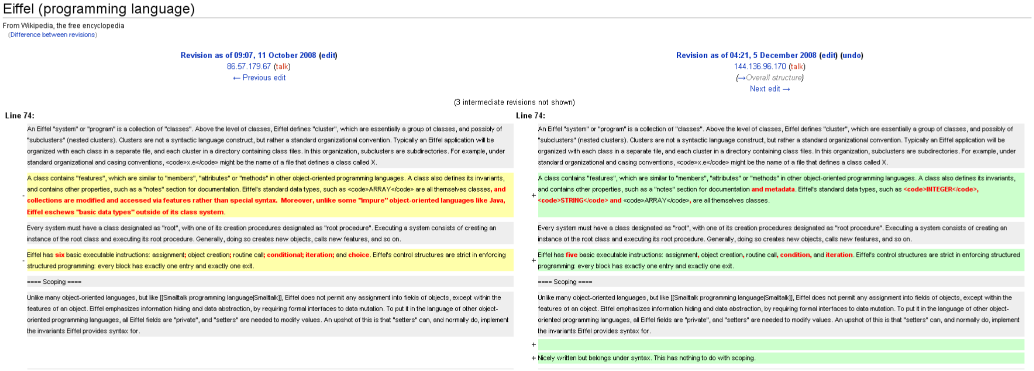
The repository is stored on a server, and users typically access it through a network. The two fundamental operations at their disposal are:

- **Update** (or *check-out*): make a local copy of a part from the repository — by default the part's latest version, but you can specify an earlier one.
- **Commit**: (or *check-in*): enter a part, possibly new but more commonly a modified local copy of an existing part that a user checked out.

A commit creates a new version, with its version identifier. It is customary to use version identifiers made of a sequence of numbers separated by periods, such as 6.3 (the version number of EiffelStudio at the time of printing). 6 is the "major number", changed only for a version that introduces essential innovations, 3 the "minor number"; the next release would be 7.0 in the first case, but will actually be 6.4. Intermediate versions, introducing for example a "patch" that corrects a bug, would have version identifiers such as 6.3.*m* or even 6.3.*m*.*n*.

Conceptually, the repository retains all old versions. This goal would seem to require a prohibitive amount of storage space; what makes it realistic is a technology known in trade talk as "diff" from the name of the Unix command that takes two files and shows how they differ: lines added, lines changed, lines removed. If you have ever clicked "Compare selected versions" in the History page of a Wikipedia entry you will have seen a diff-like view:

It is possible to consider additions and deletions only, representing a replacement by a deletion followed by an addition.



Eiffel (programming language)

From Wikipedia, the free encyclopedia
(Difference between revisions)

	Revision as of 09:07, 11 October 2008 (edit) 86.57.179.67 (talk) ← Previous edit		Revision as of 04:21, 5 December 2008 (edit) (undo) 144.136.96.170 (talk) (→Overall structure) Next edit →
	(3 intermediate revisions not shown)		
Line 74:		Line 74:	
	An Eiffel "system" or "program" is a collection of "classes". Above the level of classes, Eiffel defines "cluster", which are essentially a group of classes, and possibly of "subclusters" (nested clusters). Clusters are not a syntactic language construct, but rather a standard organizational convention. Typically an Eiffel application will be organized with each class in a separate file, and each cluster in a directory containing class files. In this organization, subclusters are subdirectories. For example, under standard organizational and casing conventions, <code>x.e</code> might be the name of a file that defines a class called X.		An Eiffel "system" or "program" is a collection of "classes". Above the level of classes, Eiffel defines "cluster", which are essentially a group of classes, and possibly of "subclusters" (nested clusters). Clusters are not a syntactic language construct, but rather a standard organizational convention. Typically an Eiffel application will be organized with each class in a separate file, and each cluster in a directory containing class files. In this organization, subclusters are subdirectories. For example, under standard organizational and casing conventions, <code>x.e</code> might be the name of a file that defines a class called X.
-	A class contains "features", which are similar to "members", "attributes" or "methods" in other object-oriented programming languages. A class also defines its invariants, and contains other properties, such as a "notes" section for documentation. Eiffel's standard data types, such as <code>ARRAY</code> are all themselves classes**, and collections are modified and accessed via features rather than special syntax. Moreover, unlike some "impure" object-oriented languages like Java, Eiffel eschews "basic data types" outside of its class system**.	+	A class contains "features", which are similar to "members", "attributes" or "methods" in other object-oriented programming languages. A class also defines its invariants, and contains other properties, such as a "notes" section for documentation **and metadata**. Eiffel's standard data types, such as **<code>INTEGER</code>, <code>STRING</code> and** <code>ARRAY</code>**,** are all themselves classes.
	Every system must have a class designated as "root", with one of its creation procedures designated as "root procedure". Executing a system consists of creating an instance of the root class and executing its root procedure. Generally, doing so creates new objects, calls new features, and so on.		Every system must have a class designated as "root", with one of its creation procedures designated as "root procedure". Executing a system consists of creating an instance of the root class and executing its root procedure. Generally, doing so creates new objects, calls new features, and so on.
-	Eiffel has **six** basic executable instructions: assignment**;** object creation**;** routine call**; conditional; iteration;** and **choice**. Eiffel's control structures are strict in enforcing structured programming: every block has exactly one entry and exactly one exit.	+	Eiffel has **five** basic executable instructions: assignment**,** object creation**,** routine call**, condition,** and **iteration**. Eiffel's control structures are strict in enforcing structured programming: every block has exactly one entry and exactly one exit.
	==== Scoping ====		==== Scoping ====
	Unlike many object-oriented languages, but like [[Smalltalk programming language|Smalltalk]], Eiffel does not permit any assignment into fields of objects, except within the features of an object. Eiffel emphasizes information hiding and data abstraction, by requiring formal interfaces to data mutation. To put it in the language of other object-oriented programming languages, all Eiffel fields are "private", and "setters" are needed to modify values. An upshot of this is that "setters" can, and normally do, implement the invariants Eiffel provides syntax for.		Unlike many object-oriented languages, but like [[Smalltalk programming language|Smalltalk]], Eiffel does not permit any assignment into fields of objects, except within the features of an object. Eiffel emphasizes information hiding and data abstraction, by requiring formal interfaces to data mutation. To put it in the language of other object-oriented programming languages, all Eiffel fields are "private", and "setters" are needed to modify values. An upshot of this is that "setters" can, and normally do, implement the invariants Eiffel provides syntax for.
		+	
		+	Nicely written but belongs under syntax. This has nothing to do with scoping.

This representation of the "diffs" between two versions of a file, say file_{n-1} and file_n, is intended for human inspection; but the diff algorithm can also generate a form d of the diffs that allows a companion algorithm to reconstruct file_n from file_{n-1} and d, or, backwards, file_{n-1} from file_n and d.

> That companion algorithm is straightforward: d describes a sequence of line additions, line deletions and line replacements at given positions; it suffices to apply these operations to the file, in order. The diff algorithm itself is more delicate.

A consequence for version control systems is that the repository only needs to store the original version of each file and successive diffs; if there is a request for a particular version it will be recomputed. Alternatively, the repository can always store the latest version — to save time, since it is the one that will be requested most often — and store backward diffs. Either way, the advantage is that diffs are typically much smaller than the full file, so the solution optimizes space usage and makes it possible to store the entire history of a file, sometimes going back a decade or more and remembering thousands of revisions.

As the Wikipedia case illustrates, revision control is useful beyond program modules. For example in our group at ETH many people work on the slides for large courses; all the slides are kept in a Subversion repository.

In the case of software development, version control is applicable not only to program modules but also to all other documents of a software project, from user requirements to design documents and test results. Version control is easy to use — through a simple discipline of updating any part before you change it, and committing it afterwards — and averts many potential disasters. The following rule is one of the most important that you should remember for the practice of software development:

Touch of Methodology: Use version control

Keep all code and all relevant documents of a software project under the management of a version control tool.

In committing, always write down the reason and nature of the changes.

The second part of the advice relates to the possibility, when you commit a new version, of entering into the version management system a message that will be retained along with the changes. You can leave it blank, but don't: just as you should always include a header comment at the time you write a routine (rather than trying a year later to remember what the routine was about), so should you record the reason and context of every change while they are fresh in your mind. Systematically applying this guideline means that over time the version control repository becomes a rich knowledge base about the evolution of your software.

← *"How long is this line?", page 55*. See also *"Touch of Methodology: Placeholder routines", page 221*.

The version control scenario assumed so far functions well with a single developer. A more delicate situation arises when several people must work on the same module, for example a class, typically as part of requests for separate improvements to the software. It is possible to lock a module on update so that no one else can check it out, but this is generally too drastic. When you commit your changes after someone else committed his or her own changes to the same module, the version control tool will detect the conflict and ask you to resolve it, helping you by displaying a "diff". In many cases the reconciliation process is straightforward as different developments tend to affect different classes or, within a class, different features, but once in a while you will get a real conflict which you will have to resolve carefully. The key in such a process is not to wait until too many changes make the task inextricable:

Touch of Methodology: Commit early and often

Commit software after every significant change, to minimize conflicts and facilitate their resolution.

A companion piece of advice involves "branching":

Touch of Methodology: Branching

Do not create a new version control branch unless you intend it to lead to a separate product, separately maintained.

Branching is the facility, available in version control systems, to split a product into two, each with its own sequence of version identifiers. The problem with branching is that each of the branched lines gets a life of its own, and after a while reconciliation becomes an arduous task.

Branching is a temptation when several people start working on the same code and extend it in different directions; it is the easy solution in the short term, enabling these developers to work independently by putting off reconciliation to some time in the future. One of the pieces of wisdom that the software community has collectively acquired is that this is almost always a bad idea. The recurring little nuisance of having to resolve diffs every few days or so is far preferable to the Big Bang of putting together a set of developments that each seemed over the past few months to proceed smoothly — and turn out to have major incompatibilities that cause uncontrollable disruptions and delays.

Unlike in physics, the Big Bangs of software happen at the end.

The only case for branching allowed by the above advice is the creation of a new product line.

> For example EiffelStudio has a research version, Eve ("Eiffel Verification Environment"), which branched at the time of 6.2. In this case the two are still expected to remain in sync, with regular reconciliations, as the changes tend to affect distinct parts of the system.

Configuration management — both the basic tasks sketched here, build management and version control, and more advanced applications — is one of the principal "best practices" of modern software engineering, which every project large or small should apply.

12.7 TOTAL PROJECT REPOSITORIES

Build management and version control address specific issues of project management. Complementing these individual solutions and often integrating them, "total project repository" platforms have appeared in recent years, providing any project with a single home. The best known such platform is SourceForge. Another example, built at ETH, is Origo.

origo.ethz.ch.

The idea of such platforms is to provide in a convenient and consistent way the set of facilities that every project will need, and that it would otherwise have to procure from many tools in an inconsistent fashion. For example if you create an Origo project you can get the following elements automatically created for you: a version control repository (using Subversion), a web site, registrations for the administrators, developers and users you select (and to whom you grant appropriate privileges), electronic discussion forums, Wiki pages for the project documentation and others.

The scope of these tools is not limited to software projects, since just about any other collaborative endeavor can also benefit.

12.8 BROWSING AND DOCUMENTATION

Large software systems include many components connected by many relation links, such as client and inheritance in the object-oriented world. Also in that world, features undergo many avatars in their journey through the inheritance structure: classes can redefine them, rename them, undefine them.

Browsing tools help programmers find their way through this maze. Among typical questions they help answer:

- What are the parents of class *C*? Its heirs, ancestors, descendants? Its clients, its suppliers?
- In what ancestor of *C* was feature *f* first defined?
- In what ancestor can I find the version of *f* applicable to *C*?

A related task is the production of **documentation** from the software text. You may want to produce class versions at different levels of abstraction (such as the contract and interface views of a class) and in various formats such as HTML and Postscript. As much as possible the process should be automatic, extracting information from the software as written rather than relying on separate documentation; the risk of such external information is that it can become obsolete as the software changes. Since the code may not contain all information of interest, some programming languages provide special constructs to state supplementary properties in the program text; examples are the Javadoc format, supporting structured comments which tools can process, and the **note** clauses which can (and should) appear in an Eiffel class to describe its general properties.

← Contract view: "What characterizes a metro line", page 53.

The documentation tools also retain *header comments* of features; this is one of the reasons why you have been repeatedly advised never to omit these comments.

12.9 METRICS

Computer science is not a natural science: its objects of study are human creations. These creations are large and complex enough to warrant the same kind of empirical, quantitative analysis that scientists apply to the objects of the physical world. Metrics tools support their process.

Properties to be measure include process attributes, characterizing the software development effort, and product attributes, characterizing the code and other outcomes of that effort. Measurable process attributes include development time (global, by team member, by module), development cost, number and type of faults ("bugs") detected. Measurable product attributes include such factors as:

→ "Process and product", page 705.

- Code size (using well-defined metrics, from number of source lines to size of generated code but also number of classes, number of features, number of exported features, percentage of the code devoted to contracts).
- Coverage of the requirements (what percentage of the system's original objectives, or "*function points*", have been realized?).

- Other measures of functionality such as the number of distinct user interface states or screens.

Metric tools collect such data. They can be very useful if they apply sound, well-defined metrics, and are part of a general quality policy that uses the measurements' results to help improve the software development process.

12.10 INTEGRATED DEVELOPMENT ENVIRONMENTS

Progress in software engineering tools has not just produced individual solutions to individual development tasks as discussed until now — compilers, interpreters, program editors, linkers, configuration management tools … — but coherent tool suites known as Integrated Development Environments or IDEs. Integration here means that users are presented with what looks like a single tool, usually interactive and graphical (command-line facilities are usually provided as well), from which they can perform all software development tasks, or most of them. So instead of preparing your program text with an editor, then saving the file and starting the compiler, then using a debugger to run the program, you do everything from a single place.

Typically the GUI will still show several subwindows — editor, compiler messages, debugger and so on — but they are connected to each other. You can for example combine them to explore various properties of a class: its text in the editor subwindow, its client structure in the documentation subwindow, execution of one of its features in the debugger subwindow. All along you can exchange information between subwindows through mechanisms such as drag-and-drop.

The best-known IDEs today are Eclipse, an open-source environment initially targeted at Java, and on the commercial side Microsoft's Visual Studio.

A comprehensive IDE may include tools covering many or all of the tasks discussed in this chapter: compiling, interpreting, editing, entering and displaying information graphically, linking and running the program, computing metrics, executing and debugging. Some IDEs integrate configuration management support, but it is more common to offer interfaces ("plug-ins") to separate configuration management systems.

12.11 AN IDE: EIFFELSTUDIO

To conclude this discussion we take a look at the EiffelStudio environment, both as a concrete example of Integrated Development Environment and because of its support for the programming concepts you are learning in this book; in particular the compilation technology provides an application of the discussion of compilation and interpretation earlier in this chapter.

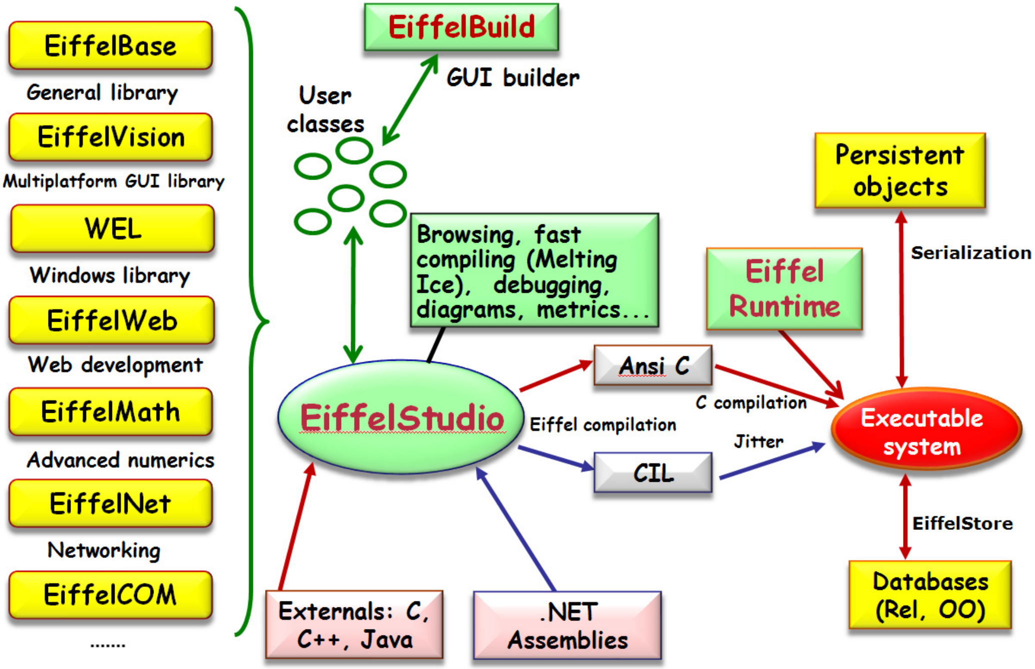
> This is a description of EiffelStudio concepts, not a user's manual; see the corresponding appendix for a quick introduction to using the environment.

→ *Appendix E.*

The implementation of EiffelStudio uses its own technology; the environment represents (at the time of writing) about 2 million lines of Eiffel code (around 6000 classes), plus some supporting C code for the runtime system.

Overall structure

The following figure shows the major components of EiffelStudio.



At the center is the environment's engine, EiffelStudio. It provides users with the key mechanisms: browsing and automatic documentation (see below), compilation, debugging, round-trip textual (EiffelStudio) and graphical (Diagram Tool) design, as well as a *metric tool* allowing you to define your own metrics and apply them to any system or any part of a system.

On the left you see a number of libraries of reusable components addressing various application areas. The fundamental ones are EiffelBase, covering common data structures and algorithms, and EiffelVision, offering graphics with portability across platforms such as Windows and Linux. EiffelBuild, at the top, is a tool for building graphical user interfaces; it generates code that calls EiffelVision features, although you can also use EiffelVision directly.

The bottom part shows mechanisms for interfacing Eiffel code with external software written in various languages and (where available) with facilities of the .NET environment.

The figure also shows the two compilation vehicles: the EiffelStudio compiler can generate C, used here as a kind of portable assembly language, and then compiled into machine code; on the .NET platform it generates the internal bytecode of that platform, known as CIL (Common Intermediate Language), for jitting by the .NET virtual machine. On .NET the code relies on the standard .NET runtime, but in the C-based scheme the result of compilation will be linked with EiffelStudio's own runtime.

Executable systems (on the right) can produce persistent object structures through a simple serialization mechanism, and exchange objects with databases, relational or object-oriented.

Browsing and documentation

The browsing and documentation facilities of EiffelStudio are particularly developed, reflecting the variety of language mechanisms for structuring systems and the versatility of inheritance. The basic metaphor is *pick and drop*: pick a "pebble" representing a software element, and drop it into a hole to perform a suitable operation on that element. In more detail:

- The elements that can be picked include classes, features, clusters and others such as error messages.

- You start a pick-and-drop by right-clicking on any representation of the element (a name, an icon) appearing anywhere.

- After picking, the cursor changes into a symbol representing the type of the selected element, for example 🔵 for a class and ✚ for a feature.

- You drop the icon into a hole by right-clicking again. In many cases an entire window or subwindow acts as a hole; for example if you drop a class into an editor window the window will "retarget" itself to the class, displaying the class text so that you can edit it.

- For comfort, you do not need (as in drag-and-drop mechanisms) to hold the mouse key down while moving the pebble. You just right-click once to pick it, and once to drop it. A left-click will cancel the process.

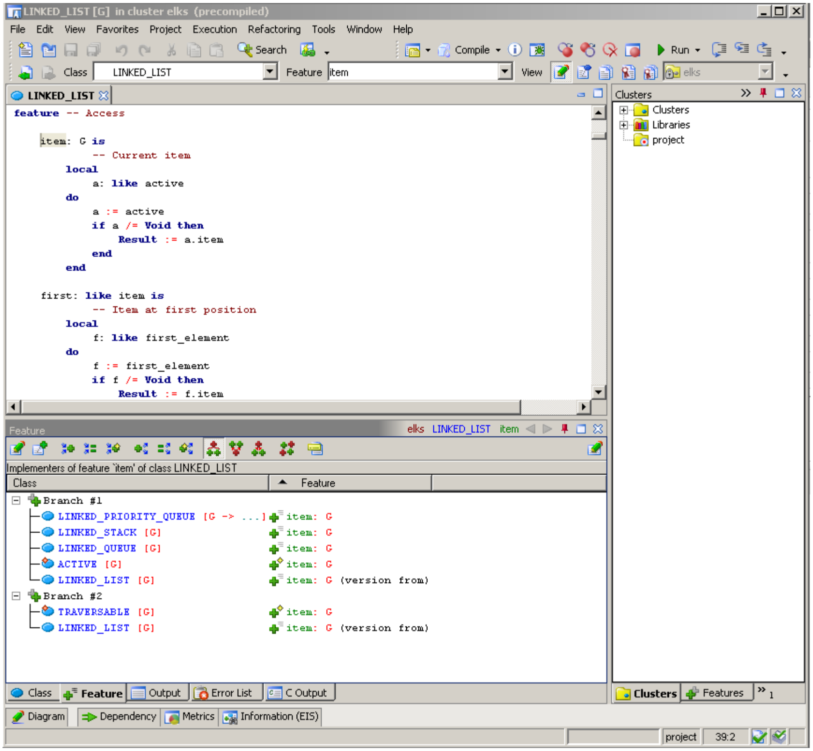
A rich set of documentation mechanisms complement pick-and-drop. For any class or feature you can display a number of **views** at various levels of abstraction, such as the contract, interface and flat views of a class, but also the lists of its clients, suppliers and features; for a feature, you can trace its *history* in ancestors and descendants.

← On the contract view: “What characterizes a metro line”, page 53.

As an example of such views, the screenshot below shows the “implementers” of the feature *item* of *LINKED_LIST*: all the classes and routines where the feature gets a new implementation. It was obtained by pick-and-dropping the feature name from the top part to the bottom part and selecting the *implementers* view. Any feature, class or cluster name appearing anywhere in the display can become the target of a pick-and-drop.

→ We will see how to provide a new implementation for a feature in “Redefinition”, 16.6, page 570.

Implementers of a feature


LINKED_LIST [G] in cluster elks (precompiled)
File Edit View Favorites Project Execution Refactoring Tools Window Help
Search
Compile
Run
Class LINKED_LIST
Feature item
View
elks
LINKED_LIST
Clusters
Clusters
Libraries
project
Feature
elks LINKED_LIST item
Class
Feature
Class Feature Output Error List C Output
Clusters Features
Diagram Dependency Metrics Information (EIS)
project 39:2

```
feature -- Access

    item: G is
            -- Current item
        local
            a: like active
        do
            a := active
            if a /= Void then
                Result := a.item
            end
        end

    first: like item is
            -- Item at first position
        local
            f: like first_element
        do
            f := first_element
            if f /= Void then
                Result := f.item
```

```
Implementers of feature `item' of class LINKED_LIST
Class                                    Feature
Branch #1
  LINKED_PRIORITY_QUEUE [G -> ...]       item: G
  LINKED_STACK [G]                       item: G
  LINKED_QUEUE [G]                       item: G
  ACTIVE [G]                             item: G
  LINKED_LIST [G]                        item: G (version from)
Branch #2
  TRAVERSABLE [G]                        item: G
  LINKED_LIST [G]                        item: G (version from)
```

You may also display such information, as well as any view that the environment can display, in HTML, PDF, RTF (Microsoft Word) and other formats (it is easy to add another format by writing the corresponding “filter”). If you work collaboratively, this facilitates sharing information about a software project, from the most detailed (source code) to abstract views and diagrams.

The melting ice technology

EiffelStudio’s compiling technology combines interpretation and compilation to ensure both fast turnaround and good run-time performance.

The underlying observation is that the most important compiling scenario in the day-to-day life of a developer is not the compiling of a large system from scratch — no one spends months writing code without compiling it — but *recompiling* a system after a set of changes following a previous compilation. The system can be small or large; the changes can be small or large. For a small system any decent compilation mechanism will be fast enough; the critical case is that of a *small* change to a *large* system, say EiffelStudio itself with its two million lines of code. You work for a few minutes to make a few specific changes or extensions; then you will want to see the results *immediately* by restarting interactive execution or running a test case.

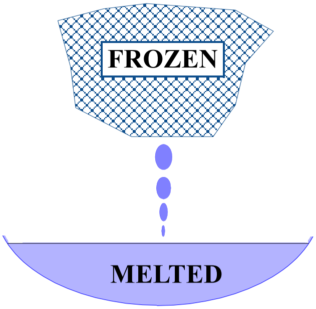
This natural attitude of any developer yields a design constraint on the environment:

Melting Ice Principle

The time to re-process a system after a change should only depend on the logical size of the change, not on the size of the system.

The compilation technology follows from this principle. The metaphor that inspires it is illustrated below. Think of a compiled system as a block of ice; the changes are like melted drops of water, dripping from the ice as a result of the heat generated by your work.

The melting ice



Melting is the process of making changes to your software; freezing, of putting back the melted parts into the “freezer” by recompiling them.

As you make changes, the resulting melted parts are not by default recompiled; EiffelStudio instead generates bytecode meant to be interpreted. But this only affects the changed parts, typically a very small part of the system (how much of two million lines will you change in five minutes or half an hour?); the rest remains compiled, so that the effect on performance is small. The whole mechanism requires that interpreted and compiled code know about each other; in particular, if a compiled routine calls a routine that has been modified, the version to use is the bytecode version; the compiled code must include a switch to ensure that the right version is called.

All of EiffelStudio is open-source, so it you want to understand the details of this delicate machinery you can simply inspect the code.

eiffelstudio.origo.ethz.ch.

The melting mechanism is automatic: you just make your changes — either through EiffelStudio editing or through an external tool — and hit the Compile button. EiffelStudio will then perform a fast change and dependency analysis to find the minimum set of software elements (down to individual features, not necessarily entire classes) that must be recompiled. This does not just include the parts that you explicitly changed or added, but also anything else that the change may affect. Information hiding helps make the process more efficient: if a feature changes but not its interface, clients need not be recompiled. The analysis does not require user intervention, in particular no makefile or equivalent: as noted earlier, there is enough semantic information in the software text itself to allow EiffelStudio to figure out the dependencies automatically. This is one of the benefits of a tool integrated in an IDE and specifically targeted to a statically typed object-oriented language.

As the share of the melted part (the water in the metaphorical bowl, or more prosaically the file containing the generated bytecode) grows, the effect on performance can become noticeable, so once in a while you may need to re-freeze. This is still an incremental operation, recompiling only what's needed, but longer than a melt — typically, for a large system, a few minutes rather than a few seconds. A freeze will also happen automatically after you change or add external code, for example in C, since interpretation cannot work for such code.

Both melting and freezing are "workbench mode" operations, meaning that they generate a system primarily intended for execution under EiffelStudio. A third compilation mode, finalization, generates a stand-alone executable, completely independent of EiffelStudio. Finalization also performs extensive optimizations such as dead code removal, discussed earlier in this chapter, and application of static binding (when equivalent to dynamic binding, as studied in the chapter on inheritance).

→ *"A peek at the implementation", 16.8, page 575*.

In workbench mode these optimizations are impossible. Dead code removal, for example, relies on an analysis showing that certain routines will never be called. But at the next opportunity the programmer might add a call to any routine. This means that such an optimization is not incremental; it requires an analysis of the entire system and is only justified when the compiler generates executable code for the entire system. Finalization is still reasonably fast; continuous progress in compilation techniques and of course hardware performance have brought it down, for the full 2-million-line EiffelStudio, from several hours a few years ago to about 15 minutes (at the time of writing) on a plain desktop PC.

Complementing the three compilation modes that we have seen — melting, freezing, finalizing — a *precompilation* mechanism processes an entire library, such as EiffelBase or EiffelVision, so that it can be integrated into any system without further compilation. Basic libraries are generally used in precompiled form; you will either download the precompiled version, or perform the precompilation yourself at installation time.

12.12 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Software engineering tools are to the software engineer what Computer-Aided Design tools are for designers and engineers in other disciplines such as architecture, mechanical engineering and electronics.
- To implement programs in high-level languages, the two basic techniques are compilation, which turns a program into an internal form, and interpretation, which provides an engine for executing the program directly.
- Some implementations combine compilation and interpretation; for example compilation might produce, instead of machine code, an internal form meant for execution by an interpreter, or more generally a "virtual machine". Another mixed compilation-interpretation strategy is EiffelStudio's Melting Ice, which supports incremental development by interpreting recently changed parts while executing unchanged parts in their compiled form.
- Compilers perform a number of tasks, from lexical and syntactic analysis to semantic analysis, code generation and optimization, which were traditionally implemented as successive passes but today tend to take the form of construction and decoration of a core data structure: abstract syntax tree plus symbol table.
- Linkers combine compiled modules; loaders ready them for execution.
- A runtime provides execution support.
- Editors and CASE tools help build programs through text and graphic input. They should support round-trip engineering

- Build tools such as Make reconstruct a system from its components on the basis of a description of dependencies.
- Version control tools such as Subversion keep track of successive versions of a software part; internally they only store differences between versions.
- Browsing and documentation tools facilitate the exploration and understanding of large software systems.
- An Integrated Development Environment (IDE) supports the principal tasks of software development by providing a collection of interconnected tools through a consistent interface.

New vocabulary

Branching
Browsing
Bytecode
CASE
Commit
Configuration management
Debugger
Diff
DSL (Domain-Specific Language
IDE
Jitter, Jitting
Melting Ice
Metric tool
Pass (of a compiler)
Round-trip engineering
Runtime
Update
Version control
Virtual machine

12-E EXERCISES

12-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

12-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise "Concept map", 11-E.2, page 319.

12-E.3 An interpreter and a compiler

Several exercises ask you to apply the concepts of this chapter by designing abstract syntax classes for a small programming language and write an interpreter a compiler and an "unparser" for it. Since an effective solution requires recursion and inheritance, these exercises appear in the chapters on the corresponding topics.

→ Exercises 14-E.4, page 501 and 16-E.3, page 616 to 16-E.6, page 618 .

PART III:

Algorithms and data structures

The three chapters in this third part cover fundamental algorithmic techniques.

The first reviews some of the data structures and associated algorithms that are the staple of every programmer's workday, and introduces fundamental techniques for reasoning about the efficiency of algorithms, as well as an equally important object-oriented mechanism: genericity.

The second describes in some depth a fundamental (and I think fascinating) programming technique, which is also a reasoning technique: recursion.

The last applies the concepts of these and previous chapters to build a completely engineered solution for an interesting algorithmic problem: topological sort, or the ordering of elements to satisfy a set of constraints.

13

Fundamental data structures, genericity, and algorithm complexity

On one of those evenings when it seems you have done nothing all day but store and retrieve things, have a kindred thought for your programs. Many of them — like Traffic with its list-like structures representing metro lines — spend a good deal of their time putting objects into repositories and searching for previously stored objects.

Such a repository, whose elements we will call "items", is known as a **container**. Lists are only one example, among many kinds differing by the speed of container operations (insert an item, retrieve an item, remove an item, search for items satisfying certain properties, apply an operation to all items…) and the space they require to store the items.

In this chapter we will study some fundamental container structures, useful across all application areas of computing science: arrays, lists of various kinds, hash tables, stacks, queues. This will also be an opportunity to discover three fundamental programming concepts:

- The role of *types* in the development of reliable software.
- *Genericity*: how to declare type-safe container classes.
- *Algorithm complexity*, a technique to estimate the performance of algorithms and data structures.

In passing we will encounter a number of rules of good software design, such as naming conventions for features of reusable components.

13.1 Static typing and genericity

The first issue container structures raise is a *typing* issue.

Static typing

All the entities of our software are declared with a certain type. This rule enables the compiler to check that any operation you want to apply to an entity *x* — for example a feature call, *x*.*f* (*a*) — uses a feature that is indeed applicable to it. The compiler only needs to look up the type *T* with which *x* has been declared:

- There must be a class for *T*.
- It must contain a feature *f*, taking an argument of the right type.

This policy is known as **static typing**: static because type properties are specified in the program text, and so can be enforced at compilation time. The alternative, *dynamic* typing, would forsake type declarations, and wait until run time to find out that a feature call *x*.*f* (*a*) tries to apply a feature *f* to an object that cannot handle it. We saw in the previous chapter that some programming languages, such as Smalltalk, have adopted dynamic typing as their policy.

← *"Definitions: Static, Dynamic", page 11.*

The case for static typing — the regime that prevails in many of today's O-O languages including Java, C# and Eiffel — relies on two main arguments:

- *Clarity*: by declaring every entity with a precise type and every feature with a precise signature, we express the intent behind them and facilitate program reading and maintenance.
- *Reliability*: an invalid feature call is always the result of a programming mistake (a bug). Letting the compiler find the mistake for you beats waiting until run time; as we have seen, the cost of error detection and correction increases dramatically as a software project advances through its lifecycle.

← *"The compiler as verification tool", page 338; see also below: "Touch of Methodology: Zen and the Art of Reacting to Compiler Messages", page 367.*

Static typing for container classes

How can we apply static typing principles to containers? We are already familiar with lists, since we saw that instances of *LINE* are lists of instances of class *STATION*, with features such as

```
extend (s: STATION)                    -- A command
        -- Add s at end of line.
item: STATION                          -- A query
        -- Station at current cursor position.
```

← *E.g. page 60.*

Now assume you want a class *LIST* that can describe lists of *anything*: a list of metro stations, a list of integers, a list of objects of some other known type. The class should have the above features, but you cannot declare the argument *s* to *extend*, or the result of *item*, without knowing the type of list items: *STATION* as above, or *INTEGER* in the second case, or any other type that you have chosen for the objects of a particular list.

You could of course write distinct classes: *LIST_OF_STATIONS*, *LIST_OF_INTEGERS* and so on. You would not want to do that, since the class texts would be identical except for some type declarations. Such duplication or quasi-duplication goes against every principle of economy and reuse.

The idea of genericity is to use a single class, here *LIST*, but **parameterize** it so that it can support many types without reprogramming.

Generic classes

Using genericity, we declare class *LIST* as

```
class LIST [ G ] feature
    extend (s: G )
            -- Add s at end of line.
        do … end
    item: G
        -- Station at current cursor position.

    …Other features and invariant …
end
```

G is just a name; it is known as a **formal generic parameter** for the class. ("A" parameter because there may be more than one.) It denotes a type, so that within the class we may use it for declarations, as here with the argument *s* of *extend* and the result of *item*.

What type does *G* denote? The class itself does not answer this question. To use the class *LIST* in practice you will declare for example

```
first_1000_primes: LIST [INTEGER]
stations_visited_today: LIST [STATION]
```

where it is the responsibility of each example to specify a type, known as an **actual generic parameter** — here *INTEGER* and *STATION* respectively —, to indicate what *G* must represent, within class *LIST*, for the particular list of interest.

This technique solves the problem of static typing for general container classes. Assuming the variables

```
some_integer: INTEGER
some_station: STATION
```

you may use the following valid instructions:

```
first_1000_primes.extend (some_integer)
stations_visited_today.extend (some_station)

some_integer := first_1000_primes.item
some_station := stations_visited_today.item
```

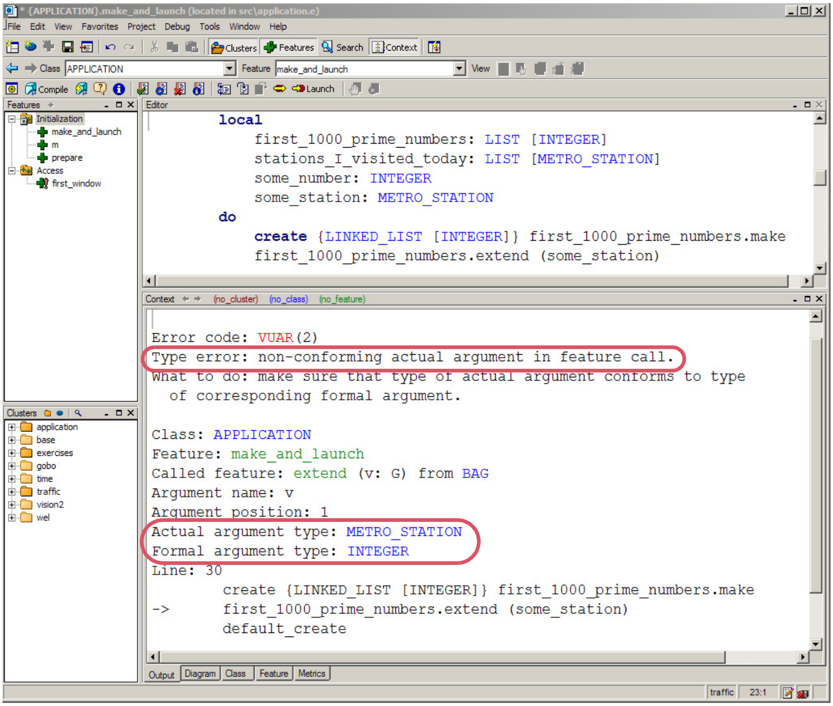
This all satisfies the type rules. The formal argument of *extend* in *LIST* is of type *G*; this means *INTEGER* for *first_1000_primes*, declared as *LIST* [*INTEGER*], and *STATION* for *stations_visited_today*; so it is legitimate to pass as actual argument an integer in the first case and a metro station in the second case. The same applies to the result of *item*.

On the other hand, if you try either of

```
first_1000_primes.extend (some_station)
stations_visited_today.extend (some_integer)
```

Warning: *invalid.*

you will not get past compilation:


* {APPLICATION}.make_and_launch (located in src\application.e)
File Edit View Favorites Project Debug Tools Window Help
Clusters Features Search Context
Class APPLICATION Feature make_and_launch View
Compile Launch
Features
Initialization
make_and_launch
m
prepare
Access
first_window
Editor
Clusters
application
base
exercises
gobo
time
traffic
vision2
wel
Context (no_cluster) (no_class) (no_feature)
Output Diagram Class Feature Metrics
traffic 23:1

```
        local
            first_1000_prime_numbers: LIST [INTEGER]
            stations_I_visited_today: LIST [METRO_STATION]
            some_number: INTEGER
            some_station: METRO_STATION
        do
            create {LINKED_LIST [INTEGER]} first_1000_prime_numbers.make
            first_1000_prime_numbers.extend (some_station)
```

```
Error code: VUAR(2)
Type error: non-conforming actual argument in feature call.
What to do: make sure that type of actual argument conforms to type
  of corresponding formal argument.

Class: APPLICATION
Feature: make_and_launch
Called feature: extend (v: G) from BAG
Argument name: v
Argument position: 1
Actual argument type: METRO_STATION
Formal argument type: INTEGER
Line: 30
          create {LINKED_LIST [INTEGER]} first_1000_prime_numbers.make
->        first_1000_prime_numbers.extend (some_station)
          default_create
```

Until the mid-eighties drivers encountered a single interruption in the entire stretch of Highway 101 from San Francisco to Los Angeles and San Diego: a traffic light in Santa Barbara. This created a perennial traffic jam. To those who complained, Jerry Brown, California governor in the seventies, once replied in a very seventies-California way that they should instead be grateful for the opportunity to pause and reflect on life. This is exactly how you should react to a compilation error such as this one:

Touch of Methodology: Zen and the Art of Reacting to Compiler Messages

When the compiler rejects your class, cut the cursing. Take a breath, have a cup of organic herb tea (optional), reflect on the deeper meaning of life, consider the hours of debugging that might have ensued if the program had been allowed to compile then produce an error at run time rather than compile time, ponder how to avoid such mistakes in the future, and rejoice.

In the present case we are being protected against our own mistakes by the "**type system**" of a modern programming language, specifically its generic mechanism, providing the right combination of safety and flexibility.

The techniques just introduced lead to a bit more terminology:

Definitions: Generic class, generic derivation

A **generic class** is a class that has one or more generic parameters.

A type obtained by providing actual generic parameters to a generic class is a **generic derivation** of that class.

LIST is a generic class; the type *LIST* [*INTEGER*], obtained from *LIST* by providing the generic parameter *INTEGER*, is a generic derivation of *LIST*.

All the container classes studied in this chapter, such as *ARRAY* [*G*], *LINKED_LIST* [*G*], *HASH_TABLE* [*G, H*], are generic. The generic parameter name *G* will always represent the type of items in the container. This is just a convention; you may use for a generic parameter any name that is not also the name of a class in your system.

Genericity is the name of the mechanism allowing classes to have generic parameters, and as a result allowing types to be defined through generic derivation.

While we are on terminology: do not confuse the **arguments** of a routine, formal and actual (representing *values* passed to the routine by its callers), with the **parameters** of a generic class, representing *types* governing a particular use of the class. This convention is not universal — you will find "parameter" used for "argument" — but it is important to keep distinct names for distinct concepts.

Validity vs correctness

The goal of the genericity mechanism is, as noted, to ensure the *type validity* of certain kinds of program (those involving container structures). Genericity is what makes such feature calls as *first_1000_primes*.*extend* (*some_integer*) "valid", meaning that they satisfy the type rules of the language and the compiler will let them through.

This does not mean, however, that such instructions will always work correctly. The target of the call, *first_1000_primes*, might be void at execution time; or *extend* might have a precondition that *some_integer* does not satisfy. We have two different notions at play:

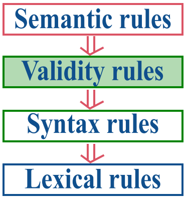
Definitions: Validity, correctness

A program is **valid** if it satisfies all the type rules and other static consistency rules of the language, guaranteeing that certain kinds of run-time malfunctions will never happen.

A valid program is **correct** if it will always execute in accordance with the desired behavior, and never cause a contract violation or other run-time malfunctions leading to failure of the execution.

The definition of correctness only applies to valid programs. Indeed, for a *statically typed* language (a language with precise validity rules, such as Eiffel) it makes no sense to ask about correctness unless the program has passed the validity checks.

> This generalizes the rule that every level of language properties only makes sense if the properties at the preceding levels hold. We already encountered this property for the original three levels of language description, to which validity adds (as the updated figure on the right shows) one more between syntax and semantics. Just as syntax rules are only defined for lexically correct texts, so are validity rules (also called *static semantics*) defined for syntactically correct texts.



(*Original figure on page 44*.)

Examples of the "certain kinds of run-time malfunctions" ruled out by validity include application of a feature to an object that cannot handle it.

Why two notions? It would be good if validity implied correctness, so that once your program has passed muster with the compiler you could go home and rest assured that it will execute properly. Dream on. Although programming languages have been getting better at defining static rules that catch errors at compile time, there remain cases that can only be detected during execution. For these, run-time mechanisms are available, such as exception handling.

← *"An introduction to exception handling", 7.10, page 200.*

Devising a framework in which validity implies correctness is an old quest, the Philosopher's Stone of programming research. The frontier regularly advances; perhaps the most striking recent example is the ability to eliminate *void calls* through type rules, as with the “attached type” mechanism sketched in an earlier chapter: what used to be a source of nasty and unpredictable run-time failures becomes a standard compiler check. This is representative of the general march towards *proofs* of program correctness.

← “Appendix: getting rid of void calls”, 6.9, page 136.

→ On proofs: “Static techniques”, page 732.

Until such proofs become routine, validity and correctness will remain distinct; but you will quickly find out that static typing, especially when combined with techniques of Design by Contract, gives you a tool to kill bugs before they have a chance to bite you.

Classes vs types

With genericity we may take a closer look at the relationship between the notions of class and type.

A type is the description of a set of run-time values: type *INTEGER* specifies the properties of integers as they will be used in your programs, type *STATION* describes the properties of run-time objects representing stations, and so on.

A class is a program module defining a collection of features (and their properties, such as the class invariant) applicable to a set of run-time objects.

The connection is very close: the “set of run-time objects” associated with a *class* is, at least if the class is not generic, a *type*. Any non-generic class such as *INTEGER* and *STATION* is indeed a type, and can be used in type declarations for entities, as in the examples used earlier:

```
some_integer: INTEGER
some_station: STATION
```

This use of classes as both the basic units (“modules”) of program texts, a *static* notion, and the typing mechanism for objects, a *dynamic* notion, is central to the object-oriented style of programming, which would better be called *class*-oriented.

The connection between classes and types remains just as strong with genericity. The new twist is that a class such as *LIST* or *ARRAY* no longer immediately gives us a type; it gives us a template for a type. To get an actual type, it suffices to perform a generic derivation by providing an actual generic parameter. For example:

- *INTEGER* and *STATION* are classes; they are types too. This is true of any non-generic class.
- *LIST* and *ARRAY* are classes; *LIST* [*STATION*] and *ARRAY* [*INTEGER*] are types. This is applicable to any generic class.

We can turn this observation into a precise definition:

Definitions: Class type, generically derived, base class

A **class type** is one of:

T1 A non-generic class.

T2 A generic derivation, that is to say, the name of a class (called the **base class** of the type) followed by appropriate actual generic parameters. In this case the type is said to be **generically derived**.

The notion is "class type" rather than "type" in general since there are a few other kinds, although class types are the most important.

Nesting generic derivations

There remains to clarify what you may use as an *actual generic parameter* for a generic class. The answer is easy to guess: a type. You can see this in the last examples: in *LIST* [*STATION*] the actual generic parameter *STATION* is a type; so is *INTEGER* in *ARRAY* [*INTEGER*].

You perhaps sense something strange in these definitions:

- We have just defined types by stating (clause T2) that they may be obtained from a class and actual generic parameters.
- Now we are defining an actual generic parameter as a type.

Is this a worthless circular definition? No. Just an example of a *recursive* definition, one that builds new elements of a set under definition — here the set of types — by using elements previously obtained under the same definition. The process is clear:

- Through clause T1 of the definition we know for example that *STATION*, a non-generic class, is a type.
- We may then use T2 to deduce that *ARRAY* [*STATION*] is also a type.

Recursion is a fascinating technique, not just for definitions of such concepts but for routines and data structures. We will have a full chapter — the next one — devoted to it, but this example should suffice to show that, in the present case, there is nothing inconsistent or strange in the recursive definition of "type".

→ Chapter 14, in particular "Recursive definitions", page 436 and "Bottom-up interpretation of a construct definition", page 482.

The definition in fact opens interesting possibilities. The type used as actual generic parameter in T2 can follow not just from T1 but also again from T2; in other words it can be generically derived. This allows types such as

```
LIST [LIST [INTEGER]]
LIST [ARRAY [STATION]]
ARRAY [ARRAY [ARRAY [INTEGER]]]
```

and so on without limitations. This is not just a theoretically pleasant possibility, but a practical mechanism, as we will encounter the need for lists of lists, lists of arrays and other multi-level containers.

13.2 CONTAINER OPERATIONS

The second part of this chapter reviews fundamental container structures, from arrays, linked lists and other kinds of lists to stacks and hash tables, all in common practical usage. They exhibit both commonality and diversity:

- Many of their basic operations are the same: insert or remove an item, find out whether a particular item is present, find the number of items…
- Each variant implements these operations differently. This diversity is due to the difficulty of providing equally efficient implementations for all operations. Arrays let you get to an item very quickly if you know its index, but are slow for insertion of new items; linked lists are reasonable for insertion, but slower than arrays for index-based access. Other structures also have their merits and demerits. No single structure is optimal for all situations.

When you need a container, you will have to choose one of the available structures depending on the operations you require.

Before studying each specific container variant, we now review these fundamental operations, looking first at queries, then at commands. *G* will denote the type of a container's items; it is the first generic parameter of the corresponding classes, as in *ARRAY* [*G*] or *LINKED_LIST* [*G*].

Queries

One of the operations we will need for each kind of container is to find out whether a container is empty (has no items). The query, returning a *BOOLEAN*, is called *is_empty*. Its signature is just

```
is_empty: BOOLEAN
```

In other words it takes no argument but is called under the form *c*.*is_empty*, yielding a boolean value for any container *c*.

To find out if a particular item appears in a container we will use

```
has (v: G): BOOLEAN
```

To find out how many items are in a container:

```
count: INTEGER
```

An invariant clause, applicable to all relevant container classes, states:

```
is_empty = (count = 0)
```

To obtain an item from the container — *any* item, chosen by the container's policy, not by the client:

```
item: G
```

Some containers such as arrays instead let you obtain an item given by an integer index, as in "give me the third item". The query is:

```
item (i: INTEGER): G
```

Using the same name as for the previous operation causes no ambiguity because the signature is different.

An integer is only a special case of a **key** enabling you to retrieve an item from some information associated with it. There are many different kinds of key; one of the most common is a **string**, as in a container representing a Web page and allowing a search engine to ask whether certain words appear on the page. For string keys the query will be

```
item (i: STRING): G
```

We will learn how to generalize the key type beyond just strings. Here too reusing the name *item* causes no confusion.

→ Through hashing, in "Hash tables", 13.9, page 411.

Commands

The creation procedure that sets up a container will usually be called *make*. Often it has no argument, but sometimes it takes one indicating an expected number of items:

```
make (n: INTEGER)
```

For all the containers of this chapter, *n* is only an indication to guide the initial creation of the data structure, not an absolute maximum.

→ *See “Touch of Methodology: Don’t box in your users”, page 375.*

The most common operation for adding or replacing an item is called *put*, with one of the following signatures, matching one of the signatures for *item* but with one more argument indicating the new value:

```
put (v: G)
put (v: G; i: INTEGER)
put (v: G; k: STRING)
```

The postcondition should always include the clause

```
inserted: has (x)
```

and, in addition, should express the relationship with the corresponding version of *item*:

- *item* = *v* if *put* has no argument (the first case).
- *item* (*i*) = *v* for the version with an integer index.
- *item* (*k*) = *v* in the last case.

The procedure *put*, when present, may either *add* an item or *replace* an existing one. Sometimes we need to distinguish, using one of

```
extend (v: G)
extend (v: G; i: INTEGER)
extend (v: G; k: STRING)
```

with the postcondition

```
one_more: count = old count + 1
```

or one of

```
replace (v: G)
replace (v: G; i: INTEGER)
replace (v: G; key: STRING)
```

with

```
same_count: count = old count
```

When either of *extend* and *replace* exists, *put* is usually a synonym for one of them, corresponding (if both are present) to the more common use. In all cases, the postcondition clause *has* (*v*) expresses that after you have added an item the structure must answer "yes" if asked about its presence.

The procedure to remove an item is, depending on the context, called *remove* or *prune*.

Standardizing feature names for basic operations

The names cited above — *item*, *has*, *put*…— recur throughout the libraries. Even a casual look at container classes will show that most of them have features bearing these names, or many of them.

This is a deliberate choice. One could of course invent new names for each class, reflecting the specific properties of the corresponding kind of container. But these peculiarities are already captured by the signature, header comments and contracts of the features, for example in *put* for *ARRAY*

```
put (v: like item; i: INTEGER)
        -- Replace i-th entry, if in index interval, by v.
    require
        valid_key: valid_index (i)
    ensure
        replaced: item (i) = v
```

→ valid_index appears in the discussion of arrays, page 382.

and in *put* for *STACK*:

```
put (v: G)
        -- Push v onto top.
    require
        extendible: extendible
    ensure
        pushed: item = v
```

so that no confusion can result. Using consistent terminology facilitates using the library and — for novices — learning to use it: when discovering a new class, readers can quickly identify the key features and their purpose.

Touch of Methodology: Standard Feature Name Principle

Use the standard names, when applicable, for features of your own classes, to enhance their consistency and readability.

Automatic resizing

We saw above that creation procedures (usually *make*) that specify an initial size always mean it as an indication, not a permanent limit. The data structures of EiffelBase (Eiffel's container library) are almost all either unbounded or, if they have an initial bound, resizable. Part of what defines a good programmer is, indeed, avoidance of absolute limits.

Do not let anyone lock you — and the users of your programs — in a fixed box. Computers have large memories. Design your data structures, whenever possible, so that if the size of the data exceeds expectations they do not give up but just reallocate themselves with a larger size. Not applying this advice exposes users of your programs to one of the most frustrating run-time situations: having to stop because execution has reached a size limit, even though there is plenty of memory left.

Even our arrays will be resizable.

It so happened that during the writing of this chapter the world was fixated on the initially unsuccessful exploration of Mars by the NASA's Spirit and Opportunity rovers. Spirit was silent for more than a day, rebooting again and again. Engineers suspected all kinds of possible equipment failures, until it surfaced that it was a software issue: the system had room for a fixed number of file handles, and needed more files than planned.

See www.newscientist.com/news/news.jsp?id=ns99994610.

One year later, a few weeks after the US elections, it transpired the vote-counting software in the San Francisco Supervisor vote had failed because of "a *hard-coded constant maximum number of voters that was set too low*".

Peter G. Neumann, "Some 2004 voting anomalies", www.risks.org/23.59.html#subj2.

Do not fall into such pitfalls:

Touch of Methodology: Don't box in your users

Do not use constant built-in limits. Let your data structures resize themselves to adapt to the size of each instance of the problem.

We will see that for some of the container variants, especially arrays, resizing is an expensive operation, time-wise. So you should always use it carefully. But efficiency concerns are never an excuse for using fixed-size structures. Such structures in fact damage efficiency on the space side, since they encourage programs to over-allocate, just in case. Better allocate what you think you will normally need, and resize dynamically if necessary.

→ *"Resizing an array", page 386.*

13.3 ESTIMATING ALGORITHM COMPLEXITY

The last comment brings to the forefront the issue of *efficiency*, or *performance*, which involves both *execution time* and *storage space*. The main reason for using different kinds of container structure is that they exhibit different time and space performance behaviors for carrying out the essential operations just reviewed.

We need a reliable way to contrast performance between various data structure choices. It is not enough to measure concrete performance on specific examples and report that "*on average, calls to the item query took 10 nanoseconds for arrays and 40 nanoseconds for linked lists*":

- To talk about averages we must have a significant statistical distribution; there is no clear way of determining such a distribution for container sizes (how many 10-item containers, how many with 1000 items etc.).
- You cannot easily infer from the measurements how the results will scale up. Some techniques can be very good for small structures, but what matters in performance-critical applications is how well they do for large sizes. (For 10,000 items, almost any container will do a decent job.)
- The result is closely tied to the context of the measurements: machine, operating system, even programming language and compiler. The same experiment may give radically different outcomes in different setups.

The main accepted measure of algorithm complexity provides an estimate liberated from such contingencies. It is known as **abstract complexity**; also as *asymptotic* complexity, and familiarly as "Big-O notation", sometimes written "Big-Oh" to emphasize that the O is a letter.

Measuring orders of magnitude

Abstract complexity relies on two principles:

- Provide the measure as a function of the *size* of the data structures under consideration. For most of the examples in this chapter it is a single parameter: *count*, the number of items in a container.
- Define the function not by an exact formula but by an *order of magnitude*, the O in "Big-O", as in **O** (*count*) (pronounced "**O** of *count*").

When we say that the time for a search operation in a list of *count* elements is **O** (*count*) we mean that for large values of *count* it grows at most **proportionally to** *count*. Another operation may be **O** ($count^2$), meaning that its execution time grows at most proportionally to the square of the number of elements. The same conventions are used for estimating space requirements.

In such a measure:

- Constant multiplicative factors do not matter: $\mathbf{O}\ (100 * count^2)$ means the same as $\mathbf{O}\ (count^2)$. The justification for this convention is that we should not attach long-term importance to multiplication by any constant, since the same algorithm implementation may become 100 times faster or slower just by being moved to a different machine; but how its computation time varies when *count* grows does not depend on such technical choices.
- Constant additive factors also do not matter: $\mathbf{O}\ (count^2 + 10000)$ means the same as $\mathbf{O}\ (count^2)$. The constant may have a strong influence for small *count*, but as *count* grows it fades away.
- Similarly, any additive factor with a smaller exponent does not matter: $\mathbf{O}\ (count^3 + count^2)$ is the same as $\mathbf{O}\ (count^3)$.

As a consequence, to express that an algorithm takes constant time — or more realistically, since several executions are unlikely to take exactly the same time, that its execution time is *bounded* by a constant on any particular platform — we say that it is $\mathbf{O}\ (1)$. We might just as well say $\mathbf{O}\ (37)$ or $\mathbf{O}\ (1000)$, but 1 is the convention.

Mathematical basis

The Big-O notation may seem informal, but it is possible to define it in a rigorous way, as a relation between two functions:

Definition: Big-O notation for abstract complexity

Let f and g be two *functions* from natural numbers to positive real numbers. Function f is said to be $\mathbf{O}\ (g)$ — or, more commonly, $f\ (n)$ to be $\mathbf{O}\ (g\ (n))$, spelling out the argument — if there exists a constant K such that $f\ (n)\ /\ g\ (n) < K$ for every natural number n.

An *algorithm* is $\mathbf{O}\ (g\ (n))$ in time or in space if the function giving its execution time or space occupation in terms of the input size n is $\mathbf{O}\ (g\ (n))$.

When reading analyses of algorithm complexity you may encounter statements such as "$f\ (n) = g\ (n) + \mathbf{O}\ (n^2)$" as an abbreviation for "$f\ (n) = g\ (n) + s\ (n)$ for some function s, where $s\ (n)$ is $\mathbf{O}\ (n^2)$". The intent is to state that f is "like" g except for a term in $\mathbf{O}\ (n^2)$.

As a consequence of the definition, if a function is $\mathbf{O}\ (n^2)$ it is also $\mathbf{O}\ (n^3)$, $\mathbf{O}\ (n^4)$ and so on. This is because a Big-O specification gives an upper bound, not a tight estimate. Useful statements of complexity will use the best (lowest) known estimate, as in "We know this algorithm is $\mathbf{O}\ (n^{2.5})$, can we show it is $\mathbf{O}\ (n^2)$?".

> To state that a function g asymptotically provides a *lower* bound as well as an upper bound, within multiplication by constant factors, algorithm analysis has the "Big-Theta" notation $\Theta\ (g\ (n))$. For simplicity we will just use $\mathbf{O}$, with the understanding that the given g functions are the best known at any stage of the discussion, and that they characterize worst-case behavior unless otherwise noted.

Logarithms frequently arise in the analysis of algorithm complexity. For example the best algorithms for *sorting* a list of n values are $\mathbf{O}\ (n * (\log n))$. Such a formula does not specify the logarithm base (such as 2 or 10), because a change of base only contributes a multiplicative constant, per the formula $\log_b n = \log_b a * \log_a n$.

Making the best use of your lottery winnings

This convention of ignoring multiplicative constants can be surprising at first. If an algorithm takes $count^2$ nanoseconds, abstract complexity considers it less good than one taking $10^6 * count$ nanoseconds, even though it runs faster for up to one million items. What the convention gives us is an understanding of the essential behavior of algorithms as a function of the growth of the problem size.

The following observation helps understand the benefit. Consider four algorithms with performance such that the biggest problem size they can tackle in 24 hours of continuous operation on your computer is respectively N_1. N_2. N_3. N_4. Their abstract complexities are $\mathbf{O}\ (n)$, $\mathbf{O}\ (n \log n)$, $\mathbf{O}\ (n^2)$, $\mathbf{O}\ (2^n)$. You win at the lottery and have the opportunity to buy a computer one thousand times faster than your current one. What does this get you?

Adapted from Aho, Hopcroft, Ullman; see "Further reading", page 432.

- With an $\mathbf{O}\ (n)$ algorithm you can now solve a problem that is also a thousand times bigger: $1000 * N_1$.
- With $\mathbf{O}\ (n \log n)$ the improvement is still multiplicative, with a factor that is close to 1000 for large N_2.
- With $\mathbf{O}\ (n^2)$ you multiply the maximum problem size by a factor of about 32 (square root of 1000).
- With $\mathbf{O}\ (2^n)$ your new dream machine *increases* N_4 — an addition, not a multiplication! — by just 10.

This question — how big a problem you can solve in a given time, rather than how much time it will take to solve a problem of a given size — is often the right way of looking at efficiency issues.

> Consider for example a next-day weather forecast program. (Meteorology has made spectacular progress in the past two decades thanks to computer modeling.) The program works from past data collected at a number of points on a geographical grid. More grid points means more accurate predictions. To assess the program's efficiency, the useful criterion is not how long it takes to process a fixed number of grid points, since an outstanding next-day forecast will not help if it takes 48 hours to complete. It is the reverse question: how many data points you can process in a fixed time, for example one hour.

This reasoning illustrates what abstract complexity gives us: a view of algorithm efficiency free from superficial technology considerations, but helping to understand the benefits of potential technology improvements.

Abstract complexity in practice

When measuring the Big-O complexity of an algorithm you may be interested in any of three variants, and should clarify which one you report:

- **Average complexity**, assessing the average time or space taken up by the algorithm. As noted this is only meaningful if we have a probability distribution on the algorithm's input; usually the distribution considers all possible inputs equally likely.
- **Maximum complexity**, also called **worst-case** complexity, assessing the time or space required by the inputs that make this measure highest.
- **Minimum** or **best-case** complexity, less often useful in practice (other than for programmers who believe in the Tooth Fairy), but sometimes interesting for purposes of comparison.

Presenting data structures

In the remainder of this chapter we look at fundamental structures. The presentation relies on the **EiffelBase** library of data structures and algorithms, which provides reusable classes for all the concepts under study: *ARRAY*, *LINKED_LIST*, *HASH_TABLE*, *STACK* and so on.

The description takes the viewpoint of the *client programmer* (also known as "you"): someone who will take advantage of these library classes to write a new application that uses arrays, linked lists etc. Features will, as a result, be introduced through their **contract forms**.

The presentation explains basic implementation techniques but does not, as a rule, show feature implementations. You *can* see implementations if you wish: EiffelBase is open source software, included with any delivery of EiffelStudio, and you are welcome to explore its code, written with the explicit goal of serving as a model of O-O style, and refined over the years.

Not to imply it's perfect …

13.4 ARRAYS

We start with one of the most ubiquitous kinds of container, arrays.

Arrays are a software notion, but their importance comes from a hardware property: the addressing mode of the type of main memory used in today’s computers, known as *Random Access Memory*, or just “Random Memory”. In spite of the name this does not mean that the computer throws a die to decide which cell to access (interesting idea, though) but that the time to access a memory cell — either to read it or to modify it — does not depend on the cell’s address. (Understand “random” as in “*you can pick an address at random and not worry about the effect on access time*”.) If you have a 2 GB memory, it will not make any difference whether the cell is the first (address 0), the last (address $2^{31}-1$) or anywhere in-between.

← *“Transient memory”, page 284.*

Random access memory stands in contrast to *sequential access* memory, where you access an item by first traversing a set of preceding elements. Magnetic tapes are a typical example: the tape head reaches a particular position by rolling the tape to that position. You may also think of analogies in non-computer devices:

Sequential and random access

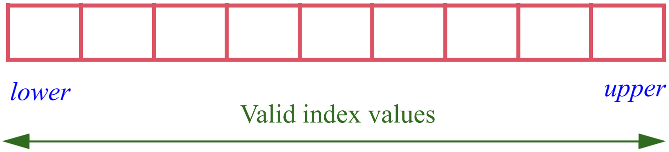
(Sequential)

(Random)

The scroll on the left will be read and written sequentially. You may access any of the mailboxes on the right directly, without going through the others first.

Arrays take advantage of the random access property by letting you define and manipulate structures made of a number of items stored in contiguous memory locations, and each identified by an index:

An array



Bounds and indexes

An array has a lower bound and an upper bound, given in the class *ARRAY* [*G*] by the queries

```
lower: INTEGER
          -- Minimum index.
upper: INTEGER
          -- Maximum index.
```

The invariant of the class states that *count*, the number of items (also accessible as *capacity*), is *upper* – *lower* + 1. Since *count* ≥ 0, we must have

```
lower <= upper +1
```

The case *lower* = *upper* corresponds to an array with one item; *lower* = *upper* +1 corresponds to an empty array (you can visualize it, in the last figure, as *lower* moving right and *upper* moving left until they cross). This is a legitimate state for an array:

Touch of Methodology: Extreme Cases Principle

When designing object structures, for example containers, consider extreme cases — empty structure, "full" structure if there is a maximum capacity — and make sure that the definition still makes sense for them.

There is a long history of bugs resulting from inadequate handling of extreme cases. People will think of (and test for) cases in which an array or other structure has items; then, in an execution for some particular input data, the container happens to be empty, and everything blows up. Following the above advice avoids such nasty problems.

The class invariant is your primary guide to checking that the definition "still makes sense". Here the case *lower* = *upper* +1 remains compatible with the invariant clause *lower* <= *upper* +1; it yields the smallest value of *upper* – *lower* +1 (that is to say, *count*) that still satisfies this requirement.

To access and modify array items, you must use integer indexes. A query is available to find out if an integer is a meaningful index:

```
valid_index (i: INTEGER): BOOLEAN
        -- Is i within array bounds?
    ensure
        Result implies ((i >= lower) and (i <= upper))
```

Creating an array

To create an array, you provide the desired lower and upper bounds:

```
your_array: ARRAY [SOME_TYPE]
…
create your_array.make (your_lower_bound, your_upper_bound)
```

using the creation procedure

```
make (min_index, max_index: INTEGER)
        -- Allocate array; set index interval to min_index .. max_index;
        -- set all values to default.
        -- (Make array empty if min_index = max_index + 1).
    require
        valid_bounds: min_index <= max_index + 1
    ensure
        lower_set: lower = min_index
        upper_set: upper = max_index
        items_set: all_default
```

As the first two postcondition clauses indicate, the procedure sets *lower* and *upper* to the given values, *your_lower_bound* and *your_upper_bound* in the example. These are arbitrary expressions; you can use constants, as in

```
create yearly_twentieth_century_revenue.make (1901, 2000)
```

where the bounds are set in the program text; but you may also use variables and more general expressions, as in

```
create another_array.make (m, m + n)
```

In examples such as these the index interval has a meaning of its own, such as directly representing the years of the 20th century. If you just want a sequence of *n* values that can start anywhere, the common convention is to use the bounds 1 and *n*:

```
create simple_array.make (1, n)
```

> The C language and its successors (C++, Java, C#) require all arrays to start their indexes at 0. In examples such as "years of the 20th century" this means that you will have to perform back-and-forth translations (here adding or subtracting 1901) between the physical index and its intended meaning. For cases such as *simple_array*, the choice of 0 or 1 as starting index is partly a matter of taste. If you are like me you think of your thumb as the first finger on your hand, not the zeroth, and of your middle finger as the third, not the second. A less subjective reason is that with the 0 convention the last item of an array of size *n* has index *n*–1, a source of errors.

The query *all_default*, in the last postcondition clause of *make*, expresses that all items of an array of type *ARRAY* [*SOME_TYPE*] will, on creation, be set to the default value for *SOME_TYPE*: zero for *INTEGER* and *REAL*, false for booleans, void reference for any reference type.

Accessing and modifying array items

The basic query and command to obtain and modify an array item are:

```
item (i: INTEGER): G
        -- Entry at index i, if in index interval.
    require
        valid_key: valid_index (i)

put (v: like item; i: INTEGER)
        -- Replace i-th entry, if in index interval, by v.
    require
        valid_key: valid_index (i)
    ensure
         inserted: item (i) = v
```

→ The declaration of item also contains **alias** *and* **assign** *clauses; see "Bracket notation and assigner commands", page 384 below.*

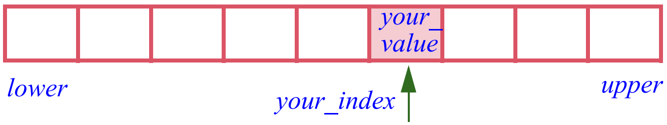
Note the precondition, requiring in both cases the index to be within bounds.

← valid_index appeared on page 382.

A typical use is, with *your_array*: *ARRAY* [*SOME_TYPE*] properly created and *your_value*: *SOME_TYPE*:

```
your_array.put (your_value, your_index)
```

which sets the corresponding array item:



Updating an array element

overwriting any value previously entered there (including the default after initialization). Note the order of arguments: first the value to be written, then the index at which to write it.

After this call to *put*, the instruction

```
your_value := your_array.item (your_index)
```

will assign to *your_value* the item found at *your_index* in the array.

The postcondition of *put* as shown on the previous page expresses that immediately after a *put* the *item* value at the given index is the given value.

The examples for both *put* and *item* are only correct if the chosen *your_index* is within bounds. If this not guaranteed, then you should use

```
if your_array.valid_index (your_index) then
    your_array.put (your_value, your_index)
else
    …
end
```

and similarly for *item*.

In any reasonable implementation of arrays the cost of a call to *put* or *item* is **O** (1) — constant time. This is the RAM property, the basic reason for using arrays.

Bracket notation and assigner commands

The following notations, using brackets, are available for class *ARRAY* and a few others in this chapter:

```
your_value := your_array [your_index]
        -- An abbreviation for your_value := your_array.item (your_index).
your_array [your_index] := your_value
        -- An abbreviation for your_array.put (your_value, your_index).
```

This is particularly convenient for such instructions as

```
a [i] := a [i] + 1                                                    [3]
```

more readable than *a.put* (*a.item* (*i*) + 1, *i*). The bracket notation follows mathematical practice; its advantage is clear for expressions that involve several array elements and mathematical operators.

There is nothing magical about the bracket notation, and it is not specific to arrays. To make it applicable to any type for which it makes sense, include a **alias** "[ ]" mark next to the name of the corresponding feature in its declaration. This is what class *ARRAY* does for *item*:

```
item alias "[]" (i: INTEGER): G assign put
        -- Entry at index i, if in index interval
    require
        valid_key: valid_index (i)
    do
        … Implementation of the feature …
    end
```

Adding **alias** "[ ]" to the feature name indicates that the brackets are an “alias” for the feature name: another way to call it. As a result, the notation

```
your_array [i]
```

is simply a synonym (an alias) for

```
your_array.item (i)
```

The declaration of *item* also specifies **assign** *put*. You may use such a clause for any query *q* — whether or not it also has a bracket alias — by marking it **assign** *c* where *c* is a command of the same class. The effect (using the present example in which *q* is *item* and *c* is *put*) is to make the following assignment-like notation valid:

← *As previewed in “Information hiding: modifying fields”, page 240.*

```
your_array.item (i) := your_value                                    [4]
```

merely as an abbreviation for a call

```
your_array.put (your_value, i)                                       [5]
```

to the command *put* which the **assign** clause has associated with *item*; such a command is called an *assigner command*.

> Terminology: a command whose principal role is to set the value of a query was called a *setter command*. A setter command becomes an “assigner command” through the language mechanism that explicitly associates it with the query.

← *“Setters and getters”, page 248.*

Because *item* now has both a bracket alias and an assigner command, it is also legitimate to use the bracket form as another synonym for the last call

```
your_array [i] := your_value                                        [6]
```

which achieves full reconciliation with traditional mathematical notation for arrays, vectors etc., while using the semantics of object-oriented operations. This is what made [3] legal.

The assigner command mechanism is applicable to any query, including attributes. The resulting instructions, such as [4] and [6], are not assignments: assigning to a field would, as you know, violate information hiding. They are plain procedure calls, equivalent to [5] and observing object-oriented principles; they simply use an assignment-like syntax for convenience.

← *Information hiding: modifying fields*, page 242.

A language note: most programming languages, from Pascal, C and C++ to Java and C#, offer such bracket notation for arrays, for both access (*your_array* [*i*]) and modification (*your_array* [*i*] := *your_value*). In most cases the notation is specific to arrays, and arrays themselves are a special built-in notion. Eiffel treats *ARRAY* as a normal class with features *item* and *put*, for consistency with other data structures and the object-oriented approach (allowing, for example, a class to inherit from *ARRAY*). The language offers bracket notation as a synonym, through the **alias** "[]" construct. This construct is general and not limited to arrays: it is available in other structures studied later in this chapter, such as hash tables and linked lists, and you can apply it to any class that you write.

Resizing an array

At any time during execution, an arrays has fixed *lower* and *upper* bounds, and hence a fixed number (*count*) of items. The precondition *valid_index* of *put* and *item* reflects this property. In many languages these properties are set once and for all, either statically (using constants bounds) or on creation. In Eiffel you can resize an array through *resize*:

```
resize (min_index, max_index: INTEGER)
        -- Rearrange array to accommodate indexes down to min_index
        -- and up to max_index. Preserve existing items.
    require
        good_indexes: min_index <= max_index
    ensure
        no_low_lost: lower  = min_index.min (old lower)
        no_high_lost: upper = max_index.max (old upper)
```

Feature *min* and *max*, used in the postcondition, are functions on *INTEGER*, giving the minimum and maximum of the current number and the argument.

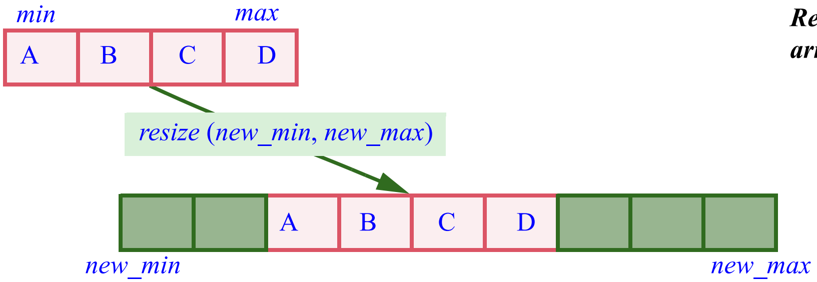
Resizing is often indirect, through the procedure *force*. To change the value of an item, the default mechanism is *put* (*v*, *i*), with the precondition that we have seen: *valid_index* (*i*). This is usually the right approach; but it assumes that you know in advance how many items you will need. If you miscalculate, the algorithm will fail. Using *force* works in such cases:

← *Page 383*.

```
force (v: like item; i: INTEGER)
        -- Replace i-th entry, if in index interval, by v.
        -- Always applicable: resize the array if i falls out of current
        -- bounds; preserve existing items.
    ensure
        inserted: item (i) = v
        higher_count: count >= old count
```

Unlike *put*, procedure *force* has no precondition and so is always applicable. If *i* falls outside of the interval *lower*..*upper*, the procedure will call *resize* to accommodate the requested entry.

Because of the continuous-memory implementation of arrays, resizing usually requires reallocating the array to a new memory area and copying the old values:



Reallocating an array to resize it

Reallocation and copying are expensive, **O** (*count*) operations. As a result, *force* itself is **O** (*count*), to be compared to the very fast, **O** (1) cost of a standard *put*. Obviously, you should use *force* with care. Note that its implementation is prudent: if it has to call *resize*, it will make sure that the new size is sufficiently bigger than the previous one, so that for example a call

```
your_array.force (some_value, your_array.count + 1)
```

increases the size by more than one; the default policy is a 50% increase. So if you repeatedly use *force* in this style to extend an array at either end, only a few of the *force* operations will cause a *resize*.

Using arrays

An array of type *ARRAY* [*G*] represents a total function from the integer interval *lower*.. *upper* to *G*. If after creation the bounds *lower* and *upper* do not change, or change only rarely, the implementation is highly efficient, since every access to the function, or modification of the function's value for a given index in the interval, is **O** (1) and very fast. This makes arrays well suited to situations where:

- You need to handle a set of values associated with an integer interval.
- Possibly after an initial period of allocating the array and filling up its initial values, the dominant operations are limited to index-based access and modification.

Because of the high cost of reallocation, arrays are not appropriate for highly dynamic data structures where elements come and go. In particular, inserting or deleting an item is expensive (**O** (*count*)) if this implies renumbering the indexes, and hence shifting all the elements to the right or left of the insertion or deletion position. For such behavior, you should use other structures studied later in this chapter.

Performance of array operations

Here is a summary of the cost of array operations.

Operation	Features in class *ARRAY*	Complexity	Comments
Index-based access	*item* **alias** "[]"	**O** (1)	
Index-based replacement	*put* **alias** "[]"	**O** (1)	
Index-based replacement outside of current bounds	*force*	**O** (*count*)	Requires reallocating the array. (Only a fraction of successive *force* operations, will, however, cause such reallocation.)
New item insertion		**O** (*count*)	Shifting indexes; not a common operation.
Removal		**O** (*count*)	Can be done in **O** (*count*) by shifting indexes; not a common operation.

13.5 TUPLES

Arrays are homogeneous: in an instance of *ARRAY* [*T*], all items are of type *T*, or a type compatible with *T*. **Tuples** are similar to arrays, but may hold values of several specified types rather than just one. If you declare

```
tup: TUPLE [number: INTEGER; street: STRING; resident: PERSON]
```

the possible values for *tup* at run time are sequences of three or more components of which the first is of type *INTEGER*, the second of type *STRING* and the third of type *PERSON*, assumed to be an existing class. Such tuples could be useful, for example, in a census application, each of them recording the observation that at a certain *number* in a certain *street* lives a certain *resident*.

To denote a tuple value it suffices to write the successive components in brackets with commas in-between, yielding an expression, or *manifest tuple*, which you can use as argument to a routine call or assign to a tuple variable such as *tup*:

```
tup := [99, "Rue de Rivoli", Louvre_museum_curator]                    [7]
```

> The term "tuple" comes from mathematics: after the *pair* — two values, whose order matter — and the *triple* there's the *quadruple*, the *quintuple* and (unless the term gets blocked by parental-control filters) the *sextuple*, so it was natural for mathematicians to start talking about "*n-tuples*" for any *n*, denoting ordered sequences of *n* values.

Tuple types are not particularly exciting as a data structure — exciting in the way arrays, lists, hash tables, binary search trees and others each bring an original way to store and retrieve data, with its own efficiency advantages and limitations. In fact a straightforward implementation of tuples is through arrays. (Ignoring the specific type information we may look at a tuple as an *ARRAY* [*ANY*] where *ANY* is the general high-level type covering all possible types.) Then for the complexity of tuple operations we can use what we just found for arrays:

→ *"Overall inheritance structure", 16.10, page 586.*

Operation	Tuple notation	Complexity	Comments
Component access	*t.comp*	**O** (1)	See below about the notations
Component replacement	*t.comp* := *value*	**O** (1)	
Insertion, Removal		Not applicable	

The interest of tuple types lies elsewhere: as a language mechanism allowing you to describe simple structures in an clear and simple way, without resorting to classes. So far the tags — *number*, *street*, *resident* — played no role in that mechanism; manifest tuples, as in [7], did not use them. Tags are useful to access and set individual components of an existing (non-void) tuple; after [7], for example, *tup.number* will have the value 99. You can also use them to set component values; they are treated like attributes with associated “assigner commands”, enabling you to write such instructions as

← *“Bracket notation and assigner commands”, page 384.*

```
tup.resident := some_person
```

All this suggests that we could do without tuple types by using classes such as

```
class CENSUS_RECORD feature
    number: INTEGER assign set_number
    street: STRING assign set_street
    resident: PERSON assign set_resident
    set_number (n: INTEGER) do number := n ensure number = n end
    … set_street, set_resident like set_number …
end
```

which allows the same operations on *cr* of type *CENSUS_RECORD* as on *tup* above: to access fields, *cr.number* etc.; to set fields, *cr.resident* := *some_person* etc.

Tuples are useful when this is all that you need from a class: a set of attributes, all public; and for each of these attributes, a setter with no precondition, assigning the argument’s value to the attribute and doing nothing else. Such a class describes plain records (composite values, as used for example in *relational databases*). Using tuples in such cases saves the need to write simple classes such as *CENSUS_RECORD*. For that reason, tuples are also called *anonymous classes*. As soon as you need anything more sophisticated, they will no longer do the job: you should declare a class (and give it a name).

To finish with the language mechanism, note that tags do not affect the tuple’s type; in fact they are optional. So you can also write the above type as *TUPLE* [*a*: *INTEGER*; *b*: *STRING*; *x*: *PERSON*], or just *TUPLE* [*INTEGER*, *STRING*, *PERSON*] if you do not need to access or set components by name.

Syntactically, such tuple types look like generically derived class types, such as *LIST* [*T*]; indeed the concepts are similar, but there is no class *TUPLE* because it would have to admit an arbitrary number of parameters, whereas a generic class always takes a fixed number (one parameter in *ARRAY* [*G*] and *LIST* [*G*], two in *HASH_TABLE* [*G*, *KEY*]). With tuple types you can describe sequences of any length: *TUPLE* with no parameters covers all sequences, *TUPLE* [*T*] sequences of at least one element with the first of type *T*, and so on.

This observation also determines *conformance* properties: you may assign an expression of type *TUPLE* [*T*, *U*, *V*] to a variable of the same type or of any of the following types: *TUPLE* [*T*, *U*]; *TUPLE* [*T*]; plain *TUPLE*. The last of these (not a class, as noted, but a type) covers all possible tuples.

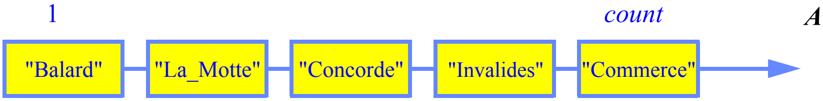
→ *A precise definition of conformance will appear in “Definition: conformance”, page 564.*

We will find tuple types particularly useful in connection with *agents*, covering applications such as iteration and event-driven programming.

→ *Chapter 18.*

13.6 LISTS

A *list*, also known as a *sequence*, is a container keeping elements in a certain order, usually the order of insertion. Mathematically, it represents a total function from the interval 1 .. *count* to *G*; this seems similar to arrays, but the big difference is that *count* can vary freely as you insert new elements.



A list

The figure illustrates a list, made of five items. The arrow is there simply to highlight that order matters. The same elements organized in a different order would make up a different list.

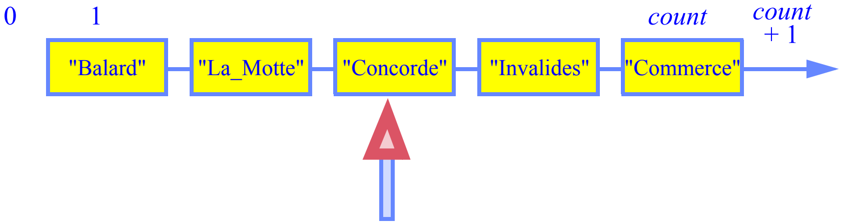
> Like for arrays and other structures where elements are numbered, we systematically start the numbering at 1.

Various implementations of lists are possible, provided by such EiffelBase classes as *LINKED_LIST*, *TWO_WAY_LIST*, *ARRAYED_LIST*, *MULTI_ARRAYED_LIST*. The present section describes properties common to all these variants and covered by class *LIST*; we will then look at the important case of linked lists, and survey the others briefly. As in the rest of this chapter the implementations will not be described in detail, but we review the essential ideas and significant implementation examples. For the full picture you can read the class texts from EiffelBase.

> *LIST*, as befits a class describing a general, abstract notion, is a *deferred* class, of which the others *inherit* directly or indirectly. This means in particular that they do not repeat common elements, but move them to the class at the highest level of generality in each case. These concepts are the topic of the chapter on inheritance.

→ *Chapter 16.*

The list classes treat a list not just as a collection of elements but as a *machine* which at any point in its existence has a *state* characterized by a *cursor*:



A list with its cursor

This notion is not new; when we manipulated a metro line as a list of stations we already had a cursor.

← *"Animating a metro line", page 166.*

Having a cursor facilitates the basic list operations — accessing, inserting or deleting an item — by letting the corresponding routines use a simple interface: instead of asking you to specify a position, they work at the cursor position. " Delete", for example, means "delete the item at cursor position". You can still perform an operation at any position you like: just move the cursor there first.

In this scheme the cursor "internal": each list has its own cursor. It is also possible to use *external* cursor objects, allowing different clients to retain separate views of where they are in a list. This is useful in particular in concurrent applications. You can look up the EiffelBase class *CURSOR* and its descendants for more details on this approach. A typical use of an external cursor, illustrated in a later chapter, is to record the initial position of the *internal* cursor and restore it after an operation requiring a traversal.

→ The iterator routine do_all in "Writing an iterator", page 631 uses this scheme.

Cursor queries

We should allow the cursor, as suggested by the last figure, to range not only from 1 to *count* — positions that may hold items — but from 0 to *count* + 1: it may fall off to the left of the first item or to the right of the last item. You will quickly see the usefulness of this convention. We can express it formally: with the query

```
index: INTEGER
        -- Current cursor position.
```

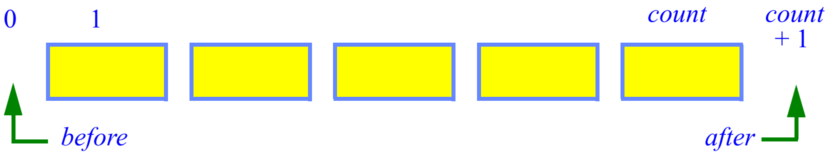
we have the invariant clauses

```
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

To characterize these extreme cases we have two queries:

```
before: BOOLEAN
        -- Is there no valid cursor position to the left of cursor?

after: BOOLEAN
        -- Is there no valid cursor position to the right of cursor?
```

Note the careful phrasing of the comments, justified by the need to cover all cases including that of an *empty* list as we will see next.



"Before" and "after" cursor positions

According to the style standard for boolean-valued queries, the above two queries should be called *is_before* and *is_after* (as in the Traffic classes representing lines, used in previous chapters). The names *before* and *after* have been around for so long that no one wants to change them. Other names used below, such as *is_empty*, follow the normal *is_something* convention.

If in the current state of a list the cursor is either "before" or "after" we say that it is "off":

```
off: BOOLEAN
        -- Is there no current item?
    ensure
        definition: Result = (after or before)
```

Further invariant clauses express the properties of these queries (postconditions are also possible):

```
before_definition: before = (index = 0)
after_definition: after = (index = count + 1)
off_definition: off = (index = 0 or index = count + 1)
```

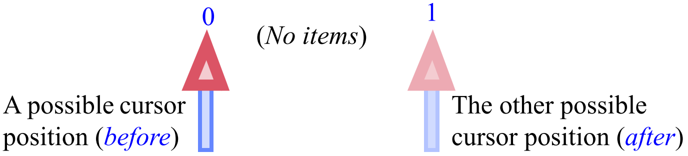
Other queries about the cursor position include:

```
is_first: BOOLEAN
        -- Is cursor on first item?
    ensure
        valid_position: Result implies (not is_empty)
is_last: BOOLEAN
        -- Is cursor on last item?
    ensure
        valid_position: Result implies (not is_empty)
```

themselves relying on

```
is_empty
        -- Are there no items?
```

A list can indeed be empty, in which case *is_first* and *is_last* always yield false as implied by the relevant invariant clauses: the cursor can only be on the first item if there is at least one item. Do not forget the Extreme Cases Principle: it is ← *Page 381.* essential to make sure that our conventions still work well in such border cases. In the absence of items, the figure illustrating the list becomes:


0
(No items)
1
A possible cursor position (before)
The other possible cursor position (after)

An empty list and its two possible cursor positions

In this case *count* is zero and the maximum *index* position satisfying the invariant, *count* + 1, is one. In such an empty list the cursor can be in position 0 or position 1. In either case *off* will hold, hence the invariant clause

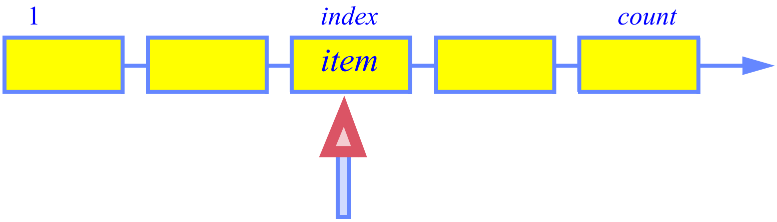
```
empty_constraint: is_empty implies off
```

Note the repeated accumulation of invariant clauses to express, little by little, what we understand of our own object structures:

Touch of Methodology: Using invariants

Use invariant clauses to make explicit the consistency properties of the classes you design, and to check (in particular by considering extreme cases, in line with the Extreme Cases Principle) that these properties are logically sound and compatible with each other.

To access the list item at cursor position



Current item

you will use

```
item: G
        -- Item at cursor position.
    require
        not_off: not off
```

This query returns a result of type *G*, the generic parameter of the list classes (*LIST* [*G*], *LINKED_LIST* [*G*] etc.). Note the precondition: in an *off* state — including for an empty list — there is no current item.

Cursor movement

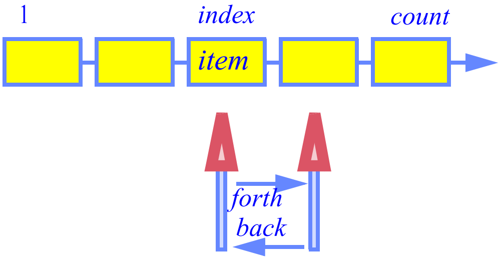
You have a number of commands at your disposal to move the cursor around. The following will bring the cursor to the beginning or end of a list:

```
start
        -- Move cursor to first position (no effect if empty).
    ensure
        at_first: (not is_empty) implies is_first
finish
        -- Move cursor to last position (no effect if empty).
    ensure
        at_last: (not is_empty) implies is_last
```

A call to *start* ensures *is_first*, and a call to *finish* ensures *is_last*, but only, for reasons just discussed, for non-empty lists. The postconditions express this.

You can also move the cursor by one position:

```
forth
        -- Move cursor to next position.
    require
        not_after: not after
    ensure
        moved_forth: index = old index + 1
back
        -- Move cursor to previous position.
    require
        not_before: not before
    ensure
        moved_back: index = old index – 1
```



The preconditions guarantee that the index remains within bounds as specified by the earlier invariant clauses *non_negative_index* and *index_small_enough*. You may also move the cursor to a specified position:

← *Page 392.*

```
go_i_th (i: INTEGER)
        -- Move cursor to i-th position.
    require
        valid_cursor_position: i >= 0 and i <= count + 1
    ensure
        position_expected: index = i
```

Iterating over a list

One of the most common manipulations on a list is to apply a given operation to every item in turn. Assume this operation is given by a routine

```
your_operation (x: G)
```

We have already seen, when dealing with metro lines and their stations, the scheme for applying *your_operation* to every list item; the general form is the loop

```
from
    your_list.start
until
    your_list.after
loop
    your_operation (your_list.item)
    your_list.forth
variant
    your_list.count – your_list.index + 1
end
```

This is for applying an operation to some existing list *your_list* from your program. The scheme also appears within the list classes themselves to perform traversals of the *current* list, using unqualified calls *start*, *after*, *forth* without "*your_list*." (that is to say, without a call target and a period). We will see examples shortly, with the routines *search* and *has* which search for a value among the items of a list.

← "Definitions: Qualified and unqualified call", page 134.

There are other forms, for example to apply a certain operation to all elements of a list *up to* and excluding the first that satisfies a certain condition:

```
from
    your_list.start
until
    your_list.after or else your_condition (your_list.item)
loop
    your_operation (your_list.item)
    your_list.forth
variant
    your_list.count – your_list.index + 1
end
```

Such a scheme is an example of *iterating* on a data structure:

Definition: Iterating

To **iterate** on an object structure is to apply a given operation to all items of the structure, or to all items satisfying a given condition.

Another term for "iterating" is *traversal*. An *iteration* is the application of an iterating mechanism to a structure, although we have also encountered the term in the sense of one step in the process ("on every iteration of the loop, the cursor moves by one position"). An *iterator* is the mechanism that transforms an operation on an individual item into an operation on all items of a structure.

→ *"Traversals", page 453.*

→ *"Definition: Iterator", page 431.*

An example of implementation using an iteration mechanism, shared by all the list classes, is the procedure *search* for finding an element in a list. Its text looks like this:

```
search (v: G)
        -- Move cursor to first position, at or after current position,
        -- where item value is v; if none, go to after position.
    do
        from
            if before and not is_empty then
                forth
            end
        until
            after or else item = v
        loop
            forth
        end
    end
```

This version compares *v* and *item* through basic equality =; it is also possible to use object equality, ~.

This feature is a command, which brings the cursor:

- If the sought value *v* occurs at the current position or any one to its right, to the first such position.
- Otherwise, to the extreme right (*after*).

This interface convention allows you to use *search* repeatedly to search for successive occurrences of a value. The procedure is also used in the implementation of *has*, the query to find out whether a value appears at all:

```
has (v: G)
        -- Does structure include an occurrence of v?
    local
        original_index: INTEGER
    do
        original_index := index
        start
        search (v)
        Result := not after
        go_i_th (original_index)
    end
```

As a query, *has* should leave the object structure in the state where it found it; it uses a local variable *original_index* to record the initial cursor position and return to it, through *go_i_th*, at the end.

Both *search* and *has* require **O** (*count*) time on maximum and average.

The iteration scheme illustrated by *search* recurs throughout the use of lists and other sequential structures; we have already encountered several examples, starting with the loop that illuminated the total travel time on a metro line.

← *Page 154.*

At the end of this chapter we will come back to the concept of iteration and take a first look at general mechanisms that enable us to use standard iteration mechanisms rather than re-implement in every case an explicit loop with *start*, *forth*, *item* and *after*.

→ *"Iterating on data structures", 13.13, page 431.*

Adding and removing items

To add an item to a list — at the beginning, the cursor position, or the end — you may use one of the operations with the following specifications:

```
put_front (v: G)
        -- Add v to beginning; do not move cursor.
put_left (v: G)
        -- Add v to left of cursor position; do not move cursor.
    require
        not_before: not before
put_right (v: G)
        -- Add v to right of cursor position; do not move cursor.
    require
        not_after: not after
extend (v: G)
        -- Add v to end; do not move cursor.
```

Unlike the last two examples, which showed full implementations, these are just interface specification of the corresponding EiffelBase features.

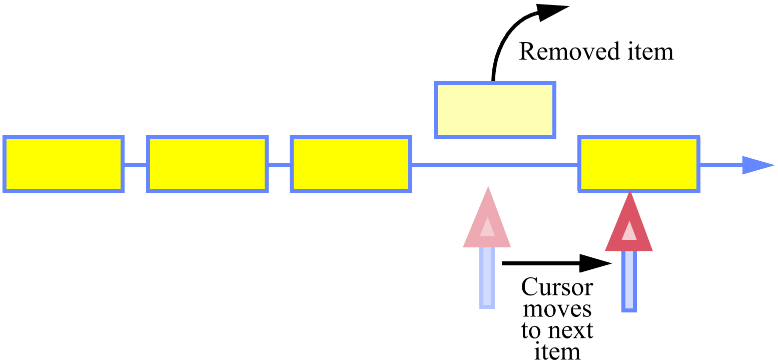
As the comments indicate, these procedures are designed to have no effect of the cursor, since there is no reason an insertion should change the currently active position in the list.

> In many cases the implementation does change the cursor temporarily; for example it is possible to implement *extend* (*v*) as
>
> ```
> original_index := index
> finish
> put_right (v)
> go_i_th (original_index)
> ```
>
> with an integer variable *original_index*, as in *has*, to record the initial index position, enabling the command to restore the cursor position at the end.

To delete elements, you may use

```
remove
        -- Remove item at cursor position; move cursor to right neighbor
        -- (or to after if no right neighbor).
    require
        item_exists: not off
    ensure
        removed: count = old count – 1
        after_when_empty: is_empty implies after
```

In this case the cursor has to be moved because the item to which it was pointing goes away:



Removing the current item

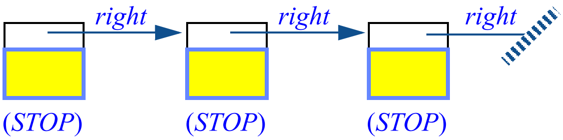
Also available are *remove_left* and *remove_right*, acting on positions next to the cursor, which do not change the cursor position. Write their specifications (signature, header comment, contract) as an exercise.

13.7 LINKED LISTS

We have now seen the basic properties and features of lists, independent of any implementation. We turn our attention to specific variants, in particular the important case of *linked* implementations, with class *LINKED_LIST*.

Linked list basics

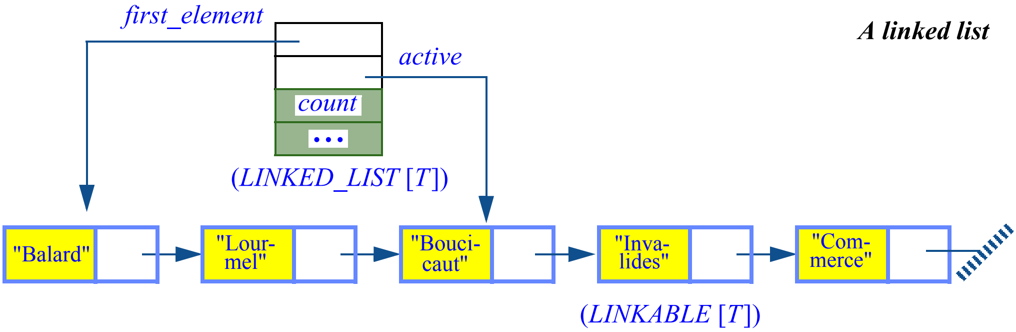
In our work with metro stations we saw the technique of linking elements of a sequential structure:



Linking stations

← This figure first appeared on page 116.

We can generalize this — thanks to the genericity mechanism — to arbitrary structures. An instance of *LINKED_LIST* [*T*] for some type *T* will refer to zero or more linked cells, or "*linkables*", each containing a value of type *T* and a reference to another possible linkable:



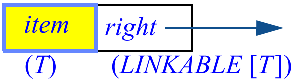
A linked list

As the figure indicates, the implementation involves two classes:

← As previewed in "Making lists explicit", page 262.

- The top object is an instance of *LINKED_LIST* [*T*]. Such an object is known as a **list header**; it contains general information about the list and provides access to items, but does not itself represent any item. Field *count* denotes the number of elements, if implemented by an attribute (it could also be a function). Other fields are references to list cells; they include *first_element*, leading to the first cell, and *active*, leading to the item at cursor position.
- The other objects represent list cells; they are instances of a class *LINKABLE*, also generic and using the same actual generic parameter, here *LINKABLE* [*T*].

In normal usage, client applications that need linked lists will only use the class *LINKED_LIST*. *LINKABLE* is an implementation class; it represents a very simple notion of list cell that can be linked to other similar cells; a typical instance looks like this:



***An instance of** LINKABLE [T]*

The implementation of *LINKED_LIST* routines relies on features from *LINKABLE*: the queries

```
item: G
          -- Value in cell.
right: LINKABLE [G]
          -- Next item.
```

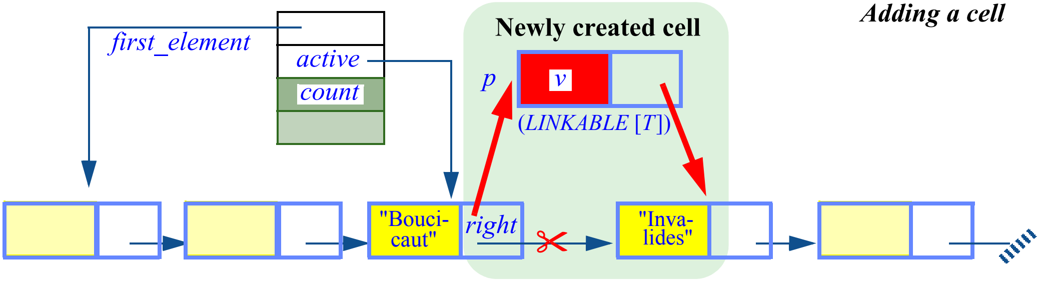
and the associated setter commands:

```
put (x): G
          -- Set item’s value to x.
     ensure
          set: item = x
put_right (other: LINKABLE [G])
          -- Link to other.
     ensure
          set: right = other
```

Insertion and removal

Below is a picture of how class *LINKED_LIST* implements the command *put_right*, which — as specified earlier — must add an item to the right of the cursor, without moving the cursor. For a linked list, it suffices to create a new *LINKABLE* cell and update the linking:

← *Page 398.*



Adding a cell

In the implementation of the routine, the operation illustrated on the previous page uses two calls to *put_right* from *LINKABLE*, highlighted below. Note how it must also include special treatment to handle the *before* case properly:

← Generalizing put_next, page 258.

```
put_right (v: G)
        -- Add v to right of cursor position; do not move cursor.
    require
        not_after: not after
    local
        p: LINKABLE [G]             -- The cell to be created
    do
        create p.make (v)
        if before then              -- Special before case:
            p.put_right (first_element)
            first_element := p
            active := p
        else                        -- The most common case:
            p.put_right (active.right)
            active.put_right (p)
        end
ensure
        next_exists: active.right /= Void
        inserted: (not old before) implies active.right.item = v
        inserted_before: (old before) implies active.item = v
    end
```

This routine is from LINKED_LIST; it includes three calls to a routine with the same name from LINKABLE. The reuse of the name is part of the notational convention ("Standardizing feature names for basic operations", page 374) and causes no ambiguity since p is of type LINKABLE.

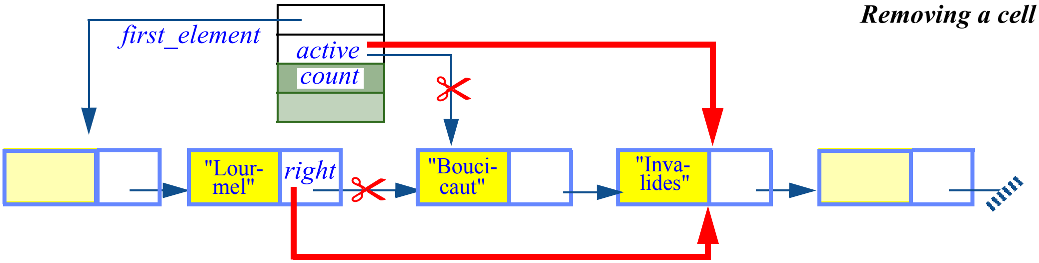
With its need to juggle references, this example illustrates the earlier discussion of how delicate it is to program with references. Even though it performs a simple operation, the algorithm requires care to ensure that it works correctly in all details. The difficulty will increase in the next two examples, item removal and list reversal.

← "Where to use reference operations?", page 263.

The procedure *remove* discards the cell at cursor position; as we have seen, the specification states that the cursor will move to the position immediately to the right as shown on the next figure. The implementation must, as illustrated, change two references:

← Page 399.

- It must reattach the *right* link of the cell just before the cursor position (with item value "Lourmel") to bypass the item at cursor position.
- To update the cursor as required, it must reattach the *active* link of the *LINKED_LIST* object to the item (here "Invalides") just after the previous cursor position.



Removing a cell

Here too you should look up the actual implementation: routine *remove* in *LINKED_LIST*. The details are more intricate than for *put_right*, as there are several special cases, including when the cursor is on the first or last item. It may help to read first the text of routine *remove_right*, somewhat simpler.

Reversing a linked list

As a final illustration of algorithms manipulating references, and the care they require, let us write a routine that — just for once in this chapter — does *not* figure in the corresponding EiffelBase class, here *LINKED_LIST*, at the time of writing. (There is no obvious reason why it is not there.) We want a *procedure* to reverse the items in a list. The basic idea is clear since we already wrote a *function* to produce a *new* list reversed from an existing one. Now we must address two more issues: the generalization to arbitrary linked lists; and the need to reverse an *existing* list in place, which makes things trickier.

← "Reversing a linked structure", page 259.

First we may note how *not* to do it. A correct but inefficient algorithm proceeds as follows: go to the last item, attach *first_element* to it and link this item to its left neighbor (the next-to-last); go to the next-to-last item, link it to its left neighbor; and so on, starting from the right and linking each item to its left neighbor. You will need a local variable to retain the original *first_element* reference before you reset it, and another to retain, in each traversal, a reference to the item immediately to the left of the one you process.

Programming Time!

An O (n^2) reversal algorithm

Turn the above informal description into a routine, and try it on some lists of integers. Instrument the code so as to count the number of loop iterations.

Since it is not a good idea to modify a library class such as *LINKED_LIST*, you can simply write a small class that *inherits* from it:

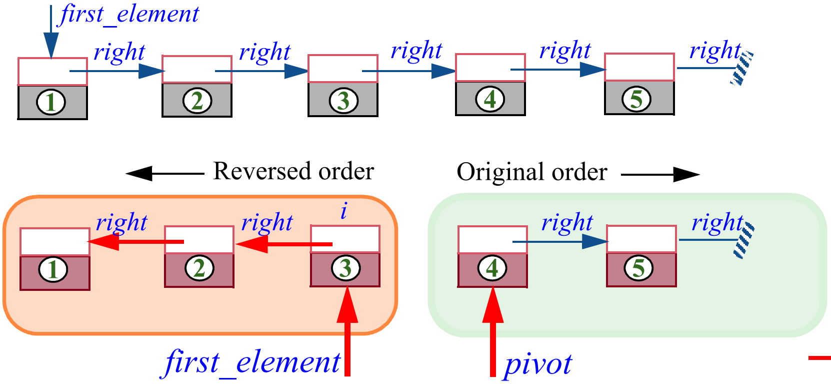
```
class MY_INTEGER_LIST inherit LINKED_LIST [INTEGER] feature
        reverse do … Your code here … end
end
```

You do not need any of the other properties of inheritance.

→ Chapter 16.

The problem with this approach is its performance: the first traversal takes *count* iterations, the second one *count* – 1, the third *count* – 2 etc., so the overall number of iterations is *count* (*count* + 1) / 2, meaning **O** (*count*^{2}). We want an **O** (*count*) algorithm. and will now devise one. It follows the same lines as the function version, using a single loop, but changes the structure in place.

As usual the only way to understand the process is to see the loop invariant. Here it is in visual form, showing the state after a typical iteration:



Reversing a linked list: intermediate state (original state at top)

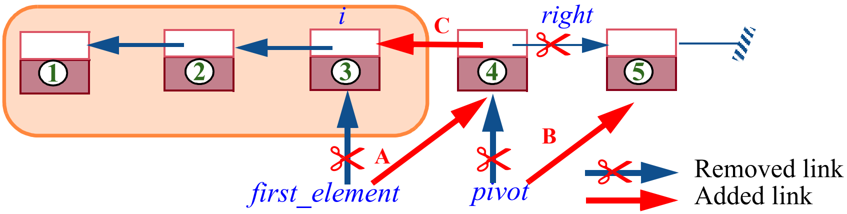
(*Compare with the figure on page 260 for the function version.*)

In this intermediate state:

- *first_element* is attached to one of the items of the original list (as in the figure), or is void.
- We consider that *first_element* always represents a position *i* in the original list: the item's position or, if *first_element* is void, the position to the left of the first item if any. (This also means that for an empty list — remember to check border cases! — *first_element* is void.)
- *pivot* is attached to the item that was the immediate right neighbor of *i* in the original list. Consistent with this rule, *pivot* is void if and only if either *i* was the last item of the original list or the list is empty.
- Starting with *pivot* and following *right* links yields a list that is made of all the original items past *i* if any, in their **original** order.
- Starting with *first_element* and following *right* links yields a list that is made of all the original items up to *i* if any, but in the **reverse** order.

This property has all the makings of a good invariant for an iterative process, meaning a process of successive approximations: it is easy and in fact trivial to ensure initially (set *pivot* to the original *first_element* and *first_element* to void); it yields the desired result upon termination (when *i* is the last position of the original list, *first_element* gives us the entire reversed list!); and when it is satisfied in an intermediate state we can easily extend it to cover one more item, using three reference reattachments as pictured:

← *"Loops as approximations", page 154.*



Reversing a linked list: adding one item

(*Compare with the figure on page 261 for the function version.*)

The code for such an iteration of the loop is simply:

```
i := first_element
first_element := pivot            -- The operation labeled A in the figure
pivot := pivot.right              -- Operation B
first_element.put_right (i)       -- Operation C
```

Note the need for a temporary variable *i* to record the original *first_element* so that we can link the new *first_element* cell to it in the last operation, **C**.

Here is the full algorithm, bringing everything together. To express the variant — things are subtle enough that we should take care to ascertain termination — we add an integer local variable *c*, counting iterations (or, equivalently, recording the position of the current *first_element* in the original).

```
reverse
        -- Re-link items in reverse order.
        -- (No precondition, will work for empty list.)
        -- Do not move cursor.
    local
        pivot, i: LINKABLE [G] ; c: INTEGER
    do
        from
            pivot := first_element ; first_element := Void ; c := 0
        invariant
            -- c is index of the first_element item, if any, in original list;
            -- list starting at first_element includes all items up to position c
            -- in original, in reverse order; list starting at pivot includes all items
            -- past position c in original, in original order.
        until
            pivot = Void
        loop
            i := first_element
            first_element := pivot
            pivot := pivot.right
            first_element.put_right (i)
            c := c + 1
        variant
            count – c + 1
        end
    end
```

Make sure you understand the *reverse* and *put_right* algorithms thoroughly, as they give a good idea of what is involved in implementing linked list operations. They are another illustration of the difficulty of programming with references and of the Reference Programming Principle: such stuff belongs in either dedicated clusters of a system or general-purpose professional libraries, not the "business logic" of an application program.

← *"Touch of Methodology: Reference Programming Principle", page 263.*

You should test your understanding of the reversal algorithm by writing its variants for other implementations studied next: arrayed lists and two-way lists.

→ *Reversing lists of various kinds.* exercise *13-E.5, page 434*

Performance of linked list operations

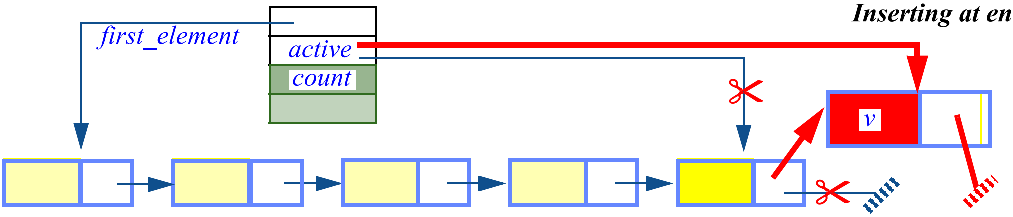
We can assess the cost of list operations in the case of a linked implementation:

- The complexity is **O** (1) for operations that need only perform operations at the cursor position: *put_right*, *remove_right*.
- Operations that may need to traverse the list are **O** (*count*). This is the case, as we already saw independently of the implementation, for *search* and *has*. The procedure *reverse* as just seen is **O** (*count*) too; so is general cursor movement, *go_i_th*, as well as *finish* (implemented as *go_i_th* (*count*)).

An interesting case is *extend*, to add an item at the end of the list. As noted, this can be implemented as *finish* followed by *put_right*, and *go_i_th* if we need to restore the cursor position. All three operations are **O** (*count*). But often you will need, for example when initializing a list, to add items repeatedly at the end. If you can let the cursor remain on the last item (query *is_last*), then *extend* will just perform a *put_right* followed by a *forth*, and hence will be **O** (1):

← *Page 399.*

Inserting at end



Some operations working at the cursor position are more delicate than *put_right* and *remove_right* since they may need a reference to the item immediately to the left. This is the case with *remove*, which removes the item at cursor position, and therefore must link its left neighbor to its right neighbor. Including an attribute *previous* attached to the left neighbor makes them **O** (1) — but slightly decreases the efficiency of other operations, since they must update *previous*.

The interface of *LINKED_LIST* and other list classes makes no obvious difference between *forth*, moving the cursor forward one position, and *back*, moving it backward. In this implementation, however, the performance is radically different: *forth* is **O** (1), but *back* has to be implemented as

```
start
go_i_th (index – 1)
```

which is **O** (*n*) (with a *previous* attribute you can perform one *back* in **O** (1), but only one, invalidating the value of *previous*, so this not very useful). Symmetric structures, such as *TWO_WAY_LIST* seen next, remove this discrepancy.

> The precondition of *back* is **not** *before*; it implies that *index* ≥ 1 and hence that *index* – 1 satisfies the precondition of *go_i_th*.

Here is the complexity summary, as for other structures in this chapter. First the insertion and removal operations at cursor position:

Operation	**Features in class** *LINKED_LIST*	**Complexity**	**Comments**
Insert at cursor position	*put_right*, *put_left*	**O** (1)	For operations left of cursor, **O** (1) requires a *previous* attribute. *remove_left* is **O** (*count*)
Remove at cursor position	*remove_right*, *remove*	**O** (1)	
Insertion at end, if cursor already there	*extend*	**O** (1)	

Then cursor movements:

Move cursor to first	*start*	**O** (1)	For operations left of cursor, **O** (1) may require a *previous* attribute.
Move cursor to last	*finish*	**O** (*count*)	For operations left of cursor, **O** (1) requires *previous* attribute.
Move cursor one step right	*forth*	**O** (1)	For operations left of cursor, **O** (1) may require a *previous* attribute.
Move cursor one step left	*back*	**O** (*count*)	

Finally, global operations that may require a traversal:

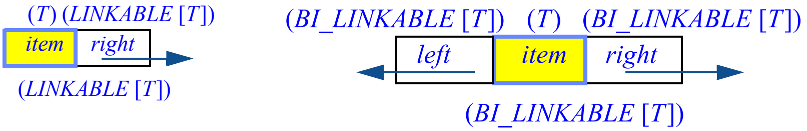
Insert at end, if cursor not there	*extend*	**O** (*count*)	
Search	*search*, *has*	**O** (*count*)	
Reverse	*reverse*	**O** (*count*)	Not in class, but given above

13.8 OTHER LIST VARIANTS

LINKED_LIST is only one of the implementations of lists.

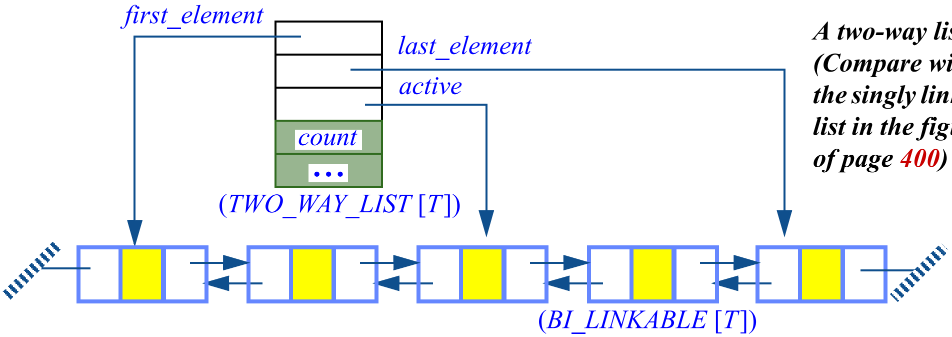
Two-way lists

LINKED_LIST favors left-to-right traversal; the huge performance discrepancy between *forth* and *back* is a consequence. Class *TWO_WAY_LIST* provides a fully symmetric version. The cost is in space, since instead of a *LINKABLE* with one reference every cell is a *BI_LINKABLE* with two:



Instances of LINKABLE [T] and BI_LINKABLE [T]

A *TWO_WAY_LIST* retains not only *first_element* but also *last_element*, a reference to the last item in the list:



A two-way list (Compare with the singly linked list in the figure of page 400)

This symmetry brings us **O** (1) performance for left-of-cursor operations such as *remove_left* and *put_left* as well as their right-of-cursor counterparts, and for *back* as well as *forth*.

Abstraction and consequences

Here I should tell a little story from the battlefield. One day the programmers at a certain company were complaining to their manager that this object-oriented stuff was way too slow. The manager asked a senior developer to check the code, only to discover that it was performing *back* operations again and again — on instances of *LINKED_LIST*. Replacing this by *TWO_WAY_LIST* yielded an instant speedup factor of 23 (that is, the code ran twenty-three times faster). The programmers lived happily ever after, and never a single time did they raise their voices again about the speed of the generated code.

This highly moral tale holds two important lessons on *abstraction*, a cornerstone of modern software development.

On the one hand, it shows the risks of abstraction: while it is elegant to consider *back* an operation applicable to lists of any kind — and indeed, we saw how to implement *back* for a one-way linked list — the danger exists that we lose track of the efficiency side of the story. By viewing a list just as a list — an abstraction that later techniques such as polymorphism and dynamic binding will encourage even further — we might forget that sometimes it is a two-way list, making *back* a snappy $\mathbf{O}(1)$ formality, and sometimes a one-way list, turning *back* into a sluggish $\mathbf{O}(n)$ chore. The first lesson is that in the practice of professional software development, where performance is one of the inescapable constraints, you should not let the benefits of functional abstraction obscure efficiency properties.

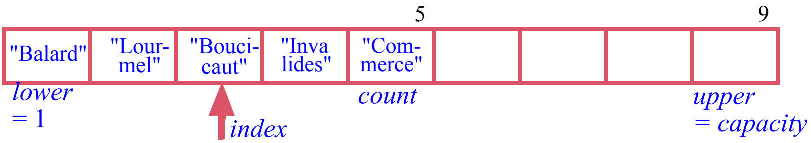
→ *“Polymorphism”, 16.2, page 557; “Dynamic binding”, 16.3, page 562.*

If this first lesson highlights the possible dark side of abstraction, the second lesson is to the credit of abstraction. To achieve the speedup, it sufficed in a few declarations to change the type from *LINKED_LIST* to *TWO_WAY_LIST*. Without object-oriented techniques and abstraction, the implementation details would have been buried deep into the software, and the change would have been far more intricate.

Abstraction is not the enemy of performance; it can initially hide performance issues for an inattentive observer, but also helps you identify these issues effectively and correct any deficiency you discover.

Arrayed lists

Rather than a linked structure, you can use an array to represent a list:



An array implementing a list

The EiffelBase class *ARRAYED_LIST* provides this implementation. Do not confuse it with *ARRAY*: the exported features — visible to the clients of *ARRAYED_LIST* — are those of *LIST*, implemented through the features of *ARRAY* such as *item* and *put*. Internally, as shown in the figure, *lower* is 1, so as a result of the array invariant *capacity* = *upper* – *lower* + 1 the upper bound *upper* is equal to *capacity*, the array’s physical size.

For an *array*, the number of items *count* is also the same as *capacity*: the array’s items are those that fit in its representation. Not so for an *arrayed list*: *count* is the *count* of lists, while *capacity* is the maximum *count* that can be accommodated without resizing. In the above figure *count* is 5 and *capacity* is 9, and the occupied positions are those with indexes 1 to *count*. The class invariant includes the property *count* <= *capacity*.

The cursor is represented by the integer *index*, which *ARRAYED_LIST* implements as an attribute. As another sign of the difference between arrayed lists and arrays, array indexes can only go from 1 to *capacity* (*lower* to *capacity*), but *index* can, as with lists in general, range from 0 to *capacity* + 1.

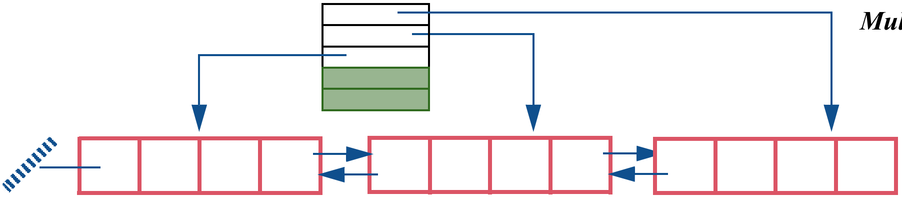
The implementation does not have to put up all items in an area starting at position 1; to support left-of-cursor insertions it may be useful to treat the array *circularly*, with two integer markers at the end; the technique will be described for circular queues. No implementation trick, however, can remedy the fatal limitation of an arrayed implementation: an insertion or removal requires moving all the items to the left or the right of the cursor. These operations are then, fundamentally, **O** (n). (For removals, we can delay the day of reckoning by leaving blank entries for a while, but at some point we will run out of entries for insertions and will have to perform the **O** (n) compaction.) If an insertion would cause *count* to exceed *capacity*, the array must be resized — an expensive operation as we know.

→ "Queues", 13.12, page 428.

This property severely limits the usefulness of arrayed lists. They remain interesting if the scenario for a certain list structure includes an initialization period where items are entered, after which the list mostly remains stable, with few if any insertions and removals. Then an arrayed list provides the benefits of arrays: in space (no need for reference fields such as *right* and *left* in the earlier solutions); and in speed if some random access will be needed. Note in particular that the one command for which arrayed lists shine in time complexity is *go_i_th* (i), implemented as just *index* := i and hence **O** (1), whereas it was **O** (n) in the linked implementations.

Multi-array lists

Many variations are possible on the themes listed. You may occasionally be interested in *MULTI_ARRAY_LIST*, which attempts to combine the benefit of arrayed and linked structure (at the price of added complexity, as you can see from the code of the class). A multi-array list is a two-way list of arrayed lists:



Multi-array list

(*BI_LINKABLE* [*ARRAYED_LIST* [*T*]])

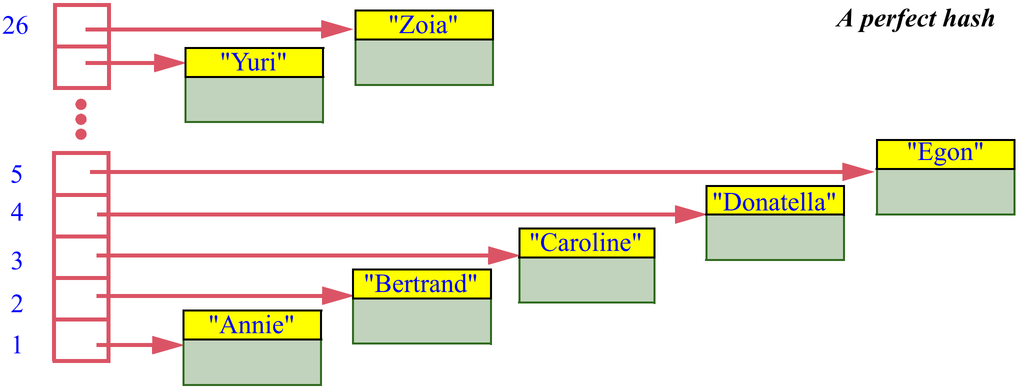
One advantage is that there is never a need to resize (and hence reallocate) an array: when running out of space, the structure allocates a new arrayed list. If it shrinks, it gives up an arrayed list. Worst-case performance remains **O** (n) for several key operations, but ordinary usage can be more favorable.

13.9 HASH TABLES

Arrays represent structures indexed by integers. What if we want other kinds of key? Strings are a common example. You may need containers where the access criterion is a character string, such as:

- A directory of people — a container where each object represents information about one person; you will want to retrieve these objects through people's names, in the same way that you find a person, in a traditional paper directory, by looking up the name.
- A collection of Web pages, as maintained by a search engines; the pages are indexed by all the words that appear in them.

Assume for a moment that in the first example all the people in the directory have names starting with a different letter: Annie, Bertrand, Caroline, … Then you could use an array of twenty-six entries, each corresponding to a letter code, 1 for A, 2 for B and so on:



A perfect hash

We have **hashed** the keys (the strings representing the names) into integer values in the interval 1..26. "Hashing" is understood here by analogy with mincing up food into small pieces. Specifically:

> ***Definition*: Hash function**
>
> A **hash function**, for a set K of possible keys, is a function h that maps K into some integer interval $a..b$.

In other words, for any $key \in K$, the function gives you a value $i = h\ (key)$ such that $a \leq i \leq b$.

In practice the interval is usually of the form 0..*capacity*–1 for some integer *capacity*, with *h* (*key*) of the form *f* (*key*) **modulo** *capacity* for some basic function *f* returning an integer. The array will then be of size *capacity*.

The example used a primitive hash function that simply returns, for a string key, the rank of the first letter, in the interval 1..26. A slightly more sophisticated function would take the ASCII codes of *all* characters in the string, add them, and take the remainder by *capacity*.

← *ASCII is the standard encoding of basic characters, with values from 0 to 255.*

The hash function depends only on the item key, not on the number of items, so if *count* is the measure of our problem's size execution will be O (1). (**O** (*l*) if we take into account the length *l* of keys, but we may assume that the hash function only uses the first *K* characters of the key for some constant *K*.)

The assumption behind the example was that each name started with a different letter, giving a different hash value. A hash function that gives a different value for every element of a given set of keys is called a **perfect hash** for those keys. With a perfect hash, insertion and search are **O** (1).

In most cases, we will not get a perfect hash, even with a better hash function such as the sum of all codes modulo *capacity*. A **collision** occurs — with a non-perfect hash function — when two different keys give the same hash value. A good hash function will cause fewer collisions — it is in this sense that we can say that the second example, sum modulo *capacity*, is "better" than the first — but will not in general avoid them completely. In fact, if the hash function computes its result modulo *capacity*, collisions are inevitable as soon as we deal with more than *capacity* keys. The implementation of hashing must be able to handle them.

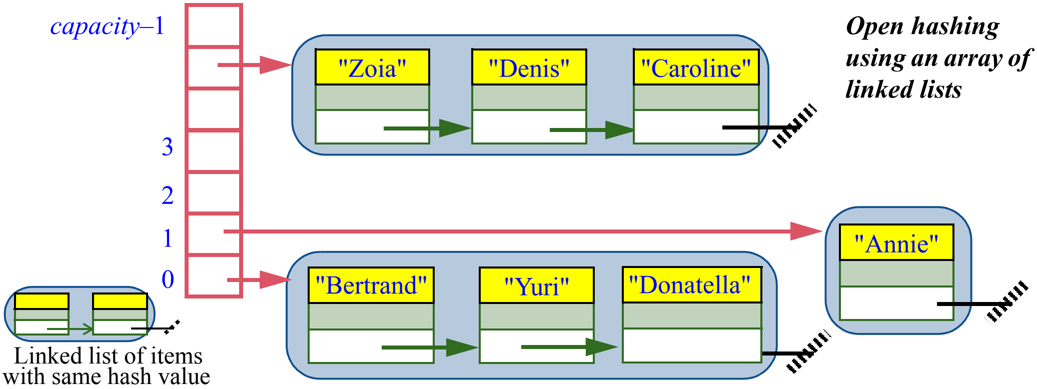
One technique is **open hashing**, which combines arrays with linked lists. In the last figure, with a perfect hash, the array directly contained items and would have been declared as

```
ARRAY [G]
```

but with open hashing we may use an array of *linked lists* of objects:

```
ARRAY [LINKED_LIST [G]]
```

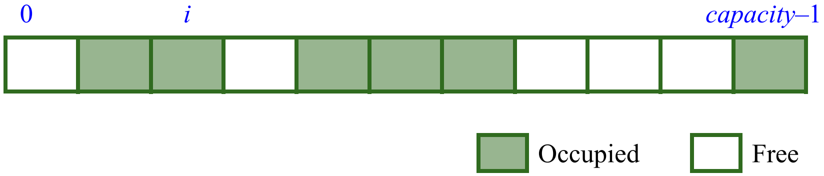
In each entry of the array, for a certain index *i*, you find the list of objects whose keys hash to *i*:



Open hashing using an array of linked lists

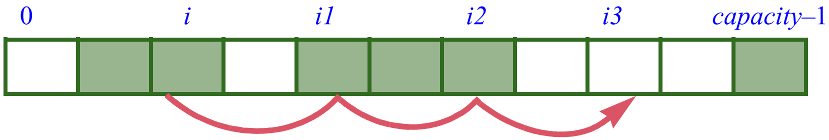
With open hashing we search for an item, or insert it, by first hashing its key into an index, which gives us an array entry, then performing a sequential search in the associated list. The cost is **O** (1) for the first operation and **O** (*c*) for the second, where *c* is the collision factor — the average number of keys hashing to a given value. If the array size *capacity* is constant, the value of *c* for a large *count* and an evenly distributed hash function will be **O** (*count* / *capacity*), that is to say **O** (*count*). To avoid this linear behavior we would need periodically to resize the array; but then it is usually better to use the other technique for collision resolution, **closed hashing**.

Closed hashing — as in the EiffelBase class *HASH_TABLE*, which you may study for a deeper understanding of hashing — uses no linked structure but only an *ARRAY* [*G*]. At any time some of its positions will be occupied and some free:



Array implementing a hash table in closed hashing

If for an insertion the hash function yields an already occupied position, for example the one marked *i* above, the mechanism will try a succession of other positions — *i1*, *i2*, *i3* below — until it finds a free one:



Looking for a free position

A common technique, if the hash function yields a first candidate position i = f (key) **modulo** capacity, is to try successive positions i + increment, i + 2 * increment, i + 3 * increment and so on, all modulo *capacity*, where *increment* (2 on the last figure) is f (key) **modulo** (capacity – 1). This is the algorithm used by *HASH_TABLE* in EiffelBase; see its implementation in routine *search_for_insertion* if you want to study it in detail.

To guarantee that the process terminates (meaning that the corresponding loop has a variant) the algorithm must always find an empty slot. This is guaranteed by an appropriate choice of parameters and by a policy that reallocates the array — *resize* is essential here — if it fills up. In fact we should not wait until the last minute: reallocation should occur as soon as the fill factor reaches a preset limit, which class *HASH_TABLE* sets at 80% (see the feature *Max_occupation* in the class). What's amazing is that with this policy, and a good choice of hash function, search and insertion in a hash table are essentially **O** (1). (For the theoretical complexity analysis leading to this property, see the references at the end of this chapter.)

This behavior means that for practical purposes you may see hash tables as almost as good as arrays, generalized to arbitrary keys, not just integers, as long as the keys are "hashable". Strings, for example, are hashable, so you may consider a hash table of string-identified objects as if it were an array indexed by strings rather than integers.

> This is a remarkable result, since *really* indexing by strings would lead to impossibly huge structures. Consider for example strings of at most 7 lower-case letters; the number of possibilities is approximately 26^7, or 8 billion, but it would be absurd to use an array of that size even if we had the memory, since any practical use needs only a small subset of these possible strings. By hashing the strings we allocate just a little more space than what we actually need (per *Max_occupation* noted above) and still get time behavior comparable to that of an array.

Finding hash functions that yield such efficient behavior is somewhat of an art; you can again take inspiration from the function used in class *HASH_TABLE*.

That class, *HASH_TABLE* [*G*, *KEY*], is our first example with two generic parameters rather than just one; *G* represents the type of items and *KEY* the type of their keys. You may use it for example to declare a hash table of objects representing persons, indexed by their names, as

```
personnel_directory: HASH_TABLE [PERSON, STRING]
```

Here are some of the fundamental features of *HASH_TABLE*. The class has a single creation procedure *make*; to create a hash table, use for example

```
create personnel_directory.make (initial_size)
```

where *initial_size* is some positive integer. It does not matter much what value you select; as the name suggests, this is just a hint for the initial allocation. If you are too far below the real need, you will just pay for one more resizing (automatic, of course) at run time.

Next, the main queries. To find out if there is an item for a certain key, use

```
has (k: KEY): BOOLEAN
```

To obtain the item associated with a given key, if any:

```
item (k: KEY) alias "[]" : G assign put
        -- Item associated with k, if any; otherwise default value of type G.
    ensure
        default_value_if_not_present:
            not (has (k)) implies (Result = computed_default_value)
```

The postcondition indicates that if there **isn't** an item for the given key, the result is the "default value" of type *G* (zero for numbers, false for booleans, void for references). This is not a good way to test for the presence of an item in a hash table, since there could be an item with the default value; so if you are not sure whether the key appears, use *has* first.

The **alias** "[]" specification indicates, as with *item* for arrays, that bracket notation is available for the item associated with a certain key: you may write

← *"Bracket notation and assigner commands", page 384.*

```
personnel_directory ["Isabelle"]
```

as a synonym for

```
personnel_directory.item ("Isabelle")
```

The bracket form is shorter and we will use it whenever applicable.

To insert an item into a hash table, you will need to provide both the item and its key, as in

```
personnel_directory.put (that_person, "Isabelle")                [8]
```

even if the key is in fact a property of the item, as in

```
personnel_directory.put (that_person, that_person.name)
```

The class offers four insertion operations with the same signature:

```
put (new: G; k: KEY)        -- This is the assigner command for item.

force (new: G; k: KEY)

extend (new: G; k: KEY)
    require
        not_present: not has (k)

replace (new: G; k: KEY)
```

Among them, *extend* has a precondition stating that it is only applicable if the key is not already used; the other three are always applicable. A "**note**" clause at the beginning of the class explains when to use each variant; since it says exactly what there is to say I am reproducing it here, omitting only a few details:

Insertion variants for hash tables
(from the text of class *HASH_TABLE*)

- Use *put* if you want to do an insertion only if there was no item with the given key, doing nothing otherwise.
- Use *force* if you always want to insert the item; if there was one for the given key it will be removed.
- Use *extend* if you are sure there is no item with the given key, enabling faster insertion.
- Use *replace* if you want to replace an already present item with the given key, and do nothing if there is none.

In the first two cases the procedure will set the value of the boolean query *found*, enabling you to find out, after insertion, if there already was an item with the given key.

The declaration of *item* both has the bracket alias and names *put* as its associated *assigner command*:

← *"Bracket notation and assigner commands", page 384.*

```
item (k: KEY) alias "[]": G assign put
    … See rest of declaration on previous page …
```

As a result, bracket notation is available for calling this procedure, allowing you to insert an item into a hash table through an assignment-like instruction:

```
personnel_directory ["Isabelle"] := that_person
```

which is just a shorthand for the call to *put* that was written earlier in ordinary dot notation [8].

To remove an item with a given key, use

```
remove (k: KEY)
```

This has no effect if the key was not present; you can find out afterwards through the query *removed*.

Calling *clear_all* will remove all the current entries.

Throughout these operations, you do not have to worry about the size of the data structure; thanks to the resizable nature of Eiffel arrays, the routines will take care of maintaining enough space for all the current items, plus breathing space to ensure the fill ratio remains at most *Max_occupation*.

If you explicitly want to change the size, a call to *accommodate* (*n*: *INTEGER*) will ensure that the table can accommodate *n* items; it will not discard any item already present in the table.

Here is a summary of the cost of hash table operations.

Operation	**Feature in class** *HASH_TABLE*	**Complexity**
Key-based access	*item*, *has*	**O** (1)
Key-based insertion	*put*, *force*, *extend*	**O** (*count*)
Key-based replacement	*replace*	**O** (1)
Removal	*remove*	**O** (1)

As you start working on systems operating on large numbers of objects that must be easily stored and retrieved based on their actual contents, you will find in hash tables one of your most consistently useful tools.

13.10 DISPENSERS

Arrays and hash tables are *indexed* structures:

- When inserting an item, you give some identifying information, such as the index in an array and the key in a hash table.
- To access an item, you must provide the associated index or key.

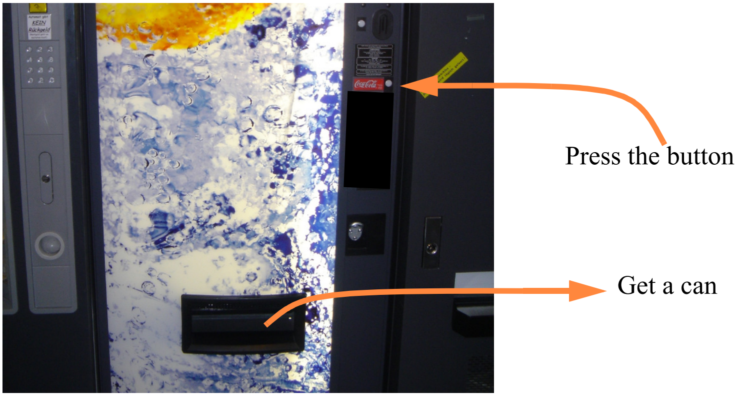
The next (and last) structures we will study follow a different policy. They use no key or other identifying information for items. You insert an item just by itself, typically through a procedure

```
put (x: G)
        -- Add x to current structure.
```

(Compare with *put* (*x*: *G*; *i*: *INTEGER*) for arrays and *put* (*x*: *G*; *k*: *KEY*) for hash tables.) When it comes to retrieving items, you do not get to choose which one you get; the basic query is

```
item: G
        -- Item obtained from current structure.
    require
        not is_empty
```

with no argument (compare with *item* (*i*: *INTEGER*): *G* for arrays and *item* (*k*: *KEY*): *G* for hash tables). We call such structures **dispensers**, by analogy with a one-button vending machine as illustrated: the provider loads the machine with cans of soft drinks; after inserting a coin, the customer will get a can — *any* can — from those in the machine. The machine, not the customer, decides which can to deliver if more than one are available.



A dispenser

Dispensers differ in the policy the machine uses to select the item to deliver:

- Last-In First-Out: choose the item inserted most recently. A dispenser with a LIFO policy is called a **stack**.
- First-In First-Out: choose the oldest item not yet removed. A dispenser with a FIFO policy is called a **queue**.
- With a **priority queue**, items have an associated "priority" (an integer or real number); the query *item* will return the item with highest priority. Although this case seems closer to indexed structures, it is still an example of dispenser, as the priority is an intrinsic property of each item, rather than information provided by clients on insertion and retrieval.

For all dispensers, the four basic features are *put* and *item* with signatures and precondition shown above, the boolean query

```
is_empty: BOOLEAN
        -- Are there no items?
```

and a command to remove an item:

```
remove
        -- Remove item from current structure.
    require
        not is_empty
```

Just as *item* does not let you choose which item to access, *remove* does not let you choose which item to remove; as the comment indicates, the item removed is the one that *item*, had it been called just before, would have yielded.

A good implementation of dispensers should make all these operations execute in constant (**O** (1)) time; we will see examples shortly.

In some libraries you will find an operation that combines the effect of *remove* and *item*: a function, say *get*, that removes an item, and returns as its result the value of that item. We could implement such a function in terms of *remove* and *item*:

← See also the discussion on getter functions in "Setters and getters", page 248.

```
get: G
        -- Side-effect-producing function, violates methodology rules!
    do
        Result := item
        remove
    end
```

We will not use any such function since it would both change the structure and return a result, violating the rule that only commands, not queries, should be permitted to affect the state ("Command-Query Separation Principle"). For reasons explained in an earlier chapters, it is preferable to let clients access and remove items through two separate features, one a command, the other a side-effect-free query.

The next two sections cover stacks and queues. We will not study priority queues, but you may look up the EiffelBase class *PRIORITY_QUEUE*.

13.11 STACKS

A stack is a dispenser applying a LIFO policy: the item that you can access at any given time is the one added most recently. The place of access is called the "top" of the stack, and indeed the natural image is that of a stack in the ordinary sense, for example the set of dictionaries on my desk, assuming I can only pick the top item:

A stack

The "Towers of Hanoi" studied in the next chapter to illustrate recursion also function as stacks.

→ *See the figure on page 441.*

As another possible illustration you can think of a piggybank where you would insert and retrieve coins at the same end.

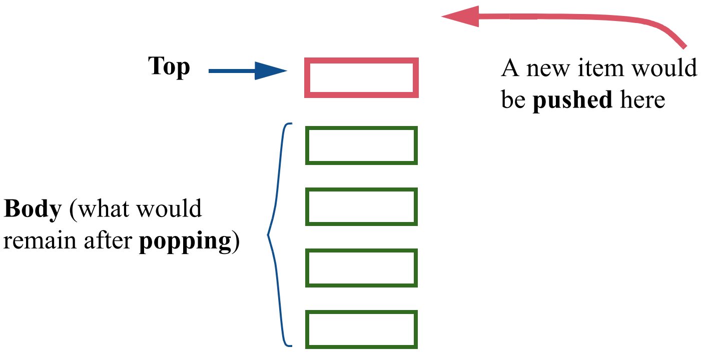
→ *Figure on page 535.*

Stack basics

The stack operations are often known as:

- **Push** an item to the top of the stack (command *put*).
- **Pop** the top item (command *remove*).
- Access the **top** element (query *item*).

which you may visualize as this:


Top
A new item would be **pushed** here
Body (what would remain after **popping**)

Conceptual image of a stack

Using stacks

Stacks have many applications in computer science. Two examples from programming language implementation — one static, the other dynamic — are *parsing*, as illustrated by the simple case of processing “Polish notation”, and the management of routine calls at run time.

Assume you want to evaluate a mathematical expression in **Polish notation**, a form often used by pocket calculators and sometimes as an internal form by compilers and interpreters. It has the advantage of being unambiguous without using parentheses: each operator applies to the operands that precede it; evaluating the operator on these operands yields an operand for the next operator.

For example the expression

$$2 + (a + b) * (c - d)$$

is represented in Polish notation as

$$2\ a\ b + c\ d - * +$$

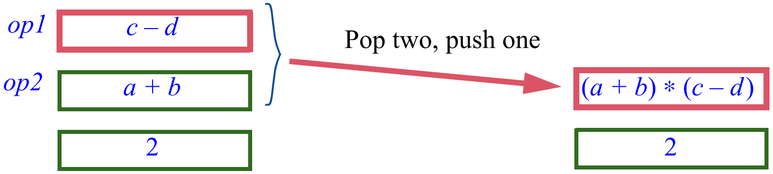
with the following meaning, corresponding to the intended value: the first $+$ applies to the previous two operands, leading to the new operand $a + b$; the $-$ operator applies to the previous two operands, leading to the new operand $c - d$; then the $*$ applies to these two resulting operands, leading to the new operand $(a + b) * (c - d)$; the final $+$ yields the sum of 2 (the first of all operands) and this result.

For simplicity all operators in this example are binary, but it is easy to adapt the scheme to operators that each specify their number of operands (*arity*).

The following algorithm, using a stack of operands *s*, evaluates a general Polish expression with binary operators:

```
from                                -- Initialization is empty
until
    "All terms of Polish expression have been read"
loop
    "Read next term x in Polish expression"
    if "x is an operand" then
        s.put (x)
    else -- x is a binary operator
            -- Obtain and pop the two top operands:
        op1 := s.item; s.remove
        op2 := s.item; s.remove
            -- Apply operator to operands and push result:
        s.put (application (x, op1, op2))
    end
end
```

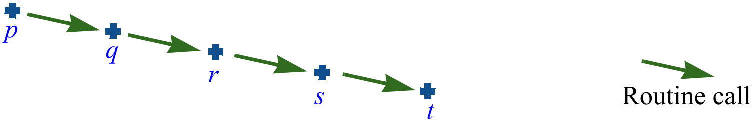
This uses two local variables *op1* and *op2* representing operands, and assumes a function *application* that yields the result of applying a binary operator to operands; for example *application* ('+', 2, 3) is 5. The following figure shows the algorithm's key operation, as expressed by the **else** clause, at the time of processing the ∗ operator in the example expression.



Polish expression evaluation

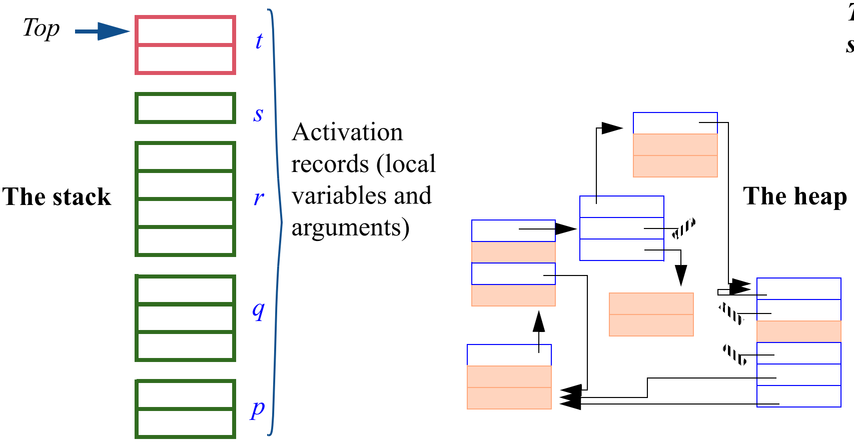
A proper implementation of the algorithm must handle erroneous input (by checking for *s.is_empty* before using *item* and *remove*, and checking in the **else** clause that *x* is an operator), and consider operators of varying arity.

Our second example underlies the run-time support of *every* modern programming language implementation and is present in every operating system (that is a strong statement, but no counter-example comes to mind). Consider a programming language where a routine can call a routine, which can call a routine, which …; at execution this yields a **call chain:**



Call chain

At any run-time moment several routines — *p* to *t* in the figure — have been started and not yet finished; the last one that started, here *t*, is the "**current routine**". Consider any one of its instructions, say an assignment $x := y + z$. Unless *x*, *y*, *z* are attributes of the enclosing class, they must belong to the current routine, as either **arguments** (not *x*, since we may not assign to arguments) or **local variables**. We use the term "locals" for both categories. To perform instructions such as this assignment, the code generated by the compiler must have access to all the locals. The solution is, on every routine call, to create an **activation record** containing the routine's locals:



The run-time stack and heap

The structure on the right is the *heap*, which contains the objects allocated through **create** instructions or equivalent. Of interest for the present discussion is the **call stack**, also called the run-time stack (and often just "The Stack"), containing the activation records for all currently active routines. Because no routine execution terminates until the execution of all the routines it has started terminates, the routine activation scheme is LIFO, and a stack is the appropriate structure.

> In many programming languages, routine texts can be *nested* (enclosed in others); then an instruction may refer not only to locals of the current routine but also to locals of any enclosing routine. This means that the execution may need access not only to the top activation record, but also to a few others below it. In this scheme the activation record structure — still called "the stack" — uses a slightly extended notion of stack. Eiffel does not need routine nesting.

On routine entry, the mechanism creates a new activation record for the routine (with the local variable entries set to the default initial values, and the argument entries set to the values of the actual arguments for the call) and pushes it onto the stack. On routine return, it pops the stack.

A benefit of using a run-time stack is that the activation records are not required to represent different routines — only different routine *executions*. As a result, this technique supports **recursive** routines: routines that call themselves, directly or indirectly. Allocating a new activation record for every new call allows each recursive call to use its own set of locals, distinct from any locals used by previous, still active incarnations of the same routine, which have their own activation records further down in the stack. Recursion is the topic of an entire chapter, and its implementation — based largely on stacks — of one of the chapter's sections.

→ *"Implementation of recursive routines", 14.9, page 486.*

Implementing stacks

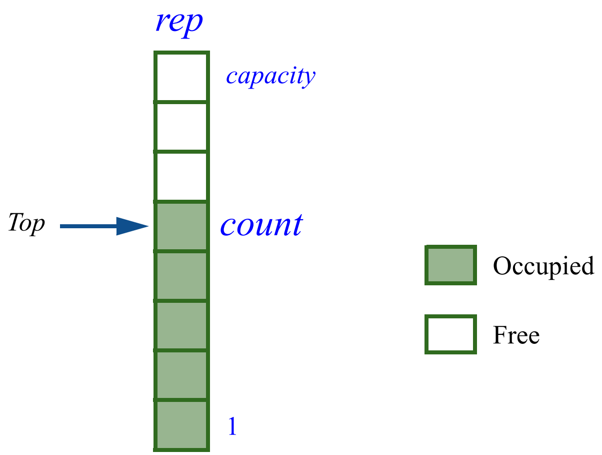
As with several other structures of this chapter, there are two general categories of stack implementations: arrayed and linked.

By far the most common implementation uses an array *rep* of type *ARRAY* [*G*] and an integer *count*, with the invariant

```
count >= 0 ; count <= rep.capacity
```

where *capacity* is the number of array items (*upper* – *lower* + 1). With the array indexed from one (*lower* = 1), the stack items if any are stored in positions 1 to *count* of the array:

← *See page 381.*



Arrayed implementation of a stack

In class *ARRAY*, the number of items is known as both *count* and *capacity*, with an invariant stating that the values of these attributes are equal. This *count* of arrays should not be confused with the *count* of stacks, which gives the number of stack items — in the arrayed implementation, the number of array positions occupied by stack elements.

We already encountered this distinction for arrayed lists which, like arrayed stacks, get their implementation from arrays and their specification (including *count*) from another container type. ← *"Arrayed lists", page 409.*

In this implementation, the query *item* giving the top item simply returns *rep* [*count*] the array item at position *count*; the command *remove* can be implemented as simply *count* := *count* – 1, and *put* (*x*) as

```
count := count + 1                                                    [9]
rep.force (x, count)
```

using the command *force* of arrays, which will perform a resizing if *count* outgrows the current *capacity* of the array.

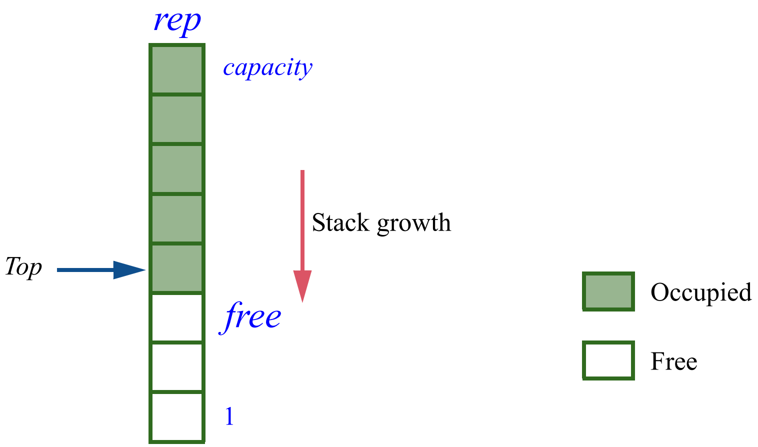
This implementation is what you will find in the EiffelBase class *ARRAYED_STACK*. (The class does not actually need *rep* since it *inherits* from class *ARRAY*, but this is conceptually equivalent and we have not studied inheritance yet.) The use of *force* for the algorithm of *put* means you do not have to worry about dimensioning the array properly; the array just starts out with a default size and gets resized as needed when you push items.

Of course, the available memory is limited in the end, so you still have to ensure that the total size of your data structures remains within control.

Array resizing is not commonly available outside of Eiffel, so most other arrayed implementations of stacks use a fixed-size array, which may be inevitable in some cases anyway (including in Eiffel) if you need to control memory tightly. The corresponding EiffelBase class is *BOUNDED_STACK*. For a bounded stack there is, along with *count*, a query *capacity* (implemented in the arrayed representation as *rep*.*capacity*) and a boolean query *is_full*, whose value is *count* = *capacity*. Then in the same way that *remove* has the precondition **not** *is_empty*, the command *put* now has the precondition ← *Page 419.* **not** *is_full*, and its array implementation (see [9] above) uses *put* rather than *force*; this is correct since ensuring **not** *is_full* will guarantee that the call to *rep*.*put* satisfies the precondition *valid_index* (*count*) of *put*, meaning here that ← *Page 383.* *count* must be between 1 and *capacity* inclusive.

All the operations cited are **O** (1) in time.

A variant of the arrayed representation, bounded, has the stack grow downward:



Downward arrayed implementation of a stack

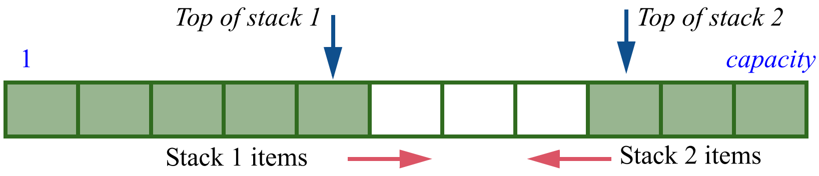
In this representation, *count* is no longer an attribute; it is replaced by a secret attribute *free* giving the index of the first free position. The query *count* must still be available to clients; it is now a function that returns *capacity* – *free*.

The invariant will state that *free* >= 0 and *free* <= *capacity*; compare with the requirement on *count* in the previous representation.

The case *free* = 0 corresponds to *is_full*, and *free* = *capacity* to *is_empty*; the items, if any, are in positions *capacity* down to *free* + 1. The implementation of *remove* is *free* := *free* + 1, and the implementation of *put* is

```
rep.force (x, free)
free := free – 1
```

If you have limited space available and *two* stacks, you can store both of them in a single array, using the upward scheme for one and the downward scheme for the other (they appear as leftward and rightward on the following figure):



Two stacks in one array

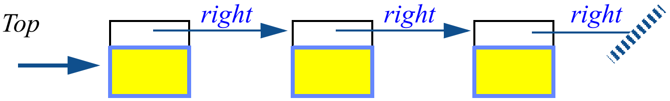
The advantage of this technique over two separate arrays is that it achieves a better use of space if the two stacks do not reach their maximum *count* together. Denoting by *max* (*x*) the maximum value of a mathematical variable *x*, we note that

$$\boxed{max\ (count1 + count2)\ \leq\ max\ (count1) + max\ (count2)}$$

so that a one-array representation of size 2 ∗ *n* might still have space available if one of the stacks has more than *n* items, whereas with two arrays of size *n* we run out of space as soon as either stack reaches *n*. An exercise asks you to write a class *TWO_STACK* implementing this idea.

→ *13-E.3, page 434.*

Along with arrayed implementations, you can use a linked representation for stacks. Indeed a linked list as studied earlier in this chapter provides a ready-made implementation of a stack. The figure below illustrates the technique: the first cell is the top, the rest of the list is the stack body.



Linked stack

The operation *put* (*x*) is implemented simply as *rep*.*put_front* (*x*), where *rep* is the linked list; *item* is just *rep*.*first* (where *first* for linked lists yields the first element, *i_th* (1)); and so on. Class *LINKED_STACK*, in EiffelBase, provides such an implementation. The basic operations are still **O** (1), although slower than in the arrayed versions; for example, *put_front* of *LINKED_LIST* and hence *put* of *LINKED_STACK* must allocate a new *LINKABLE* cell.

All the basic operations are indeed constant-time in the various implementations of stacks we have seen — with, as noted, the occasional exception of a call to *force* in a resizable stack:

Operation	**Feature in stack classes**	**Complexity**	**Comments**
Access top	*item*	**O** (1)	
Push to top	*put*	**O** (1)	With automatic resizing, occasionally **O** (*count*)
Pop	*remove*	**O** (1)	

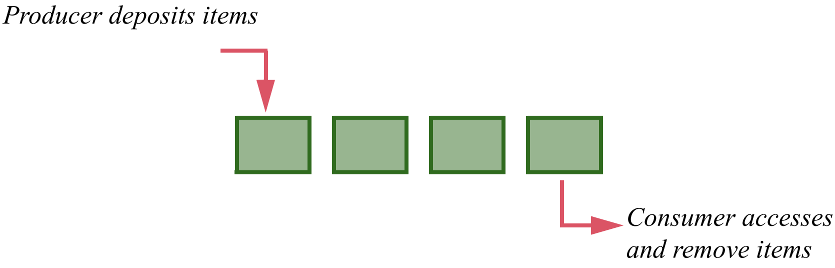
13.12 QUEUES

With their First-In First-Out policy, queues are useful in many applications. Two typical examples:

- In *simulation* applications, especially the variant known as *discrete-event* simulation, a program perform steps simulating what is happening in some process — an assembly line producing cars from parts, a network transmitting messages, a store serving customers — to analyze waiting times and remove inefficiencies. Often, the handling of events (parts arriving on the assembly line for processing, messages arriving on the network, customers arriving at the store) is FIFO; a queue will represent the pending events.

← The original application area of O-O languages, see "Object-oriented languages", page 327.

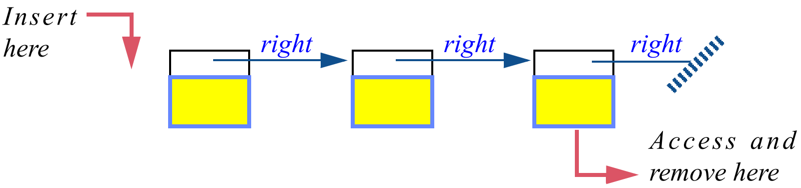
- A similar situation arises in a Graphical User Interface (GUI) system, where the events triggered by users — mouse clicks, cursor movements, key presses — should be processed in the order of arrival.
- In operating systems and other cases of concurrent programming, a frequently useful scheme is *producer-consumer* communication where one process, the producer, generates some information, which another, the consumer, reads and processes in the order of production. The structure used for the exchange of information is a queue, in a variant adapted for concurrent processing and known as the **buffer**.



Consumer-producer communication through a buffer

The last figure can serve as a conceptual representation of any queue, not just a buffer: insert items at one conceptual end, remove them at the other end.

As with stacks, we may use linked and arrayed representations. A linked implementation is straightforward:



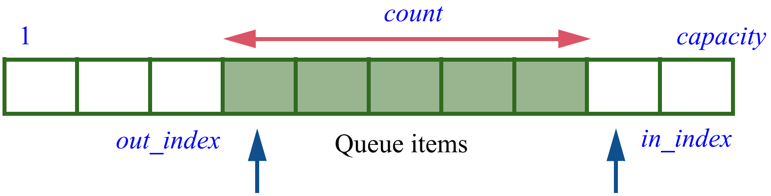
Linked queue

For example the operation *put* (*v*) will just be *rep*.*put_front* (*v*) (if as usual *rep* is the implementing structure, here a linked list), *item* returns the last item of the list, and *remove* removes it. The EiffelBase implementation, class *LINKED_QUEUE*, maintains the invariant

```
is_always_after: not empty implies rep.after
```

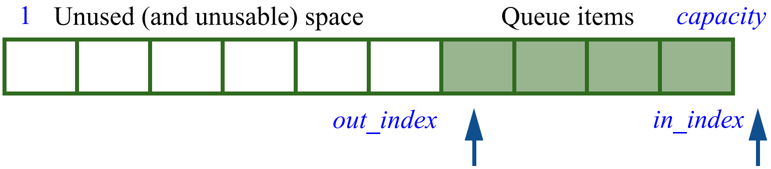
meaning that the list cursor is always past the last item.

The arrayed representation is a little more tricky than for stacks because we must remove elements at one end and insert new ones at the other. Instead of just one integer marker *count* we should maintain two, which the class *ARRAYED_QUEUE* calls *in_index* and *out_index*, both secret attributes. (The public query *count* is still there, giving the number of items.) It is not good enough, however, to use the interval *in_index*..*out_index* to store items, as in this simple picture



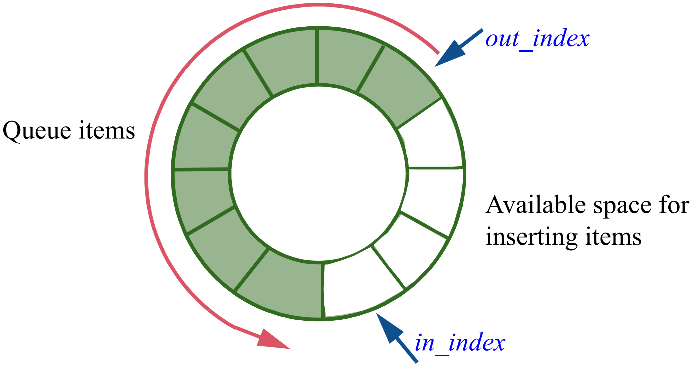
A possible state for an arrayed queue

(with the obvious implementation of *remove* as *out_index* := *out_index* + 1 and *put* (*v*) as *rep* [*in_index*] := *v* ; *in_index* := *in_index* + 1, where *rep* is an array); with this technique we would quickly run out of space after a few *put* even if, as a result of one or more *remove*, space remains unused at the beginning of the array:



Arrayed queue reaching right end of array

The solution: when the *in_index* marker reaches past *capacity*, the next *put* should cycle it back to the beginning of the array, and similarly with *remove* for *out_index*. Conceptually it is as if we wrung the array to bring its two ends together, turning it into a ring:



Portrait of the array as a doughnut

Here for example is *put* from *ARRAYED_QUEUE*:

```
put (v: G)
        -- Add v as newest item.
    do
        if count + 1 = rep.count then grow end
        rep [in_index] := v
        in_index := (in_index + 1) \\ capacity
        if in_index = 0 then in_index := capacity end
    end
```

The first instruction reallocates the array if we truly run out of space. ("*Don't box us in*".) Procedure *grow* simply calls *resize* on *rep*. When we increment *in_index* in the highlighted instruction, we do it modulo *capacity*; *i* \\ *j* is the integer remainder of *i* by *j*, as *i* // *j* is their integer quotient. The implementation is tuned (see the final **if**...) to an array *rep* indexed from 1 to *capacity*; it is also possible — and a recommended exercise — to see what it gives for an array indexed from zero, and to write the corresponding *remove* implementation.

← *"Automatic resizing", page 375.*

→ *13-E.4, page 434.*

Queues, with proper representation, yield the same performance as stacks:

Operation	**Feature in queue classes**	**Complexity**	**Comments**
Access oldest item	*item*	**O** (1)	
Add item	*put*	**O** (1)	With automatic resizing, occasionally **O** (*count*)
Remove oldest item	*remove*	**O** (1)	

13.13 ITERATING ON DATA STRUCTURES

Container data structures such as the ones we have studied in this chapter are repositories of objects. A common need on such structures is, as we have seen, to apply a certain operation repeatedly to all these objects. This is known as *iterating* on a data structure. Complementing the definition of this term, we also have a name for the mechanism that turns an operation on individual items into an operation on the entire container:

← *"Definition: Iterating", page 397.*

Definition: Iterator

An iterator is a mechanism that can yield, from one or more operations applicable to individual items of a container data structure, an operation applicable to the structure as a whole.

We have already seen many examples of iterating on container structures. They all follow a common pattern: if *your_list* is a *LINKED_LIST* [*T*] or more generally a *LIST* [*T*] (in any implementation of lists) and there is a procedure

```
some_operation (x: T) …
```

the following scheme will iterate the operation over the list

```
from
    your_list.start
invariant
    -- All elements before cursor have been subjected to some_operation
until
    your_list.after
loop
    some_operation (your_list.item)
    your_list.forth
variant
    your_list.count – your_list.index + 1
end
```

Alternatively, you may want to apply the operation only to items that satisfy a certain condition given by a function *your_condition* (*x*: *T*): *BOOLEAN*; the loop body then changes to

```
    if your_condition (your_list.item) then
        some_operation (your_list.item)
    end
```

Other variants of iteration include:

- Apply an operation to all items until the first one that satisfies, or does not satisfy, a certain condition.
- Find out if at least one item, or all items, satisfy a condition.

Isolating general schemes is good; making them reusable, so that one does not have to code them anew each time, is better. You can indeed apply the basic iteration mechanisms, *without writing loops*, by using features such as *do_all* and *do_if* available in all list classes (including *LIST*, *LINKED_LIST* and others cited in this chapter). They capture the preceding loop structures once and for all, so that you may write the last two examples as

```
your_list.do_all (agent your_operation)
your_list.do_if  (agent your_operation, agent your_condition)
```

where **agent** *your_operation* denotes an object that represents the procedure *your_operation* ready to be applied to each item, and **agent** *your_condition* similarly represents the query. To understand the details we have to wait for the general notion of agent in a later chapter.

→ *Chapter 18.*

Even with agents at your disposal, you will have opportunities to write explicit list traversals (*iterations*), using the above schemes as your guides.

13.14 OTHER STRUCTURES

The data structures that we have seen are among the most important in programming, but by no means the only ones. We have already had a glimpse of *trees* (in the form of abstract syntax trees) and will see more of them in the next chapter. A generalization of trees, useful in many applications (for example networking) is the notion of *graph*, directed or not, and *multigraph*.

The following bibliographic section cites books that review the fundamental data structures, usually in connection with fundamental algorithms. In addition a number of textbooks address the "Data Structures and Algorithms" courses offered by most computer science curricula.

13.15 FURTHER READING

Donald E. Knuth: *Fundamental Algorithms*, volumes 1 (*Fundamental Algorithms*) and 3 (*Sorting and Searching*) of *The Art of Computer Programming*, 3rd edition; Addison-Wesley, 1997.

> Widely considered the ultimate reference on algorithms and data structures. Part of a planned seven-volume set of which three have appeared (and some of the fourth in fascicle form).

Knuth (2005)

Aho (2007)

Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

A compact survey of the most important algorithms and data structures. Still an excellent survey of the field, suitable after a first introduction as given in this chapter.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*, second edition, MIT Press, 2001.

An excellent modern textbook on algorithms, giving correctness concerns their due place.

Bertrand Meyer: *Reusable Software*, Prentice Hall, 1994.

A presentation of design principles for building quality reusable libraries, illustrated through the example of EiffelBase.

13.16 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Static typing makes program clearer and catches errors at compile time.
- A generic class has one or more parameters representing types. This provides flexibility and is particularly useful for describing container structures.
- Data structures should support resizing.
- For the consistency of a library, a standard feature naming policy is desirable.
- Abstract complexity estimates the performance of algorithms, independent of hardware choices, by focusing on algorithm behavior for large data sizes and ignoring constant multiplicative and additive factors.
- "Big-O" notation, as in $\mathbf{O}(n^2)$, expresses abstract complexity.
- Arrays provide fast, constant-time access and replacement of items known through their indexes in a given range. Although they can be reallocated, they are not suited for the representation of structures with frequent item insertions and deletions.
- Hash tables generalize arrays to indexes that can be not just integers but almost arbitrary "keys", for example strings, while keeping access and replacement time essentially constant.
- Lists describe sequential structures, and — in linked representations — support insertions and deletions.
- Dispensers let you access, insert and remove elements at only one place. A Last-In First-Out policy yields stacks, First-In First-Out yields queues.
- Stacks are particularly useful to represent nested structures and have applications throughout operating systems and compilers. An array implementation is the most common; a single array can host two stacks.
- Queues are particularly useful in modeling, and in concurrent programming as "buffers". With an array implementation, array indexes should cycle past the upper bound after repeated insertions and deletions.

New vocabulary

Abstract complexity	**Activation record**	**Actual generic parameter**
Array	**Call chain**	**Complexity**
Correctness	**Cursor**	**Dispenser**
Dynamic typing	**FIFO**	**Formal generic parameter**
Generic class	**Generic derivation**	**Genericity**
Hash table	**Heap**	**Linked list**
LIFO	**List**	**Parameter**
Priority queue	**Queue**	**Run-time stack**
Stack	**Static typing**	**Validity**

13-E EXERCISES

13-E.1 Vocabulary

Give a precise definition of all the terms in the above "New vocabulary" list.

13-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise "Concept map", 12-E.2, page 360.

13-E.3 Two in one

Write a class *DOUBLE_STACK* [*G*] implementing two stacks in a single array; you may call the features *put_1*, *put_2*, *remove_1*, *remove_2* and so on. The stacks are of bounded size; make sure to include the right preconditions and invariant clauses.

← Page 383.

13-E.4 Indexing from zero

The implementation we saw for arrayed queues, similar to he EiffelBase class *ARRAYED_QUEUE* but without inheritance, uses an array *rep* indexed from one.

← Page 430.

1 Using for inspiration the implementation of *put* given in the text, write the routine *remove*, and a creation procedure *make* setting up the queue as empty of any items.

2 Rewrite all three routines using an implementation array indexed from zero.

13-E.5 Reversing lists of various kinds

Write a list reversal procedure for two-way lists and arrayed lists, putting it in a class that inherits from the corresponding EiffelBase class: *TWO_WAY_LIST* or *ARRAYED_LIST*.

← See reverse, page 405, for linked lists.

14

Recursion and trees

The cow shown laughing on the Laughing Cow® box holds, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably — I say “presumably” because here my eyesight fails me, I don’t know about yours — holding, as if for earrings, two Laughing Cow® boxes each featuring a cow shown laughing and presumably holding… (you get the idea).

This 1921 advertising gimmick, still doing very well, is an example of a structure defined *recursively*, in the following sense:

www.bel-group.com. Picture credit: page 847.

Recursive definition

A definition for a concept is recursive if it involves one or more instances of the concept itself.

“*Recursion*” — the use of recursive definitions — has applications throughout programming: it yields elegant ways to define *syntax structures*; we will also see recursively defined *data structures* and *routines*.

We may say “*recursive*” as an abbreviation for “recursively defined”: recursive grammar, recursive data structure, recursive routine. But this is only a convention, because we cannot say that a concept or a structure is by itself recursive: all we know is that we can *describe* it recursively, according to the above definition. Any particular notion — even the infinite Laughing Cow structure — may have both recursive and non-recursive definitions.

When proving properties of recursively defined concepts we will use recursive *proofs*, which generalize inductive proofs as performed on integers.

Recursion is *direct* when the definition of A cites an instance of A; it is *indirect* if for $1 \leq i < n$ (for some $n \geq 2$) the definition of every A_i cites an instance of A_{i+1}, and the definition of A_n cites an instance of A_1.

In this chapter we are interested in notions for which a recursive definition is elegant and convenient. The examples include recursive routines, recursive syntax definitions and recursive data structures. We will also get a glimpse of recursive proofs.

One class of recursive data structures, the *tree* in its various guises, appears in many applications and embodies the very idea of recursion. This chapter covers the important case of *binary* trees.

14.1 BASIC EXAMPLES

At this point you may be wondering whether a recursive definition makes any sense at all. How can we define a concept in terms of itself? Does such a definition mean anything at all, or is it just a vicious circle?

You are right to wonder. Not all recursive definitions define anything at all. When you ask for a description of someone and all you get is "*Sarah*? *She is just Sarah*, *what else can I say*?" you are not learning much. So we will have to look for criteria that guarantee that a definition is useful even if recursive.

Before we do this, however, let us convince ourselves in a more pragmatic way by looking at a few typical examples where recursion is obviously useful and seems, just as obviously, to make sense. This will give us a firm belief — little more than a belief indeed, based on hope and a prayer — that recursion is a practically useful way to define grammars, data structures and algorithms. Then it will be time to look for a proper mathematical basis on which to establish the soundness of recursive definitions.

→ *"Making sense of recursion", 14.7, page 473.*

Recursive definitions

With the introduction of genericity, we were able to define a *type* as either:

← *"Definitions: Class type, generically derived, base class", page 370.*

T1 A non-generic class, such as *INTEGER* or *STATION*.

T2 A generic derivation, of the form *C* [*T*], where *C* is a generic class and *T* is a type.

This is a recursive definition; it simply means, using the generic classes *ARRAY* and *LIST*, that valid classes are:

- *INTEGER*, *STATION* and such: non-generic classes, per case T1.
- Through case T2, direct generic derivations: *ARRAY* [*INTEGER*], *LIST* [*STATION*] etc.
- Applying T2 again, recursively: *ARRAY* [*LIST* [*INTEGER*]], *ARRAY* [*ARRAY* [*LIST* [*STATION*]]] and so on: generic derivations at any level of nesting.

You may consider using a similar technique to answer the exercise which, in the first chapter, asked you to define "alphabetical order".

← *1-E.3, page 14.*

Recursively defined grammars

Consider an Eiffel subset with just two kinds of instruction:

← *This discussion was previewed in "Recursive grammars", page 307.*

- Assignment, of the usual form *variable := expression*, but treated here as a terminal, not specified further.
- Conditional, with only a **then** part (no **else**) for simplicity.

A grammar defining this language is:

```
Instruction ≜ Assignment | Conditional
Conditional ≜ if Condition then Instruction end
```

For our immediate purposes Condition is, like Assignment, a terminal. This grammar is recursive, since the definition of Instruction involves Conditional as one of the choices, and Conditional in turn involves Instruction as part of the aggregate. But since there is a non-recursive alternative, Assignment, the grammar productions clearly imply what an instruction may look like:

- Just an assignment.
- A Conditional containing an assignment: **if** *c* **then** *a* **end**.
- The same with any degree of nesting: **if** c_1 **then** **if** c_2 **then** *a* **end** **end**, **if** c_1 **then** **if** c_2 **then** **if** c_3 **then** *a* **end** **end** **end** and so on.

Recursive grammars are indeed an indispensable tool for any language that — like all significant programming languages — supports nested structures.

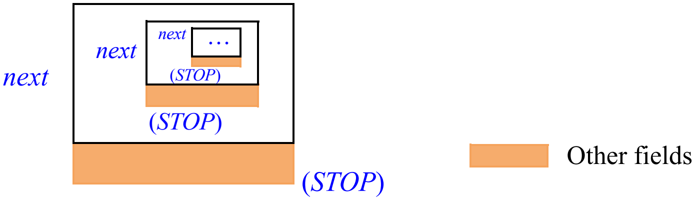
Recursively defined data structures

The class *STOP* represented the notion of stop in a metro line:

← *Page 123.*

```
class STOP create
      …
feature
    next: STOP
            -- Next stop on same line.
      … Other features omitted (see page 123) …
end
```

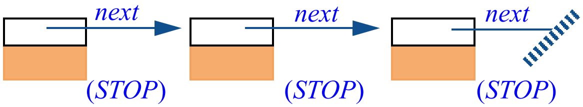
A naïve interpretation would deduce that every instance of *STOP* contains an instance of *STOP*, which itself contains another ad infinitum, as in the Laughing Cow scheme. This would indeed be the case if *STOP* were an expanded type:



Nested fields (not the correct interpretation)

This is impossible, however, and *STOP* is in any case a **reference** type, like any type defined as **class** *X* … with no other qualification. So the real picture is the one originally shown:

← *Page 116.*



A linked line

Recursion in such a data structure definition simply indicates that every instance of the class contains a reference to a potential instance of the same class — "potential" because the reference may be void, as for the last stop in the figure.

In the same chapter we encountered another example of self-referential class definition: a class *PERSON* with an attribute *spouse* of type *PERSON*.

This is a very common case in definitions of useful data structures. From linked lists to *trees* of various kinds (such as the binary trees studied later in this chapter), the definition of a useful object type often includes references to objects of the type being defined, or (indirect recursion) a type that depends on it.

Recursively defined algorithms and routines

The famous Fibonacci sequence, enjoying many beautiful properties and many applications to mathematics and the natural sciences, has the following definition:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \qquad \text{-- For } i > 1$$

Touch of History: Fibonacci's rabbits

Leonardo Fibonacci from Pisa (1170-1250) played a key role in making Indian and Arab mathematics known to the West and, through many contributions of his own, helping to start modern mathematics. He stated like this the problem that leads to his famous sequence (which was already known to Indian mathematicians):

About Fibonacci: www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html; about the sequence: www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Fibonacci.html.

> *A man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if every month each pair begets a new pair, which becomes productive from the second month on?*

Fibonacci

The answer is that the pairs at month i include those already present at month $i - 1$ (no rabbits die), numbering F_{i-1}, plus those begot by pairs already present at month $i - 2$ (since pairs are fertile starting the second month), numbering F_{i-2}. This gives the above formula; successive values are 0, 1, 1, 2, 3, 5, 8 and so on, each the sum of the previous two.

The formula yields a recursive routine computing F_n for any n:

```
fibonacci (n: INTEGER): INTEGER
        -- Element of index n in the Fibonacci sequence.
    require
        non_negative: n >= 0
    do
        if n = 0 then
            Result := 0
        elseif n = 1 then
            Result := 1
        else
            Result := fibonacci (n – 1) + fibonacci (n – 2)
        end
    end
```

Programming Time!
Recursive Fibonacci

Write a small system that includes the above recursive routine and prints out its result. Try it for a few values of *n* — including 12, as in Fibonacci's original riddle — and verify that the results match the expected values.

The function includes two recursive calls, highlighted. That it works at all may look a bit mysterious (that's why it is good to check it for a few values); as you progress through this chapter, the legitimacy of such recursively defined routines should become increasingly convincing.

The principal argument in favor of writing the function this way is that it elegantly matches the original, mathematical definition of the Fibonacci sequence. On further look it is not that exciting, because a non-recursive version is also easy to obtain.

Programming Time!
Non-recursive Fibonacci

Can you write (without first turning the page) a function that computes any Fibonacci number, using a loop rather than recursion?

The following function indeed yields the same result as the above *fibonacci* (try it for a few values too):

```
fibonacci1 (n: INTEGER): INTEGER
        -- Element of index n in the Fibonacci sequence.
        -- (Non-recursive version.)
    require
        positive: n >= 1
    local
        i, previous, second_previous: INTEGER
    do
        from
            i := 1 ; Result := 1
        invariant
            Result = fibonacci (i)
            previous = fibonacci (i – 1)
        until i = n loop
            i := i + 1
            second_previous := previous
            previous := Result
            Result := previous + second_previous
        variant
            n – i
        end
    end
```

For convenience this version assumes $n \geq 1$ rather than $n \geq 0$. Thanks to the initialization rules *previous* starts out as zero, ensuring the initial satisfaction of the invariant since $F_0 = 0$. The variable *second_previous* is set anew in each loop iteration and does not need specific initialization.

This version, just a trifle more remote from the original mathematical definition, is still simple and clear; note in particular the loop invariant (which, however, refers for convenience to the recursive function, which it takes as the official mathematical definition). Some may prefer the recursive version anyway, but this is largely a matter of taste. Depending on the compiler, that version may (as we will see) be less efficient at run time.

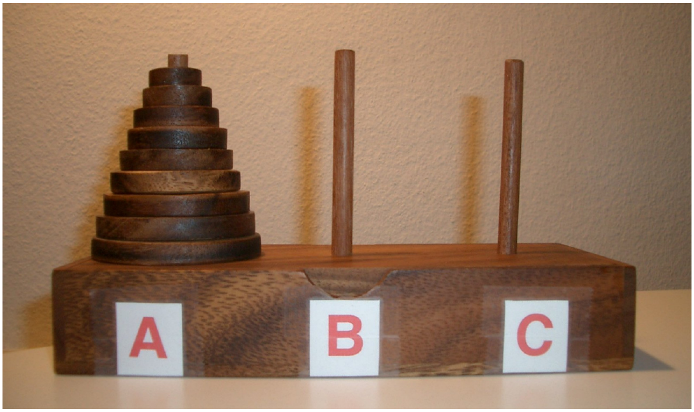
Taste and efficiency aside, if it were only for such examples we would have a hard time convincing ourselves of the indispensability of recursive routines. We need cases in which recursion provides a definite plus, for example because any non-recursive competitor is significantly more abstruse.

Such problems indeed abound. One that concentrates many of the interesting properties of recursion, with the smallest amount of irrelevant detail, arises from a delightful puzzle: the Tower of Hanoi.

14.2 THE TOWER OF HANOI

In the great temple of Benares, under the dome that marks the center of the world, three needles of diamond are set on top of a brass plate. Each needle is a cubit high, and thick as the body of a bee. On one of these needles God strung, at the beginning of ages, sixty-four disks of pure gold. The largest disk rests on the brass and the others, ever smaller, rest over each other all the way to the top. That is the sacred tower of Brahma.

Night and day the priests, following one another on the steps of the altar, work to transfer the tower from the first diamond needle to the third, without deviating from the rules just stated, set by Brahma. When all is over, the tower and the Brahmins will fall, and it will be the end of the worlds.



Tower of Hanoi (or should it be Benares?) with 9 disks, initial state

In spite of its oriental veneer, this story is the creation of the French mathematician Édouard Lucas (signing as "N. Claus de Siam", anagram of "Lucas d'Amiens", after his native city). On a market in Thailand — Siam indeed — I bought the above rendition of his tower. The labels **A**, **B** and **C** are my addition. I will not expand on why I chose a model made of wood rather than diamond, gold and brass, but it is legitimate, since I did have a large suitcase, to ask why it has only nine disks:

Quiz time!
Hanoi tower size

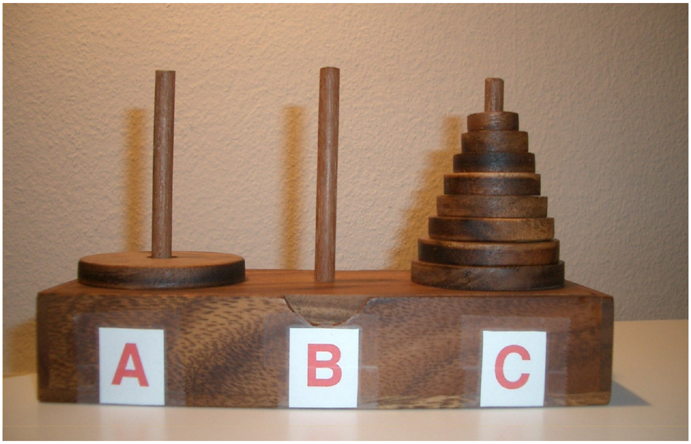
Why do commercially available models of the Towers of Hanoi puzzle have far fewer than 64 disks?

(Hint: the game comes with a small sheet of paper listing a solution to the puzzle, in the form of a sequence of moves: **A** to **C**, **A** to **B** etc.)

To answer this question, we may assess the minimum number H_n of individual "move" operations required — if there is a solution — to transfer n disks from needle **A** to needle **B**, using needle **C** as intermediate storage and following the rules of the game; n is 64 in the original version and 9 for the small model.

We observe that any strategy for moving n disks from **A** to **B** must at some point move the largest disk from **A** to **B**. This is only possible, however, if needle **B** is free of *any* disks at all, and **A** contains only the largest disk, all others having been moved to **C** — since there is no other place for them to go:

Intermediate state



What is the minimum number of moves to reach this intermediate situation? We must have transferred $n - 1$ disks (all but the largest) from **A** to **C**, using **B** as intermediate storage; the largest disk, which must stay on **A**, plays no role in this operation. The problem is symmetric between **B** and **C**; so the minimum number of moves to achieve the intermediate situation is H_{n-1}.

Once we have reached that situation, we must move the largest disk from **A** to **B**; it remains then to transfer the $n - 1$ smaller disks from **C** to **B**. In all, the minimum number of moves H_n for transferring n disks, for $n > 0$, is

$$\boxed{H_n = 2 * H_{n-1} + 1}$$

(H_{n-1} moves to transfer $n - 1$ disks from **A** to **C**, one move to take the largest disk from **A** to **B**, and H_{n-1} again to transfer the $n - 1$ smaller disks from **A** to **C**). Since $H_0 = 0$, this gives

$$\boxed{H_n = 2^n - 1}$$

and, as a consequence, the answer to our quiz: remembering that 2^{10} (that is, 1024) is over 10^3, we note that 2^{64} is over $1.5 * 10^{19}$; that's a lot of moves.

A year is around 30 million seconds. At one second per move — very efficient priests — the world will collapse in about 500 billion years, over 30 times the estimated age of the universe. As to the paper for printing the solution to a 64-disk game, it would require cutting down the forests of a few planets.

2^{13} sheets per tree (tinyurl.com/6azaht); 2^{10} moves per page (very small print); double-sided since we are environmentally conscious; maybe 400 billion (over 2^{38}) usable trees on earth (tinyurl.com/yfppyd): adding three similar planets will get us started.

This reasoning for the evaluation of H_n was *constructive*, in the sense that it also gives us a **practical strategy** for moving n disks (for $n > 0$) from **A** to **B** using **C** as intermediate storage:

- Move $n - 1$ disks from **A** to **C**, using **B** as intermediate storage, and respecting the rules of the game.
- Then **B** will be empty of any disk, and **A** will only have the largest disk; transfer that disk from **A** to **B**. This respects the rules of the game since we are moving a single disk, from the top of a needle, to an empty needle.
- Then move $n - 1$ disks from **C** to **B**, using **A** as intermediate storage, respecting the rules of the game; **B** has one disk, but it will not cause any violation of the rules of the game since it is larger than all the ones we want to transfer.

This strategy turns the number of moves $H_n = 2^n - 1$ from a theoretical minimum into a practically achievable goal. We may express it as a recursive routine, part of a class *NEEDLES*:

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
        -- Transfer n disks from source to target, using other as
        -- intermediate storage, according to the rules of the
        -- Tower of Hanoi puzzle.
    require
        non_negative: n >= 0
        different1: source /= target
        different2: target /= other
        different3: source /= other
    do
        if n > 0 then
            hanoi (n–1, source, other, target)
            move (source, target)
            hanoi (n–1, other, target, source)
        end
    end
```

The discussion of contracts for recursive routines will add other precondition clauses and a postcondition.

→ "Contracts for recursive routines", 14.8, page 485.

By convention, we represent the needles as characters: 'A', 'B' and 'C'. Another convention for this chapter (already used in previous examples) is to highlight recursive branches; *hanoi* contains two such calls.

The basic operation *move* (*source*, *target*) moves a single disk, the top one on needle *source*, to needle *target*; its precondition is that there is at least one disk on *source*, and that on *target* either there is no disk or the top disk is larger than the top disk on *source*. If you have access to the wireless network of the Great Temple of Benares you can program *move* to send an instant message to the cell phone of the appropriate priest or an email to her Blackberry, directing her to move a disk from *source* to *target*. For the rest of us you can write *move* as a procedure that displays a one-disk-move instruction in the console:

```
move (source, target: CHARACTER)
        -- Prescribe move from source to target.
    do
        io.put_character (source)
        io.put_string (" to ")
        io.put_character (target)
        io.put_new_line
    end
```

Programming Time!
The Tower of Hanoi

Write a system with a root class *NEEDLES* including the procedures *hanoi* and *move* as shown. Try it for a few values of *n*.

For example executing the call

```
hanoi (4, 'A', 'B', 'C')
```

will print out the sequence of fifteen ($2^4 - 1$) instructions

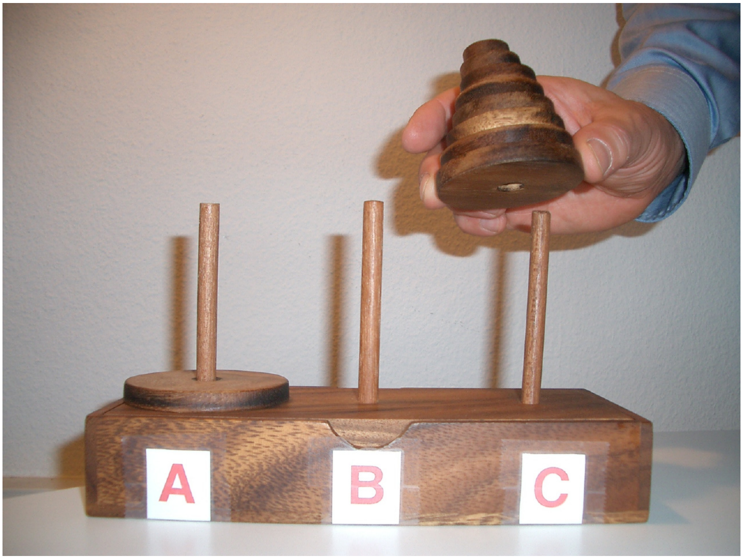
```
A to C          B to C          B to A
A to B          A to C          C to B
C to B          A to B          A to C
A to C          C to B          A to B
B to A          C to A          C to B
```

Shown here split into three columns; read it column by column, top to bottom in each column. The move of the biggest disk has been highlighted.

which indeed moves four disks successfully from **A** to **B**, respecting the rules of the game.

One way to look at the recursive solution — procedure *hanoi* — is that it works as if we were permitted to move the top *n*–1 disks all at once to a needle that has either no disk, or the biggest disk only. In that case we would start by performing this operation from *source* to *other* (here **A** to **C**):

Fictitious initial global move



Then we would move the biggest disk from **A** to **B**, our final *target*; this single-disk move is clearly legal since there is nothing on **B**. Finally we would again perform a global move of *n*–1 disks from **C**, where we have parked them, to **B**, which is OK because they are in order and the largest of them is smaller than the disk now on **B**.

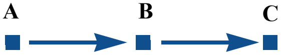
Of course this is a fiction since we are only permitted to move one disk at a time, but to move *n*–1 disks we may simply apply the same technique recursively, knowing that the target needle is either empty or occupied by a disk larger than all those we manipulate in this recursive application. If $n = 0$, we have nothing to do.

Do not be misled by the apparent frivolity of the Tower of Hanoi example. The solution serves as a model for many recursive programs with important practical applications. The simplicity of the algorithm, resulting from the use of two recursive calls, makes it an ideal testbed to study the properties of recursive algorithms, as we will do when we return to it later in this chapter.

14.3 RECURSION AS A PROBLEM-SOLVING STRATEGY

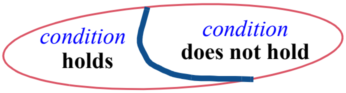
In earlier chapters we saw control structures as problem-solving techniques:

- A **compound** (sequence) solution means "*I know someone who can get me from here to B and someone else who can get me from B to C, so let me ask them one then the other and that will get me to C*".



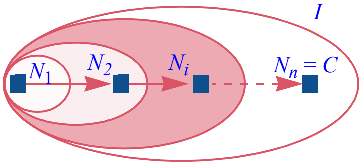
(*Figure from page* 147.)

- A **conditional** solution means "*I know someone who can solve the problem in one case and someone else for the other possible case, so let me ask them separately*".



(*Figure from page* 174.)

- A **loop** solution means "I *do not know how to get to C, but I know a region I (the invariant) that contains C, someone (the initialization) to take me into I, and someone else (the body) who whenever I am in I and not yet in C can take me closer to C, decreasing the distance (the variant) in such a way that I will need her only a finite number of times; so let me ask my first friend once to get into I, then bug my other friend as long as I have not reached C yet*".



(*Figure from page* 155.)

- A **routine** solution means "*I know someone who has solved this problem in the general case, so let me just phrase my special problem in his terms and ask him to solve it for me*".

What about a recursive solution? Whom do I ask?

I ask myself.

Possibly several times! (As in the *Hanoi* case and many to follow.)

Why rely on someone else when I trust myself so much more? (At least I think I do.)

By now we know that this strategy is not as silly as it might sound at first. I ask myself to solve the same problem, but on a **subset** of the original data, or several such subsets. Then I may be able to pull it off, if I have a way to extend these partial solutions into a solution to the entire problem.

Such is the idea of recursion viewed as a general problem-solving strategy. It is related to some of the earlier strategies:

- Recursion resembles the *routine* strategy, since it relies on an existing solution, but in this case we use a solution to the *same problem* — not only that, the *same solution* to that problem: the solution that we are currently building and that we just pretend, by a leap of faith, already exists.

← *Chapter 8.*

- Recursion also shares properties with a *loop* solution: both techniques approximate the solution to the whole problem by solutions covering part of the data. But recursion is more general, since each step may combine more than one such partial solution. Later in this chapter we will have the opportunity of comparing the loop and recursion strategies in detail.

← *"The loop strategy", page 155.*

← *"From loops to recursion", 14.6, page 471.*

14.4 BINARY TREES

If the Tower of Hanoi solution is the quintessential recursive routine, the binary tree is the quintessential recursive data structure. We may define it as follows:

Definition: binary tree

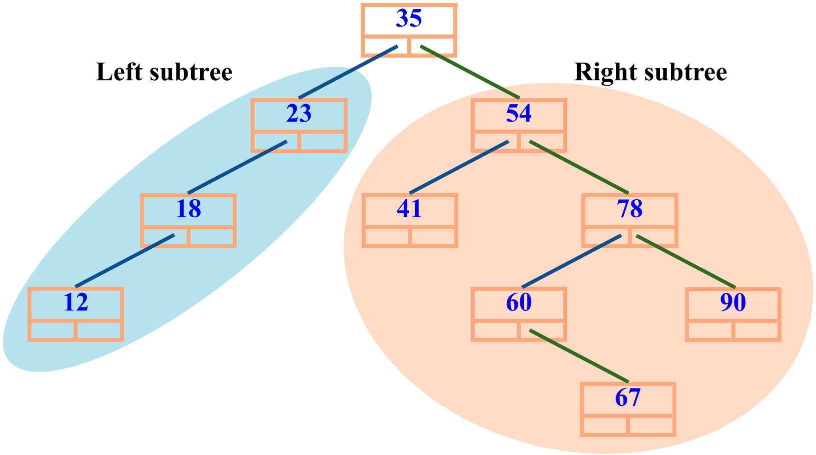
A binary tree over *G*, for an arbitrary data type *G*, is a finite set of items called **nodes**, each containing a value of type *G*, such that the nodes, if any, are divided into three disjoint parts:

- A single node, called the **root** of the binary tree.
- (Recursively) two binary trees over *G*, called the *left subtree* and *right subtree*.

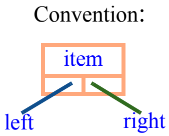
It is easy to express this as a class skeleton, with no routines yet:

```
class BINARY_TREE [G] feature
    item: G
    left, right: BINARY_TREE [G]
end
```

where a void reference indicates an empty binary tree. We may illustrate a binary tree — here over *INTEGER* — as follows:

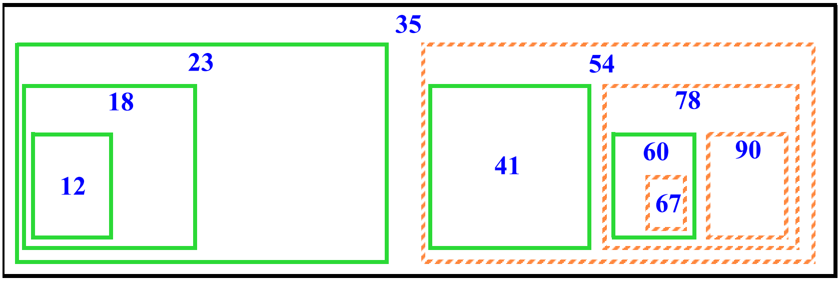


***A binary tree* (*“branching” representation*)**



This “branching” form is the most common style of representing a binary tree, but not the only one; as in the case of abstract syntax trees, we might opt for a *nested* representation, which here would look like the following.

← *“Nesting and the syntax structure”, page 40.*



A binary tree in nested representation

Convention:

□ Left subtree

□ Right subtree

The definition explicitly allows a binary tree to be empty (“the nodes, if any”). Without this, of course, the recursive definition would lead to an infinite structure, whereas our binary trees are, as the definition also prescribes, finite.

If not empty, a binary tree always has a root, and may have: no subtree; a left subtree only; a right subtree only; or both.

Any node *n* of a binary tree *B* itself defines a binary tree B_n. The association is easy to see in either of the last two figures: for the node labeled **35**, B_n is the full tree; for **23** it is the left subtree; for **54**, the right subtree; for **78**, the tree rooted at that node (right subtree of the right subtree); and so on. This allows us to talk about the left and right subtrees of a *node* — meaning, of its associated subtree. We can make the association formal through another example of recursive definition, closely following the structure of the definition of binary trees:

Definition: Tree associated with a node

Any node *n* of a binary tree *B* defines a binary tree B_n as follows:

- If *n* is the root of *B*, then B_n is simply *B*.
- Otherwise we know from the preceding definition that *n* is in one of the two subtrees of *B*. If *B’* is that subtree, we define B_n as B'_n (the node associated with *n*, recursively, in the corresponding subtree).

A recursive routine on a recursive data structure

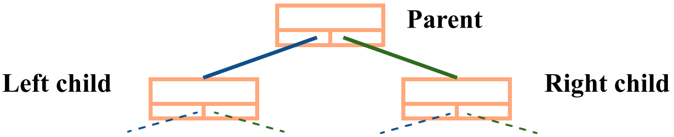
Many routines of a class that defines a data structure recursively will follow the definition’s recursive structure. A simple example is a routine computing the number of nodes in a binary tree. The node count of an empty tree is zero; the node count of a non-empty tree is one — corresponding to the root — plus (recursively) the node counts of the left and right subtrees, if any. We may turn this observation into a recursive function of the class *BINARY_TREE*:

```
    count: INTEGER
            -- Number of nodes.
        do
            Result := 1
            if left /= Void   then Result := Result + left.count   end
            if right /= Void  then Result := Result + right.count  end
        end
```

Note the similarity of the recursive structure to procedure *Hanoi*.

Children and parents

The **children** of a node — nodes themselves — are the root nodes of its left and right subtrees:



A binary tree (“branching” representation)

If *C* is a child of *B*, then *B* is a **parent** of *C*. We may say more precisely that *B* is “*the*” parent of *C* thanks to the following result:

> ***Theorem*: Single Parent**
>
> Every node in a binary tree has exactly one parent, except for the root which has no parent.

The theorem seems obvious from the picture, but we have to prove it; this gives us an opportunity to encounter *recursive proofs*.

Recursive proofs

The recursive proof of the Single Parent theorem mirrors once more the structure of the recursive definition of binary trees.

If a binary tree *BT* is empty, the theorem trivially holds. Otherwise *BT* consists of a root and two disjoint binary trees, of which we assume — this is the “recursion hypothesis” — that they both satisfy the theorem. It follows from the definitions of “binary tree”, “child” and “parent” that a node *C* may have a parent *P* in *BT* only through one of the following three ways:

P1 *P* is the root of *BT*, and *C* is the root of either its left or right subtree.

P2 They both belong to the left subtree, and *P* is the parent of *C* in that subtree.

P3 They both belong to the right subtree, and *P* is the parent of *C* in that subtree.

In case P1, *C* has, from the recursion hypothesis, no parent in its subtree; so it has one parent, the root, in *BT* as a whole. In cases P2 and P3, again by the recursion hypothesis, *P* was the single parent of *C* in their respective subtree, and this is still the case in the whole tree.

Any node *C* other than the root falls into one of these three cases, and hence has exactly one parent. In none of these cases can *C* be the root which, as a consequence, has no parent. This completes the proof.

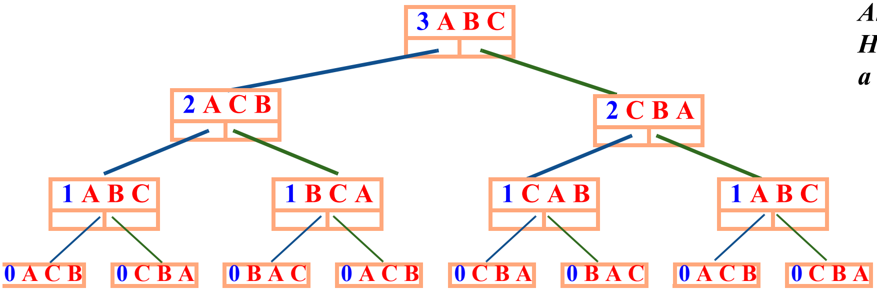
Recursive proofs of this kind are useful when you need to establish that a certain property holds for all instances of a recursively defined concept. The structure of the proof follows the structure of the definition:

- For any non-recursive case of the definition, you must prove the property directly. (In the example the non-recursive case is an empty tree.)
- A case of the definition is recursive if it defines a new instance of the concept in terms of existing instances. For those cases you may assume that the property holds of these instances (this is the *recursion hypothesis*) to prove that it holds of the new one.

This technique applies to recursively defined concepts in general. We will see its application to recursively defined routines such as *hanoi*.

A binary tree of executions

An interesting example of a binary tree is the one we obtain if we model an execution of the *hanoi* procedure, for example with three disks on needles **A**, **B**, **C**. Each node contains the arguments to the given call; the left and right subtrees correspond to the first and second recursive calls.



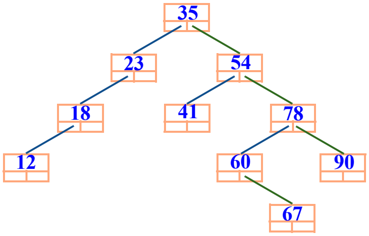
An execution of Hanoi viewed as a binary tree

By adding the *move* operations you may reconstruct the sequence of operations; we will see this formally below.

→ *Page 454.*

This example illustrates the connection between recursive algorithms and recursive data structures. For routines that have a variable number of recursive calls, rather than exactly two as *hanoi*, the execution would be modeled by a general tree rather than a binary tree.

More binary tree properties and terminology



(*From the figure on page 447.*)

As noted, a node of a binary tree may have:

- Both a left child and a right child, like the top node, labeled **35**, of our example.
- Only a left child, like all the nodes of the left subtree, labeled **23**, **18**, **12**.
- Only a right child, like the node labeled **60**.
- No child, in which case it is called a **leaf**. In the example the leaves are labeled **12**, **41**, **67** and **90**.

We define an **upward path** in a binary tree as a sequence of zero or more nodes, where any node in the sequence is the parent of the previous one if any. In our example, the nodes of labels **60**, **78**, **54** form an upward path. We have the following property, a consequence of the Single Parent theorem:

Theorem: Root Path

From any node of a binary tree, there is a single upward path to the root.

Proof: consider an arbitrary node *C* and the upward path starting at *C* and obtained by adding the parent of each node on the path, as long as there is one; the Single Parent theorem ensures that this path is uniquely defined. If the path is finite, its last element is the root, since any other node has a parent and hence would allow us to add one more element to the path; so to prove the theorem it suffices to show that all paths are finite.

The only way for a path to be infinite, since our binary trees are finite sets of nodes, would be to include a **cycle**, that is to say if a node *n* appeared twice (and hence an infinite number of times). This means the path includes a subsequence of the form *n* … *n*. But then *n* appears in its own left or right subtree, which is impossible from the definition of binary trees.

Considering downward rather than upward paths gives an immediate consequence of the preceding theorem:

Theorem: Downward Path

For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of *left* and *right* links.

The **height** of a binary tree is the maximum number of nodes on a downward path from the root to a leaf (or the reverse upward path). In the example (see figure above) the height is 5, obtained through the path from the root to the leaf labeled **67**.

It is possible to define this notion recursively, following again the recursive structure of the definition of binary trees: the height of an empty tree is zero; the height of a non-empty tree is one plus the maximum of (recursively) the heights of its two subtrees. We may add the corresponding function to class *BINARY_TREE*:

```
height: INTEGER
        -- Maximum number of nodes on a downward path.
    local
        lh, rh: INTEGER
    do
        if left /= Void   then lh := left.height   end
        if right /= Void then rh := right.height end
        Result := 1 + lh.max (rh)
    end
```

x.max (y) is the maximum of x and y.

This adapts the recursive definition to the convention used by the class, which only considers non-empty binary trees, although either or both subtrees, *left* and *right*, may be empty. Note again the similarity to *hanoi*.

Binary tree operations

Class *BINARY_TREE* as given so far has only three features, all of them queries: *item*, *left* and *right*. We may add a creation procedure

← Page 447.

```
make (x: G)
        -- Initialize with item value x.
    do
        item := x
    ensure
        set: item = x
    end
```

and commands for changing the subtrees and the root value:

```
add_left (x: G)
        -- Create left child of value x.
    require
        no_left_child_behind: left = Void
    do
        create left.make (x)
    end

add_right … Same model as add_left …

replace (x: G)
        -- Set root value to x.
    do item := x end
```

Note the precondition, which prevents overwriting an existing child. It is possible to add procedures put_left and put_right, which replace an existing child and do not have this precondition.

In practice it is convenient to specify *replace* as an assigner command for the corresponding query, by changing the declarations of this query to

```
    item: G assign replace
```

making it possible to write *bt*.*item* := *x* rather than *bt*.*put* (*x*).

← *"Bracket notation and assigner commands", page 384.*

Traversals

Being defined recursively, binary trees lead, not surprisingly, to many recursive routines. Function *height* was one; here is another. Assume that you are requested to print all the *item* values associated with nodes of the tree. The following procedure, to be added to the class, does the job:

```
print_all
        -- Print all node values.
    do
        if left /= Void then print_all (left) end
        print (item)
        if right /= Void then print_all (right) end
    end
```

This uses the procedure *print* (available to all classes through their common ancestor *ANY*) which prints a suitable representation of a value of any type; here the type is *G*, the generic parameter in *BINARY_TREE* [*G*].

→ *"Overall inheritance structure", 16.10, page 586.*

Remarkably, the structure of *print_all* is identical to the structure of *hanoi*.

Although the business of *print_all* is to print every node item, the algorithm scheme is independent of the specific operation, here *print*, that we perform on *item*. The procedure is an example of a binary tree **traversal**: an algorithm that performs a certain operation once on every element of a data structure, in a precisely specified order. Traversal is a case of *iteration*.

← *"Definition: Iterating", page 397. For further study see "Agents for iteration", 17.3, page 627.*

For binary trees, three traversal orders are often useful:

Binary tree traversal orders

- **Inorder**: traverse left subtree, visit root, traverse right subtree.
- **Preorder**: visit root, traverse left, traverse right.
- **Postorder**: traverse left, traverse right, visit root.

In these definitions, "visit" means performing the individual node operation, such as *print* in the *print_all* example; " traverse " means a recursive application of the algorithm to a subtree, or no action if the subtree is empty.

Preorder and other traversals that always go as deep as possible into a subtree before trying other nodes are known as *depth-first*.

The procedure *print_all* is an illustration of inorder traversal. We may easily express the other two variants in the same recursive form; for example, a routine *post* for postorder traversal will have the routine body

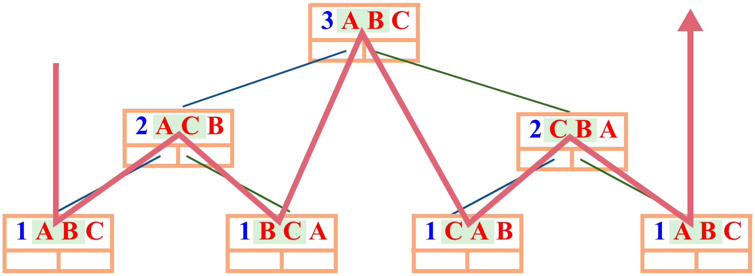
```
if left /= Void  then  post (left)    end
if right /= Void then  post (right)   end
visit (item)
```

where *visit* is the node operation, such as *print*.

> In the quest for software reuse, it is undesirable to write a different routine for variants of a given traversal scheme just because the *visit* operation changes. To avoid this, we may use the operation itself as an argument to the traversal routine. This will be possible through the notion of **agent** in a later chapter.

→ “Writing an iterator”, page 631.

As another illustration of inorder traversal, consider again the binary tree of executions of *hanoi*, for $n = 3$, with the nodes at level 0 omitted since nothing interesting happens there:



Hanoi execution as inorder traversal
(From the figure on page 450)

Procedure *hanoi* is the mother of all inorder traversals: traverse the left subtree if any; visit the root, performing *move* (*source*, *target*), as highlighted for each node (*source* and *target* are the first two needle arguments); traverse the right subtree if any. The inorder traversal, as illustrated by the bold line, produces the required sequence of moves **A B**, **A C**, **B C**, **A B**, **C A**, **C B**, **A B**.

Binary search trees

For a general binary tree, procedure *print_all*, implementing inorder traversal, prints the node values in an arbitrary order. For the order to be significant, we must move on from binary trees to binary *search* trees.

The set *G* over which a general binary tree is defined can be any set. For binary search trees, we assume that *G* is equipped with a **total order relation** enabling us to compare two arbitrary elements of *G* through the boolean

→ We will learn more about total orders in the study of topological sort: “Total orders”, page 514.

expression $a < b$, such that exactly one of $a < b$, $b < a$ and $a \sim b$ (object equality) is true. Examples of such sets include *INTEGER* and *REAL*, with the usual $<$ relation, but G could be any other set on which we know a total order.

As usual we write $a <= b$ for $(a < b)$ **or** $(a \sim b)$, and $a > b$ for $b < a$. Over such totally ordered sets we may define binary search trees:

The EiffelBase class is BINARY_SEARCH_TREE.

Definition: binary search tree

A binary search tree over a totally ordered set G is a binary tree over G such that, for any subtree of root *item* value r:

- The item value le of any node in the left subtree satisfies $le < r$.
- The item value ri of any node in the right subtree satisfies $ri > r$.

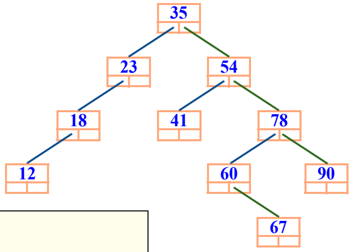
The node values in the left subtree are less than the value for the root, and those in the right subtree are greater; this property must apply not only to the tree as a whole but also, recursively, to any of its immediate or indirect subtrees. We will call it the **Binary Search Tree Invariant**.

This definition implies that all the *item* values of the tree's node are different. We will use this convention for simplicity. It is also possible to accept duplications; then the conditions in the definitions become $le <= r$ and $r <= ri$. An exercise asks you accordingly to adapt the binary search tree algorithms that we are going to see.

→ Exercise 14-E.3, page 500.

Our example binary tree of integers is a binary search tree: all the values in the left subtree are less than the root value, **35**, all those in the right subtree are greater, and the same properties hold recursively in every subtree.

The procedure *print_all*, applied to a binary search tree, will print all the node items in order, from smallest to greatest.



Programming Time! Printing values in order

Using the procedures given so far, write a program that builds the example tree, then prints the node items using *print_all*. Check that the values are in order.

Performance

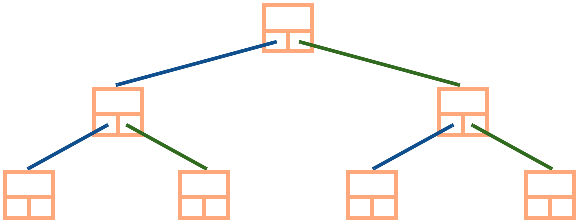
Let us look more closely at why binary search trees are useful as container structures — a potential competitor to hash tables. Indeed they usually provide much better performance than sequential lists. Assuming random data, a sequential list provides us, n being the number of items, with

← Second performance table on page 407.

- $\mathbf{O}\ (1)$ insertion (if we keep the items in the order of insertion).
- $\mathbf{O}\ (n)$ search.

With a binary search tree, both operations can be **O** $(\log n)$, much better than **O** (n) for large n. (Remember that in big-**O** notation it does not matter what base we choose for the logarithms.) Here is the analysis for a **full** binary tree, that is to say one in which both subtrees of any given node have exactly the same height h:

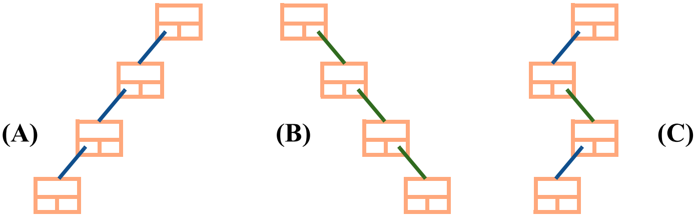
A full binary tree



It is clear, by induction on h, that the number of nodes n in a full tree of height h is $2^h - 1$ (in the above figure, h is 3 and n is 7). This implies that for a given number of nodes n the height is $\log_2 (n + 1)$, which is **O** $(\log n)$. In a full tree, both a search and an insertion — using algorithms given below, which you can already guess — will start from the root and follow a downward path to a leaf, taking **O** $(\log n)$ time. This is the major attraction of binary search trees.

← "Theorem: Downward Path", page 451.

Of course most practical binary trees are not full; if you are out of luck with the order of insertion, the performance can be as bad as with sequential lists, **O** (n) — with added storage costs since each node has both a *left* field and a *right* field where a linked list cell has just one. The following figure shows such cases: insertions in descending order **(A)**, ascending order **(B)**, greatest then smallest then second greatest and so on **(C)**.



Some binary search tree schemes causing O (n) behavior

With a random enough order of insertions, however, the binary search tree will remain sufficiently close to full to ensure **O** $(\log n)$ behavior. You can actually *guarantee* **O** $(\log n)$ insertions, searches and deletions by using the **AVL** or "red-black" variants of binary search trees, which remain near-full.

On these techniques, see the bibliographic references of the previous chapter, for example Cormen et al. (page 433).

Inserting, searching, deleting

Here is a recursive routine for searching a binary search tree (this routine and the following ones are to be added to the binary search tree class):

```
has (x: G): BOOLEAN
        -- Does x appear in any node?
    require
        argument_exists: x /= Void
    do
        if x ~ item then
            Result := True
        elseif x < item then
            Result := (left /=Void) and then left.has (x)
        else -- x > item
            Result := (right /= Void) and then right.has (x)
        end
    end
```

~ is object equality.

The algorithm is **O** (*h*) where *h* is the height of the tree, meaning **O** (log *n*) for full or near-full trees.

In this case there is a reasonably simple non-recursive version, using a loop:

```
has1 (x: G): BOOLEAN
        -- Does x appear in any node?
    require
        argument_exists: x /= Void
    local
        node: BINARY_TREE [G]
    do
        from
            node := Current
        until
            Result or node = Void
        invariant
            -- x does not appear above node on downward path from root
        loop
            if x < item then
                node := left
            elseif x > item then
                node := right
            else
                Result := True
            end
        variant
            -- (Height of tree) – (Length of path from root to node)
        end
    end
```

← The variant and invariant are pseudocode; see "Touch of Style: Highlighting pseudocode", page 109.

For *inserting* an element, we may use the following recursive procedure:

```
put (x: G)
        -- Insert x if not already present.
    require
        argument_exists: x /= Void
    do
        if x < item then
            if left = Void then
                add_left (x)
            else
                left.put (x)
            end
        elseif x > item then
            if right = Void then
                add_right (x)
            else
                right.put (x)
            end
        end
    end
```

← *About add_left and add_right see page 452.*

The absence of an **else** clause for the outermost **if** reflects the decision to ban duplicate information. As a consequence, a call to *put* with an already present value will have no effect. This is correct behavior ("*not a bug but a feature*"), since the header comment is clear. Some users might, however, prefer a different API with a precondition stating **not** *has* (*x*).

← *See page 455.*

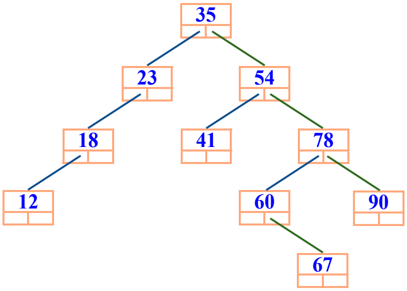
The non-recursive version is left as an exercise.

→ *14-E.5, page 502.*

The next natural question is how to write a deletion procedure *remove* (*x*: *G*). This is less simple because we cannot just remove the node containing *x* (unless it is a leaf and not the root, in which case we make the corresponding *left* or *right* reference void); we also cannot leave an arbitrary value there since it would destroy the Binary Search Tree Invariant.

More precisely we could put a special boolean attribute in every node, indicating whether the *item* value is meaningful, but that makes things too complicated, wastes space and affects the other algorithms.

What we should do is reorganize the node values, moving up some of those found in subtrees of the node where we find *x* to reestablish the Binary Search Tree Invariant.



(*From the figure on page 447*.)

In the example binary search tree, a call *remove* (35), affecting the value in the root node, might either:

- Move up all the values from the left subtree (where each node has a single child, on the left).

- Move up the value in the right child, **54**, then recursively apply a similar choice to move values up in one of its subtrees.

Like search and insertion, the process should be **O** (h) where h is the height of the tree, in favorable cases.

The deletion procedure is left as an exercise; I suggest you try your hand at it now, following the inspiration of the preceding routines:

Programming Time!
Deletion in a binary search tree

Write a procedure *remove* (x: G) that removes from a binary search tree the node, if any, of item value x, preserving the Binary Search tree Invariant.

14.5 BACKTRACKING AND ALPHA-BETA

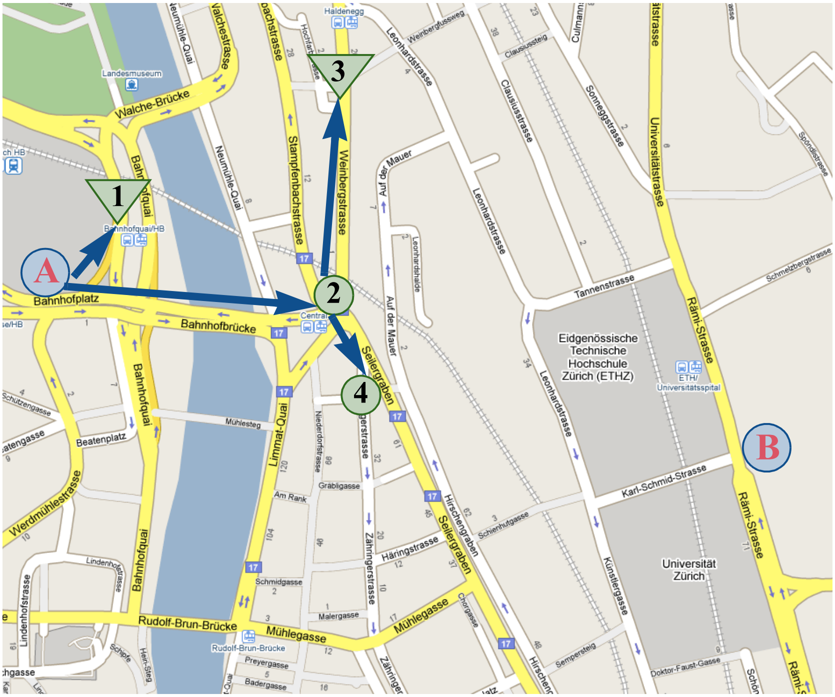
Before we explore the theoretical basis of recursive programming, it is useful to look into one more application, or rather a whole class of applications, for which recursion is the natural tool: backtracking algorithms.

The name carries the basic idea: a backtracking algorithm looks for a solution to a certain problem by trying successive paths and, whenever a path reaches a dead end, backing up to a previous path from which not all possible continuations have been tried. The process ends successfully if it finds a path that yields a solution, and otherwise fails after exhausting all possible paths, or hitting a preset termination condition such as a search time limit.

A problem may be amenable to backtracking if every potential solution can be defined as a sequence of choices.

The plight of the shy tourist

You may have applied backtracking, as a last resort, to reach a travel destination. Say you are at position A (Zurich main station) and want to get to B (between the main buildings of ETH and the University of Zurich):



Trying and backtracking

Not having a map and too shy to ask for directions, you are reduced to trying out streets and checking, after each try, if you have arrived (you do have a photo of the destination). You know that the destination is towards the East; so, to avoid cycles, you ignore any westward street segment.

At each step you try street segments starting from the north, clockwise: the first attempt takes you to position **1**. You realize that it is not your destination; since the only possible segment from there goes west, this is a dead end: you backtrack to A and try the next choice from there, taking you to **2**. From there you try **3**, again a dead end as all segments go west. You backtrack to the previous position, **2**.

If all valid (non-westward) positions had been tried, **2** would be a dead-end too, and you would have to go back to A, but there remains an unexplored choice, leading to **4**.

The process continues like this; you can complete the itinerary on the map above. While not necessarily the best technique for traveling, it is sometimes the only possible one, and it is representative of the general trial-and-error scheme of backtrack programming. This scheme can be expressed as a recursive routine:

```
find (p: PATH): PATH
        -- Solution, if any, starting at p.
    require
        meaningful: p /= Void
    local
        c: LIST [CHOICE]
    do
        if p.is_solution then
            Result := p
        else
            c := p.choices
            from c.start until
                (Result /= Void) or c.after
            loop
                Result := find (p + c)
                c.forth
            end
        end
    end
```

This uses the following conventions: the choices at every step are described by a type *CHOICE* (in many cases you may be able to use just *INTEGER*); there is also a type *PATH*, but a path is simply a sequence of choices, and *p + c* is the path obtained by appending *c* to *p*. We identify a solution with the path that leads to it, so *find* returns a *PATH*; by convention that result is void if *find* finds no solution. To know if a path yields a solution we have the query *is_solution*. The list of choices available from *p* — an empty list if *p* is a dead end — is *p.choices*.

To obtain the solution to a problem it suffices to use *find* (*p0*) where *p0* is an initial, empty path.

As usual, **Result** is initialized to **Void**, so that if in a call to *find* (*p*) none of the recursive calls on possible extensions *p + c* of *p* yields a solution — in particular, if there are no such extensions as *p.choices* is empty — the loop will terminate with *c.after*; then *find* (*p*) itself will return **Void**. If this was the original call *find* (*p0*), the process terminates without producing a solution; but if not, it is a recursively triggered call, and the parent call will simply resume by trying the next remaining choice if any (or returning **Void** too if none are left).

If, on the other hand, the call finds *p* to be a solution, it returns *p* as its result, and all the callers up the chain will return it as well, terminating their list traversals through the **Result /= Void** clause of the exit condition.

Recursion is clearly essential to handle such a scheme. It is a natural way to express the trial-and-error nature of backtracking algorithms; the machinery of recursion takes care of everything. To realize its contribution, imagine for a second how you would program such a scheme without recursion, keeping track of previously visited positions. (I am not suggesting you actually write out the full non-recursive version, at least not until you have read about derecursification techniques further in this chapter.)

→ *"Implementation of recursive routines", 14.9, page 486.*

The later discussion also shows how to improve the efficiency of the given algorithm by removing unnecessary bookkeeping. For example it is not really necessary to pass the path *p* as an explicit argument, taking up space on the call stack; *p* can instead be an attribute, if we add $p := p + x$ before the recursive call and *p* := *p*.*head* after it (where *head* yields a copy of a sequence with its last element removed). We will develop a general framework allowing us to carry out such optimizations safely.

→ *"Preserving and restoring the context", page 488.*

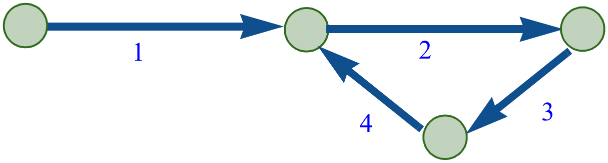
Getting backtracking right

The general backtracking scheme requires some tuning for practical use. First, as given, it is not guaranteed to terminate, as it could go on exploring ever longer paths. To ensure that any execution terminates, you should either:

- Have a guarantee (from the problem domain) that there are no infinite paths; in other words, that repeatedly extending any path will eventually yield a path with an empty *choices* list.
- Define a maximum path length and adapt the algorithm so that it treats any path as a dead-end when it reaches that limit. Instead of the path length you may also limit the computation time. Either variant is a simple change to the preceding algorithm.

→ *Exercise "Backtracking curtailed", 14-E.8, page 503.*

In addition, a practical implementation can usually detect that a path is equivalent to another; for example, with the situation pictured



Path with a cycle

the paths [1 2 3 4], [1 2 3 4 2 3 4], [1 2 3 4 2 2 3 4 2 3 4] etc. are all equivalent. The example of finding an itinerary to a destination avoided this problem through an expedient — never go west, young man — but this is not a generalizable solution. To avoid running into such cycles, the algorithm should keep a list of previously visited positions, and ignore any path leading to such a position.

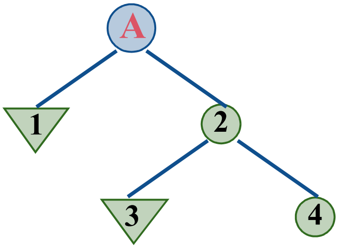
→ Exercise "Cycles despised", 14-E.9, page 503.

Backtracking and trees

Any problem that lends itself to a backtracking solution also lends itself to modeling by a tree. In establishing this correspondence, we use trees where a node may have any number of children, generalizing the concepts defined earlier for binary trees. A path in the tree (sequence of nodes) corresponds to a path in the backtracking algorithm (sequence of choices); the tree of the itinerary example, limited to the choices that we tried, is:

← "Binary trees", 14.4, page 447.

→ "Trying and backtracking", page 460.



Backtrack tree

We can represent the entire town map in this way: nodes for locations, connected by edges representing street segment. The result is a *graph*. A graph only yields a tree if it has no cycles. Here this is not the case, but we can get a tree, called a *spanning tree* for the graph, containing all of its nodes and some of its edges, through one of the techniques mentioned earlier: using a cycle-avoiding convention such as never going west, or building paths from a root and excluding any edge that leads to a previously encountered node. The above tree is a spanning tree for the part of our example that includes nodes A, **1**, **2**, **3** and **4**.

With this tree representation of the problem:

- A solution is a node that satisfies the given criterion (the property earlier called *is_solution*, adapted to apply to nodes rather than paths).
- An execution of the algorithm is simply a **preorder** (depth-first) traversal of the tree.

← About this adaptation see "Definition: Tree associated with a node", page 448.

In the example, our preorder traversal visited nodes A, **1**, **2**, **3** and **4** in this order.

This correspondence indicates that "Preorder" and "backtracking" are essentially the same idea : the rule that whenever we consider a possible path we exhaust all its possible extensions — all the subtrees of its final node — *before* we look at any of the alternative choices at the same level, represented by siblings of its node. For example if **A** in the previous figure has a third child, the traversal will not consider it before it has exhausted all the subtrees of **2**.

The only property distinguishing a backtracking algorithm from an ordinary preorder traversal is that it stops as soon as it finds a node satisfying the given criterion.

> "Preorder" was defined for binary trees as root first, then left subtree, then right subtree. The left-to-right order — generalized to arbitrary trees by assuming that the children of each node are ordered — is not essential here; "depth-first" does not imply any such ordering. It is just as simple, however, to assume that the choices open to the algorithm at every stage are numbered, and tried in order.

Minimax

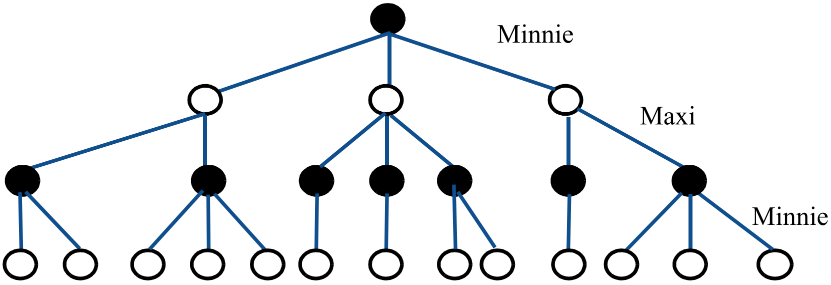
An interesting example of the backtracking strategy, also modeled naturally as a tree, is the "minimax" technique for games such as chess. It is applicable if you can make the following assumptions about the game:

- It is a two-player game. We assume two players called Minnie and Maximilian, the latter familiarly known as Maxi.

- To evaluate the situation at any time during a game, you have an *evaluation function* with a numerical value, devised so that a lower value is better for Minnie and a higher one for Maxi.

> A primitive evaluation function in checkers, assuming Maxi is Black, would be $(mb - mw) + 3 * (kb - kw)$ where mb, mw are the numbers of black and white "men" and kb, kw the corresponding numbers of "kings"; the evaluation function considers a king to be worth three times as much as a man. Good game-playing programs use far more sophisticated functions.

Minnie looks for a sequence of moves leading to a position that minimizes the evaluation function, and Maxi for one that maximizes it.



Game tree

Each player uses the minimax strategy to choose, from a game position, one of the legal moves. The tree model represents possible games; successive levels of the tree alternatively represent the moves of each player.

In the figure, we start from a position where it is Minnie's turn to play. The goal of the strategy is to let Minnie choose, among the moves available from the current position (three in the figure), the one that guarantees the best outcome — meaning, in her case, the minimal guaranteed evaluation function value in leaves of the tree. The method is symmetric, so Maxi would rely on the same mechanism, maximizing instead of minimizing.

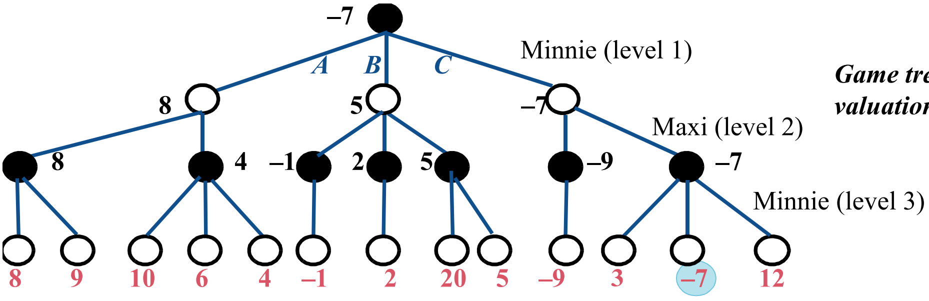
This assumption of symmetry is essential to the minimax strategy, which performs a depth-first traversal of the tree of moves to assign a value to every node:

M1 The value of a leaf is the result of applying the evaluation function to the corresponding game position.

M2 The value of an internal node from which the moves are Maxi's is the *maximum* of the values of the node's children.

M3 In Minnie's case it is the *minimum* of the children's values.

The value of the game as a whole is the value associated with the root node. To obtain a strategy we must retain for each internal node, in cases M2 and M3, not only the value but also the child choice that leads to this value. Here is an illustration of the strategy obtained by assuming some values for the evaluation function (shown in **color**) in the leaves of our example tree:



Game tree with valuations

You can see that the value at each node is the minimum (at levels 1 and 3) or maximum (at level 2) of the values of the children. The desirable move for Minnie, guaranteeing the minimum value –7, is choice ***C***.

Backtracking is appropriate for minimax since the strategy must obtain the values for every node's children before it can determine the value for the node itself, requiring a depth-first traversal.

The following algorithm, a variation on the earlier general backtracking scheme, implements these ideas. It is expressed as a function *minimax* returning a pair of integers: guaranteed value from a starting position *p*, initial choice leading to that value. The second argument *l* is the level at which position *p* appears in the overall game tree; the first move from that position, returned as part of the result, is Minnie's move as in the figures if *l* is odd, and Maxi's if *l* is even.

```
minimax (p: POSITION; l: INTEGER):   TUPLE [value, choice: INTEGER]
        -- Optimal strategy (value + choice) at level l starting from p.
    local
        next: TUPLE [value, choice: INTEGER]
        c: LIST [INTEGER]
    do
        if p.is_terminal (l) then
            Result := [value: p.value; choice: 0]
        else
            c := p.choices
            from
                Result := worst (l )
                c.start
            until c.after loop
                next := minimax (p.moved (c.item), l + 1)
                Result := better (next, Result, l)
            end
        end
    end
```

To represent the result, we use a tuple of integers representing the value and the choice.

The auxiliary functions *worst* and *better* are there to switch between Minnie's and Maxi's viewpoints: the player is minimizing for any odd level *l* and maximizing for any even *l*.

```
worst (l: INTEGER): INTEGER
        -- Worst possible value for the player at level l.
    do
        if l \\ 2 = 1 then Result := Max else Result := Min end
    end


better (a, b: TUPLE [value, choice: INTEGER]; l: INTEGER):
                                        TUPLE [value, choice: INTEGER]
        -- The better of a and b, according to their value, for player at level l.
    do
        if l \\ 2 = 1 then
            Result := (a.value < b.value)
        else
            Result := (a.value > b.value)
        end
    end
```

\\ is integer remainder.

To avoid the repeated use of the TUPLE type, you may instead define a small class GAME_RESULT with integer attributes value and choice.

To determine the *worst* possible value for either player we assume constants *Max*, with a very large value, and *Min*, with a very small value, for example the largest and smallest representable integers.

Function *minimax* assumes the following features from class *POSITION*:

- *is_terminal* indicates that no moves should be explored from a position.
- In that case *value* gives the value of the evaluation function. (The query *value* may have the precondition *is_terminal*.)
- For a non-terminal position *choices* yields the list of choices, each represented by an integer, leading to a legal moves.
- If *i* is such a choice, *moved* (*i*) gives the position resulting from applying the corresponding move to the current position.

The simplest way to ensure that the algorithm terminates is to limit the depth of the exploration to a set number of levels *Limit*. This is why *is_terminal* as given includes the level *l* as argument; it can then be written as just

```
is_terminal (l: INTEGER): BOOLEAN
        -- Should exploration, at level l, stop at current position?
    do
        Result := (l = Limit) or choices.is_empty
    end
```

In practice a more sophisticated cutoff criterion is appropriate; for example the algorithm could keep track of CPU time and stop exploration from a given position when the exploration time reaches a preset maximum.

To run the strategy we call *minimax* (*initial*, 1) where *initial* is the initial game position. Level 1, odd, indicates that the first move is Minnie's.

Alpha-beta

The minimax strategy as seen so far always performs a full backtracking traversal of the tree of relevant moves. An optimization known as alpha-beta pruning can significantly improve its efficiency by skipping the exploration of entire subtrees. It is a clever idea, worth taking a look at not just because it is clever but also as an example of refining a recursive algorithm.

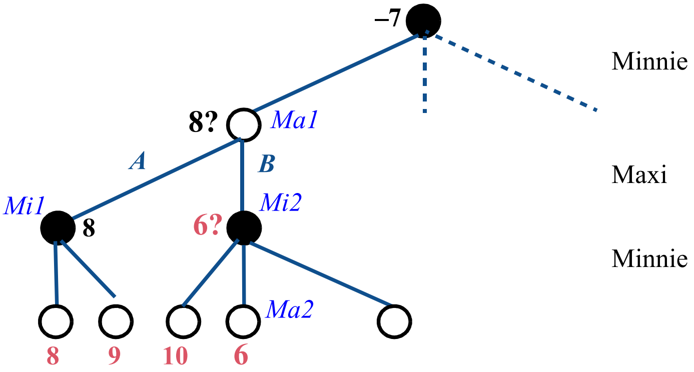
Alpha-beta is only meaningful if, as has been our assumption for minimax, the game strategy for each of the two players assumes that the *other* player's strategy is reversed (one minimizes, the other maximizes) but otherwise identical.

The insight that can trim entire subtrees in the exploration is that it is not necessary for a player at level $l + 1$ to continue exploring a subtree if it finds that this could only deliver a result better for the player itself, and hence worse for its adversary, than what the adversary has already guaranteed at level l: the adversary, which uses the reversed version of the strategy, would never select that subtree.

This discussion refers to a player as "it" since our players are program elements.

The previous example provides an illustration. Consider the situation after the minimax algorithm has explored some of the initial nodes:

Trimming barren subtrees



We are in the process of computing the value (a maximum) for node *Ma1*, and as part of this goal the value (a minimum) for node *Mi2*. From exploring the first subtree of *Ma1*, rooted at *Mi1*, we already have a tentative maximum value for *Ma1*: **8**, signaled by a question mark since it is only temporary. This means a guarantee for Maxi that he will not do, at *Ma1*, worse than **8**. For Maxi, "worse" means lower. In exploring the *Mi2* subtree we come to *Ma2*, where the value — obtained in this case from the evaluation function since *Ma2* is a leaf, but the reasoning would apply to any node — is **6**. So at node *Mi2* Minnie will not do worse (meaning, in her case, higher) than 6. But then Maxi would never, from node *Ma2*, take choice ***B*** leading to *Mi2*, since he already has a better result from choice ***A***. Continuing to explore the subtree rooted at *Mi2*, part of choice ***B***, would just be a waste of time. So as soon as it has found value **6** at *Ma2* the alpha-beta strategy discards the rest of the *Mi2* subtree.

> In the figure's example there is only one node left in the *Mi2* subtree after *Ma2* and we are at the leaf level, but of course *Ma2* could have many more right siblings with large subtrees.

Not only is this optimization an interesting insight; it also provides a good opportunity to hone our recursive programming skills. Indeed do not wait for the solution (that is to say, refrain from turning the page just now!) and try first to devise it by yourself:

Programming time!
Adding Alpha-beta to Minimax

Adapt the minimax algorithm, as given earlier, so that it will use the alpha-beta strategy to avoid exploring useless subtrees.

← Function minimax, page 466.

The extension is simple. (Well, as you will have noted if you did try, it requires some care to get the details right, in particular to avoid getting our *better* comparisons upside down.) The routine needs one more argument to denote the value, if any, already guaranteed for the adversary at the level immediately above. Here is minimax updated for Alpha-beta, additions highlighted:

← The parts not highlighted are unchanged from minimax, page 466 (departing from the convention of the rest of this chapter, which highlights recursive branches).

```
alpha_beta (p: POSITION; l: INTEGER; guarantee: INTEGER):
                                        TUPLE [value, choice: INTEGER]
        -- Optimal strategy (value + choice) at level l, starting from p.
        -- Even level minimizes, odd level maximizes.
    local
        next: TUPLE [value, choice: INTEGER]
    do
        if p.is_terminal (l) then
            Result := [value: p.value; choice: 0]
        else
            c := p.choices
            from
                Result := worst (l)
                c.start
            until c.after or better (guarantee, Result, l – 1) loop
                next := minimax (p.moved (c.item), l + 1), Result)
                Result := better (next, Result, l)
            end
        end
    end
```

Each player now stops exploring **its** alternatives whenever it finds a result that is "*better*" **for the adversary** than the "*guarantee*" the adversary may already have assured.

Since *better* was defined without a precondition it will accept a zero level, so it is acceptable to pass it $l - 1$. We might equivalently pass $l + 1$. In fact a slightly simpler variant of *better* (*guarantee*, **Result**, l – 1) is *better* (**Result**, *guarantee*, l); it is equivalent thanks to the symmetric nature of the strategy.

The recursive call passes as a "*guarantee*" to the next level the best **Result** obtained so far for the current level. As a consequence, alpha-beta's trimming, which stops the traversal of a node's children when it hits the new exit trigger *better* (*guarantee*, **Result**, l – 1), will never occur when the node itself is the *first child* of its own parent; this is because the loop initializes **Result** to the *worst* value for the player, so the initial *guarantee* is useless. Only when the traversal moves on to subsequent children does it get a chance to trigger the optimization.

Minimax and alpha-beta provide a representative picture of backtracking algorithms, which have widespread applications to problems defined by large search spaces. The key to successful backtracking strategies is often — as illustrated by alpha-beta — to find insights that avoid exhaustive search.

14.6 FROM LOOPS TO RECURSION

Back to the general machinery of recursion.

We have seen that some recursive algorithms — Fibonacci numbers, search and insertion for binary search trees — have a loop equivalent. What about the other way around?

It is indeed not hard to replace *any* loop by a recursive routine. Consider an arbitrary loop, given here without its invariant and variant (although we will see their recursive counterparts later):

```
from Init until Exit loop Body end
```

We may replace it by

```
Init
loop_equiv
```

with the procedure

```
loop_equiv
        --Emulate a loop of exit condition Exit and body Body.
    do
        if not Exit then
            Body
            Loop_equiv
        end
    end
```

In **functional languages** (such as Lisp, Scheme, Haskell, ML), the recursive form is the preferred style, even if loops are available. We could use it too in our framework, replacing for example the first complete example of the discussion of loops, which animated a Metro line by moving a red dot, with

← *“Functional programming and functional languages”, page 324.*

← *Page 168.*

```
Line8.start
animate_rest (Line8)
```

relying on the auxiliary routine

```
animate_rest (line: LINE)
        -- Animate stations of line from current cursor position on
    do
        if not line.after then
            show_spot (line.item.location)
            line.forth
            animate_rest (line)
        end
    end
```

(A more complete version should restore the cursor to its original position.)

The recursive version is elegant, but there is no particular reason in our framework to prefer it to the loop form; indeed we will continue to use loops.

> The conclusion might be different if we were using functional programming languages, where systematic reliance on recursive routines is part of a distinctive style of programming.

Even if just for theoretical purposes, it is interesting to know that loops are conceptually not needed if we have routines that can be recursive. As an example, recursion gives us a more concise version of the loop-based routine *paradox* demonstrating the unsolvability of the Halting Problem:

← *"An application: proving the undecidability of the halting problem", page 223.*

```
recursive_paradox
        -- Terminate if and only if not.
    do
        if terminates ("C:\your_project") then
                recursive_paradox
        end
    end
```

Knowing that we can easily emulate loops with recursion, it is natural to ask about the reverse transformation. Do we really need recursive routines, or could we use loops instead?

We have seen straightforward cases: *Fibonacci* as well as *has* and *put* for binary search trees. Others such as *hanoi*, *height*, *print_all* do not have an immediately obvious recursion-free equivalent. To understand what exactly can be done we must first look more closely into the meaning and properties of recursive routines.

14.7 MAKING SENSE OF RECURSION

The experience of our first few recursive schemes allows us to probe a bit deeper into the meaning of recursive definitions.

Vicious circle?

First we go back to the impolite but inevitable question: does the recursive emperor have any clothes? That is to say, does a recursive definition mean anything at all? The examples, especially those of recursive routines, should by now be sufficiently convincing to suggest a positive answer, but we should still retain a healthy dose of doubt. After all we keep venturing dangerously close to definitions that make no sense at all — vicious circles. With recursion we try to define a concept in terms of itself, but we cannot just define it *as* itself. If I say

> “Computer science is the study of computer science”

I have not defined anything at all, just stated a tautology; not one of those tautologies of logic, which are things to prove and hence possibly interesting, just a platitude. If I refine this into

← “Definition: Tautology”, page 78.

> “Computer science is the study of programming, data structures, algorithms, applications, theories and other areas of computer science”

I have added some usable elements but still not produced a satisfactory definition. Recursive routines can, similarly, be obviously useless, as:

```
p (x: INTEGER)
        -- What good is this?
    do p (x) end
```

which for any value of the argument would execute forever, never producing any result.

> “Forever” in this case means, for a typical compiler’s implementation of recursion on an actual computer, “until the stack overflows and causes the program to crash”. So in practice, given the speed of computers, “forever” does not last long. — you can try the example for yourself.

→ You can see an example of the result on page 665.

How do we avoid such obvious misuses of recursion? If we attempt to understand why the recursive definitions seen so far seem intuitively to make sense, we can nail down three interesting properties:

Touch of Methodology: Well-formed recursive definition

A useful recursive definition should ensure that:

R1 There is at least one non-recursive branch.

R2 Every recursive branch occurs in a context that differs from the original.

R3 For every recursive branch, the change of context (R2) brings it closer to at least one of the non-recursive cases (R1).

For a recursive routine, the change of "context" (R2) may be that the call uses a different argument, as will a call *r* (*n*–1) in a routine *r* (*n*: *INTEGER*); that it applies to a different target, as in a call *x*.*r* (*n*) where *x* is not the current object; or that it occurs after the routine has changed at least one field of at least one object.

The recursive routines seen so far satisfy these requirements:

- The body of *Hanoi* (*n*, …) is of the form **if** *n* > 0 **then** … **end** where the recursive calls are in the **then** part, but there is no **else** part, so the routine does nothing for *n* = 0 (R1). The recursive calls are of the form *Hanoi* (*n*–1, …), changing the first argument and also switching the order of the others (R2). Replacing *n* by *n*–1 brings the context closer to the non-recursive case *n* = 0 (R3). ← *Page 443.*

- The recursive *has* for binary search trees has non-recursive cases for *x* = *item*, as well as for *x* < *item* if there is no left subtree, and *x* > *item* if there is no right subtree (R1). It calls itself recursively on a different target, *left* or *right* rather than the current object (R2); every such call goes to the left or right subtree, closer to the leaves, where the recursion terminates (R3). The same scheme governs other recursive routines on binary trees, such as *height*. ← *Page 457.* ← *Page 452.*

- The recursive version of the metro line traversal, *animate_rest*, has a non-recursive branch (R1), doing nothing, for a cursor that is *after*. The recursive call does not change the argument, but it is preceded by a call *line*.*forth* which changes the state of the *line* list (R2), moving the cursor closer to a state satisfying *after* and hence to the non-recursive case (R3). ← *Page 472.*

R1, R2 and R3 also hold for recursive definitions of concepts other than routines:

- The mini-grammar for Instruction has the non-recursive case Assignment. ← *Page 437.*

- All our recursively defined data structures, such as *STOP*, are recursive through *references* (never through expanded values), and references can be void; in linked structures, void values serve as terminators. ← *Page 437.*

In the case of recursive routines, combining the above three rules suggests a notion of **variant** similar to the loop variants through which we guarantee that loops terminate:

← *"Loop termination and the halting problem", page 161.*

Touch of Methodology: Recursion Variant

Every recursive routine should be declared with an associated recursion variant, an integer quantity associated with any call, such that:

- The routine's precondition implies that the variant is non-negative.
- If an execution of the routine starts with a value *v* for the variant, the value *v'* of the variant for any recursive call satisfies $0 \le v' < v$.

The variant may involve the arguments of the routine, as well as other parts of its environment such as attributes of the current object or of other objects. In the examples just reviewed:

- For *Hanoi* (*n*, …), the variant is *n*.

- For *has*, *height*, *print_all* and other recursive traversals of binary trees, the variant is *node_height*, the longest length of a path from the current node to a leaf.

- For *animate_rest*, the variant is, as for the corresponding loop, *Line8*.*count* – *Line8*.*index* + 1. ← *Page 168.*

There is no special syntax for recursion variants, but we will use a comment of the following form, here for *hanoi*:

```
        -- variant n
```

Boutique cases of recursion

The well-formedness rules seem so reasonable that we might think they are necessary, not just sufficient, to make a recursive definition meaningful. Such is indeed the case with the first two properties:

- R1: if all branches of a definition are recursive, it cannot ever yield any instance we do not already know. In the case of a recursive routine, execution will not terminate, except in practice through a crash following memory exhaustion.

- R2: if a recursive branch applies to the original context, it cannot ever yield an instance we do not already know. For a recursive routine — say *p* (*x*: *T*) with a branch that calls *p* (*x*) for the same *x* with nothing else changed — this means that the branch, if taken, would lead to non-termination. For other recursive definitions, it means the branch is useless.

The story is different for R3, if we take this rule as requiring a clearly visible recursion variant such as the argument *n* for *Hanoi*. Some recursive routines which do terminate violate this property. Here are two examples. They have no practical application, but highlight general properties of which you must be aware.

McCarthy's 91 function was devised by John McCarthy, a professor at Stanford University, designer of the Lisp programming language (where recursion plays a prominent role) and one of the creators of Artificial Intelligence. We may write it as follows:

← See "Functional programming and functional languages", page 324 (with photograph of McCarthy).

```
mc_carthy (n: INTEGER): INTEGER
        -- McCarthy's 91 function.
    do
        if n > 100 then
            Result := n – 10
        else
            Result := mc_carthy (mc_carthy (n + 11))
        end
    end
```

The value for $n > 100$ is clearly $n - 10$, but it is far less obvious — from a computation shrouded in two nested recursive calls — that for any integer up to 99, including negative values, the result will be 91, explaining the function's name. The computation indeed terminates on every possible integer value. Yet it has no obvious variant; *mc_carthy* (*mc_carthy* (*n* + 11)) actually uses as argument of the innermost recursive call a *higher* value than the original!

Here is another example, also a mathematical oddity:

```
bizarre (n: INTEGER): INTEGER
        -- A function that can yield only a 1.
    require
        positive: n >= 1
    do
        if n = 1 then
            Result := 1
        elseif even (n) then
            Result := bizarre (n // 2)
        else            -- i.e. for n odd and n > 1
            Result := bizarre ((3 ∗ n + 1) // 2)
        end
    end
```

This uses the operator // for rounded down integer division (5 // 2 and 4 // 2 are both 2), and a boolean expression *even* (*n*) to denote whether *n* is an even integer; *even* (*n*) can also be expressed as *n* \\ 2 = 0, using the integer remainder operator \\. The two occurrences of a // division in the algorithm apply to even numbers, so they are exact.

n / 2, using the other division operator /, would give a REAL result; for example 5 /2 is 2.5.

Clearly, if this function gives any result at all, that result can only be 1, the value produced by the sole non-recursive branch. Less clear is whether it will give this result — that is to say, terminate — for *any* possible argument. The answer seems to be yes; if you write the program, and try it on sample values, including large ones, you will be surprised to see how fast it converges. Yet there is no obvious recursion variant; here too the change seems to go in the wrong direction: the new argument in the second recursive branch, (3 ∗ *n* + 1) // 2, is actually larger than *n*, the previous value.

These are boutique examples, but we must take their existence into account in any general understanding of recursion. They mean that some recursive definitions exist that do *not* satisfy the seemingly reasonable methodological rules discussed above — and still yield well-defined results.

Note that such examples, if they terminate for every possible argument, do have a variant: since for any execution of the routine the number of remaining recursive calls is entirely determined by the program's state at the time of the call; it is a function of the state, and can serve as a variant. Rather, it *could* serve as a variant if we knew how to express it. If we don't, its theoretical existence does not help us much.

> You will have noted that it is not possible to determine automatically — through compilers or any other program analysis tools — whether a routine has a recursive variant, even less to derive such a variant automatically: that would mean that we can solve the Halting Problem.

← *"An application: proving the undecidability of the halting problem", page 223.*

In practice we dismiss such examples and limit ourselves to recursive definitions that possess properties R1, R2 and R3, guaranteeing that they are safe. In particular, whenever you write a recursive routine, you must always — as in the examples of the rest of this chapter — explicitly list a recursive variant.

Keeping definitions non-creative

Even with well-formedness rules and recursion variants, we are not yet off the hook in our attempts to use recursion and still sleep at night. The problem is that a recursive "definition" is not a definition in the usual sense because it can be **creative**.

An *axiom* in mathematics is creative: it tells us something that we cannot deduce without it, for example (in the standard axioms for integers) that $n < n'$ holds for any integer n, where n' is the next integer. The basic *laws* of natural sciences are also creative, for example the rule that nothing can travel faster than the speed of light.

Theorems in mathematics, and specific results in physics, are not creative: they state properties that can be deduced from the axioms or laws. They are interesting on their own, and may start us on the path to new theorems; but they do not add any assumptions, only consequences of previous assumptions.

A definition too should be non-creative. It gives a new name for an object of our world, but all statements we can express with the definition could be expressed without it. We do not *want* to express them without it — otherwise we would not introduce the definition — but we trust that in principle we could. If I say

Define x^2, for any x, as $x * x$

I have not added anything to mathematics; I am just allowing myself to use the new notation e^2, for any expression e, in lieu of the multiplication. Any property that can be proved using the new form could also be proved — if more clumsily — using the form that serves to define it.

The symbol $\triangleq$, which we have taken to mean "is defined as" (starting with BNF productions), assumes this principle of non-creativity of definitions. But now consider a recursive definition, of the form

← *From page 298 on.*

$$f \triangleq \textit{some\_expression} \qquad \textbf{[1]}$$

where *some_expression* involves *f*. It does not satisfy the principle any more! If it did we could replace any occurrence of *f* by *some_expression*; this involves *f* itself, so we would have to do it again, and so on ad infinitum. We have not really defined anything.

Until we have solved this issue — by finding a convincing, non-creative meaning for "definitions" such as [1] — we must be careful in our terminology. We will reserve the $\triangleq$ symbol for non-recursive definitions; a property such as [1] will be expressed as an equality

$$f = some\_expression \qquad [2]$$

which simply states a property of the left and right sides. (We may also view it as an **equation**, of which *f* must be a solution.) To be safe when talking about recursive "definitions", we will quarantine the second word in quotes.

The quarantine ends on page 482.

The bottom-up view of recursive definitions

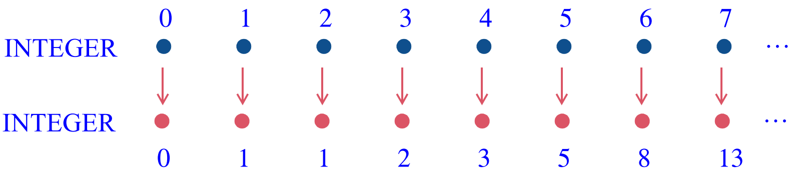
To sanitize recursion and bring it out of the quarantined area, it is useful to take a **bottom-up** view of recursive routines and, more generally, recursive "definitions". I hope this will remove any feeling of dizziness that you may still experience when seeing concepts or routines defined — apparently — in terms of themselves.

In a recursive "definition", the recursive branches are written in a *top-down* way, defining the meaning of a concept in terms of the meaning of the same concept for a "smaller" context — smaller in the sense of the variant. For example, *Fibonacci* for n is expressed in terms of *Fibonacci* for $n - 1$ and $n - 2$; the moves of *Hanoi* for n are expressed in terms of those for $n - 1$; and the syntax for *Instruction* involves a Conditional that contains a smaller *Instruction*.

The bottom-up view is a different interpretation of the same definition, treating it the other way around: as a mechanism that, from *known* values, gives new ones. Here is how it works, first on the example of a function. For any function *f* we may build the **graph of the function**: the set of pairs $[x, f(x)]$ for every applicable x. The graph of the Fibonacci function is the set

$$F \triangleq \{[0, 0], [1, 1], [2, 1], [3, 2], [4, 3], [5, 5], [6, 8], [7, 13] \ldots\} \qquad [3]$$

consisting of all pairs $[n, Fibonacci\ (n)]$ for all non-negative integers n. This graph contains all information about the function. You may prefer to think of it in the following visual representation:



A function graph (for the Fibonacci function)

The top row lists possible arguments to the function; for each of them, the bottom row gives the corresponding *fibonacci* number.

To give the function a recursive "definition" is to say that its graph F — as a set of pairs — satisfies a certain property

$$F = h\,(F) \qquad \textbf{[4]}$$

for a certain function h applicable to such sets of pairs. This is like an equation that F must satisfy, and is known as a **fixpoint equation**. A fixpoint equation expresses that a certain mathematical object, here a function, remains invariant under a certain transformation, here h.

For example to "define" the Fibonacci function recursively as

$fib\,(0) = 0$
$fib\,(1) = 1$
$fib\,(i) = fib\,(i-1) + fib\,(i-2)$-- For $i > 1$

is to state that its graph F — the above set of pairs [3] — satisfies the fixpoint equation $F = h\,(F)$ [4] where h is the function that, given such a set of pairs, yields a new one containing the following pairs:

G1 Every pair already in F.

G2 $[0, 0]$. -- The pair for $n = 0$: $[0, fib\,(0)]$

G3 $[1, 1]$. -- The pair for $n = 1$: $[1, fib\,(1)]$

G4 Every pair of the form $[i, a + b]$ for some i such that F contains both a pair of the form $[i-1, a]$ and another of the form $[i-2, b]$.

We can use this view to give any recursive "definition" a clear meaning, free of any recursive mystery. We start from the function graph F_0 that is empty (it contains no pair). Next we define

$$F_1 \triangleq h\,(F_0)$$

meaning, since G1 and G4 are not applicable in this case (as F_0 has no pair), that F_1 is simply $\{[0, 0], [1, 1]\}$, with the two pairs given by G2 and G3. Next we apply h once more to get

$$F_2 \triangleq h(F_1)$$

Here and in subsequent steps G2 and G3 give us nothing new, since the pairs $[0, 0]$ and $[1, 1]$ are already in F_1, but G4, applied to these two pairs from F_1, adds to F_2 the pair $[2, 1]$. Continuing like this, we define a sequence of graphs: F_0 is empty, and each F_{i+1} for $i > 0$ is defined as $h(F_i)$. Now consider the infinite union F of all the F_i for every natural integer i: $F_0 \cup F_1 \cup F_2 \cup \ldots$, more concisely written

$$\bigcup_{i \in \mathbf{N}} F_i$$

where $\mathbf{N}$ is the set of natural integers. It is easy to see that this F satisfies the property $F = h(F)$ [4].

This is the non-recursive interpretation — the semantics — we give to the recursive "definition" of Fibonacci.

In the general case, a fixpoint equation of the form [4] on function graphs, stating that F must be equal to $h(F)$, admits as a solution the function graph

$$F \triangleq \bigcup_{i \in \mathbf{N}} F_i$$

where F_i is a sequence of function graphs defined as above:

$$F_0 \triangleq \{\} \qquad \text{-- Empty set of pairs}$$

$$F_i \triangleq h(F_{i-1}) \qquad \text{-- For } i > 0$$

The empty set can, of course, be written also as ∅. The notation { } emphasizes that it is a set of pairs.

This fixpoint approach is the basis of the bottom-up interpretation of recursive computations. It removes the apparent mystery from these definitions because it no longer involves defining anything "in terms of itself": it simply views a recursive "definition" as a fixpoint equation, and admits a solution obtained as the union (similar to the limit of a sequence in mathematical analysis) of a sequence of function graphs.

> This immediately justifies the requirement that any useful recursive "definition" must have a non-recursive branch: if not, the sequence, which starts with the empty set of pairs $F_0 = \{\}$, never gets any more pairs, because all the cases in the definition of h are like G1 and G4 for Fibonacci, giving new pairs deduced from existing ones, but there are no pairs to begin with.

← R1, page 474.

This technique reduces recursive "definitions", with all the doubts they raise as to whether they define anything at all, to the well-known, traditional notion of defining a sequence by induction.

The Fibonacci function is a good example for understanding the concepts, but perhaps not sufficient to get really excited: after all, its usual definition in mathematics textbooks already involves induction; only computer scientists look at the function in a recursive way. What we saw is that we can treat its recursive "definition" as an inductive definition — a good old definition, without the quotes — of the function's graph. We did not learn anything about the function itself, other than a new viewpoint. Let us see whether the bottom-up view can teach us something about a few of our other examples.

This is the end of the "quarantine" decreed on page 479.

Bottom-up interpretation of a construct definition

Understood in a bottom-up spirit, the recursive definition of "type" has a clear meaning. As you will remember, it said that a type is either:

← "Definitions: Class type, generically derived, base class", page 370.

T1 A non-generic class, such as *INTEGER* or *STATION*.

T2 A generic derivation, of the form $C\,[T]$, where C is a generic class and T a type.

T1 is the non-recursive case. The bottom-up perspective enables us to understand the definition as building the set of types as a succession of layers. Limiting for simplicity the number of possible generic parameters to one:

- Layer L_0 has all the types defined by non-generic classes: *INTEGER*, *STATION* and so on.
- Layer L_1 has all the types of the form $C\,[X]$, where C is a generic class and X is at level L_0: *LIST [INTEGER]*, *ARRAY [STATION]* etc.
- More generally, layer L_n for any $n > 0$, has all the types of the form $C\,[X]$, where X is at level L_i for $i < n$.

This way we get all possible types, generically derived or not.

The towers, bottom-up

Now consider the Tower of Hanoi solution from a bottom-up perspective. We may understand the routine as recursively defining a sequence of moves. Let's denote such a sequence — move a disk from the top of needle A to B, then one from C to A and so on — as $<\mathrm{A} \to \mathrm{B}, \mathrm{C} \to \mathrm{A}, \ldots>$. The empty sequence of moves will be $<>$ and the concatenation of sequences will use a simple "+", so that $<\mathrm{A} \to \mathrm{B}, \mathrm{C} \to \mathrm{A}> + <\mathrm{B} \to \mathrm{A}>$ is $<\mathrm{A} \to \mathrm{B}, \mathrm{C} \to \mathrm{A}, \mathrm{B} \to \mathrm{A}>$.

Then we may express the recursive solution to the Towers of Hanoi problem as a function *han* with four arguments (an integer and three needles), yielding sequences of moves, and satisfying the fixpoint equation

$$\begin{array}{lll} han\,(n, s, t, o) = & & \\ \quad <> & \text{-- If } n = 0 & \textbf{[5]} \\ \quad han\,(n-1, s, o, t) + <s \to t> + han\,(n-1, o, t, s) & \text{-- If } n > 0 & \textbf{[6]} \end{array}$$

defined only when the values of s, t, o (short for *source*, *target*, *other*) are different — we take them as before to range over 'A', 'B', 'C' — and n is positive.

The bottom-up construction of the function that solves this equation is simple. [5] lets us initialize the function's graph to all pairs for $n = 0$, each of the form

$$[(0, s, t, o),\ <>]$$

for s, t, o ranging over all permutations of 'A', 'B', 'C'. Let us call H_0 this first part of the graph, made of six pairs.

Now we may use [6] to obtain the next part H_1, containing all the values for $n = 1$; they are all of the form

$$[(1, s, t, o),\ <s \to t>]$$

since for any sequence x the concatenation $<> + x$ or $x + <>$ is x itself. The next iteration of [6] gives us H_2, whose pairs are of the form

$$[(2, s, t, o),\quad f1 + <s \to t> + g1]$$

for all s, t, o such that H_1 contains both a pair of the form $[(1, s, o, t), f1]$ and one of the form $[(1, o, t, s), g1]$.

Iterating again will give us H_3 and subsequent elements of the graph. The complete graph — infinite of course, since it includes pairs for all possible values of n — is the set of all pairs in all elements of the sequence, $\bigcup_{i \in \mathbf{N}} H_i$.

Here I strongly suggest that you get a concrete grasp of the bottom-up view of recursive computation by writing a program that actually builds the graph:

***Programming time*:**
Producing the graph of a function

Write a program (not using recursion) that produces successive elements H_0, H_1, H_2 … of the function graph for the recursive Hanoi solution.

→ *Details in exercise 14-E.11, page 503.*

A related exercise asks you to determine (without programming) the mathematical properties of the graph.

→ *14-E.10, page 503.*

Another important exercise directs you to apply a similar analysis to binary tree traversals. You will have to devise a model for representing the solution, similar to the one we have used here; instead of sequences of moves you will simply use sequences of nodes.

→ *14-E.12, page 503.*

Grammars as recursively defined functions

The bottom-up view is particularly intuitive for a recursive grammar, as in our small example:

```
Instruction ≜ ast | Conditional
Conditional ≜ ifc Instruction end
```

← *Actual version on page 437.*

distilled even further here: **ifc** represents "**if** Condition **then**" and **ast** represents Assignment, both treated as terminals for this discussion.

It is easy to see how to generate successive sentences of the language by interpreting these productions in a bottom-up, fixpoint-equation style:

```
ast
ifc ast end
ifc ifc ast end end
ifc ifc ifc ast end end end
```

and so on. You can also look again, in light of the notion of bottom-up recursive computation, at the earlier discussion of the little Game language.

← *"Recursive grammars", page 307.*

It is possible to generalize this approach to arbitrary grammars by taking a matrix view of a BNF description.

→ *Exercise 14-E.17, page 504.*

14.8 CONTRACTS FOR RECURSIVE ROUTINES

We have learned to equip our classes and their features with **contracts** stating their correctness properties: routine preconditions, routine postconditions, class invariants; the same concerns applied to algorithms gave us loop variants and loop invariants. How does recursion affect the picture?

We have already seen the notion of **recursion variant**. If a routine is recursive directly or indirectly, you should include a mention of its variant. As noted, we do not have specific language syntax for this but add a clause

← *"Touch of Methodology: Recursion Variant", page 475.*

```
-- variant: integer_expression
```

to the routine's header comment.

A recursive routine may have a precondition and postcondition like any other routine. Because ensuring a precondition is always the responsibility of the caller, and here the routine is its own caller, the novelty is that you must ensure that all calls within the routine (or, for indirect recursion, in associated routines) satisfy the precondition.

Here is the Towers of Hanoi routine with more complete contracts; the new clauses, expressed as comments, are highlighted.

← *The original was on page 443.*

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
        -- Transfer n disks from source to target, using other as intermediate
        -- storage, according to rules of Tower of Hanoi puzzle.
         -- invariant: disks on each needle are piled in decreasing size.
         -- variant: n
    require
        non_negative: n >= 0
        different1: source /= target
        different2: target /= other
        different3: source /= other
       -- source has n disks; any disks on target and other are all
       -- larger than all the disks on source.
    do
        if n > 0 then
            hanoi (n–1, source, other, target)
            move (source, target)
            hanoi (n–1, other, target, source)
        end
    ensure
        -- Disks previously on source are now on target, in same order,
        -- on top of those previously there if any; other is as before.
    end
```

A properly specified recursive routine has a **recursion invariant**: a set of properties that must hold both before and after each execution. In the absence of a specific language mechanism they will just appear twice, in the precondition as well as in the postcondition; for clarity you may also, as here, include them in the header comment under the form

```
-- invariant: boolean_expression
```

This is not a language construct but relies on the following convention:

- If the recursion invariant is just pseudocode expressed as a comment, as in this example, do not repeat it in the precondition and postcondition; here this means omitting from the precondition and postcondition the property that any disks on the affected needles are piled up in decreasing size.

- Any recursion invariant clause that is formal (a boolean expression) should be included in the precondition and postcondition, since there is no other way to express it formally.

14.9 IMPLEMENTATION OF RECURSIVE ROUTINES

Recursive programming works well in certain problem areas, as illustrated by the examples in this chapter. When recursion facilitates your job you should not hesitate to use it, since in modern programming languages you can take recursion for granted.

Since there is usually no direct support for recursion in machine code, compilers for high-level languages must map a recursively expressed algorithm into a non-recursive one. The applicable techniques are obviously important for compiler writers, but even if you do not expect to become one it is useful to know the basic ideas, both to gain further insight into recursion (complementing the perspectives opened by previous sections) and to understand the potential performance cost of using recursive algorithms.

We will look at some recursive schemes and ask ourselves how, if the language did *not* permit recursion, we could devise non-recursive versions, also called **iterative**, achieving the same results.

A recursive scheme

Consider a routine *r* that calls itself:

```
r (x: T)
    do
        code_before
        r (y)
        code_after
    end
```

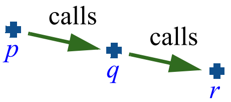
There might be several recursive calls, but we look at just one. What does it mean — if we revert to a top-down view — to execute that call?

The presence of recursion implies that neither the beginning of the routine's code nor its end are just what they pretend to be:

- When *code_before* executes, this is not necessarily the beginning of a call *a*.*r* (*y*) or *r* (*y*) executed by some client routine: it could result from an instance of *r* calling itself recursively.
- When *code_after* terminates, this is not necessarily the end of the *r* story: it may simply be the termination of one recursively called instance; execution should resume for the last instance started and not terminated.

Routines and their execution instances

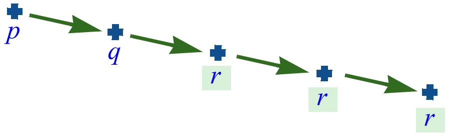
The key novelty in the last observation is the concept of **instance** (also called **activation**) of a routine. We know that classes have instances — the "objects" of object-oriented program execution — but we have not yet thought of routines in a similar way.



A call chain, without recursion

At any moment during a program's execution, the state of the computation is characterized by a **call chain** as pictured above: the root procedure *p* has called *q* which has called *r*… When an execution of a routine in the chain, say *r*, terminates, the suspended execution of the calling routine, here *q*, resumes just after the place where it had called *r*.

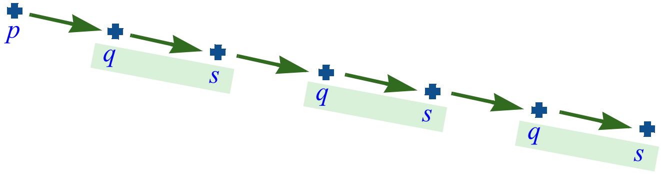
In the absence of recursion, we did not need to make the concept of routine instance explicit since any routine had, at any time, at most one active instance. With recursion, the call chain may include two or more instances of the same routine. Under *direct* recursion they will be contiguous:



Call chain with direct recursion

For example a call *hanoi* (2, *s*, *t*, *o*) immediately starts a call *hanoi* (1, *s*, *o*, *t*) which starts a call *hanoi* (0, *s*, *t*, *o*); at that stage we have three instances of the procedure in the call chain.

A similar situation arises with *indirect* recursion:



Call chain with indirect recursion

Preserving and restoring the context

All instances of a routine share their program code; what distinguishes them is their execution *context*. We have seen that in a useful case of recursion the context of every call must differ by at least one element. The context elements characterizing a routine instance (rather than object states) are:

← *R2, page 474.*

- The values of the actual routine arguments, if any, for the particular call.
- The values of the local variables, if any.
- The location of the call in the text of the calling routine, defining where execution should continue once the call completes.

As we saw when studying how stacks support the execution of programs in modern languages, a data structure representing such a routine execution context is called an **activation record**.

← *"Using stacks", page 421.*

Assume a programming language that does *not* support recursion. Since at any time during execution there is at most one instance of any routine, the compiler-generated program can use a single activation record per routine. This is known as **static allocation**, meaning that the memory for all activation records can be allocated once and for all at the beginning of execution.

With recursion each activation of the routine needs its own context. This leaves two possibilities for implementation:

I1 We can resort to *dynamic allocation*: whenever a routine instance starts, create a fresh activation record to hold the routine's context. Use this activation record whenever the routine execution needs to access an argument or local variable; use it too on instance termination, to determine where execution must continue in the caller's code. Resuming the caller's execution implies going back to its own activation record.

I2 To save space, we may note that the reason for keeping context information in an activation record is to be able to *restore* it when an execution resumes after a recursive call. An alternative to saving that information is to *recompute* it. This is possible when the change performed by the recursive call is **invertible**. The recursive calls in procedure *hanoi* (n, …) are of the form *hanoi* ($n - 1$, …); rather than storing the value of n into an activation record, creating a new record holding the value $n - 1$, then restoring the previous record on return, we may use a single location for n in all recursive instances, as with static allocation: at call time, we decrease the value by one; at return time, we *increase* the value by one.

The two techniques are not exclusive: you can save space by using I2 for values whose transformation in calls (such as replacing n by $n - 1$) admits an easily implemented inverse, and retain an activation record for the rest of the context. The decision may involve a space-time tradeoff if the inverse transformation, unlike the $n := n + 1$ example, is computationally expensive.

Using an explicit call stack

Of the two strategies for handling routine contexts let us look first at I1, which relies on explicit activation records.

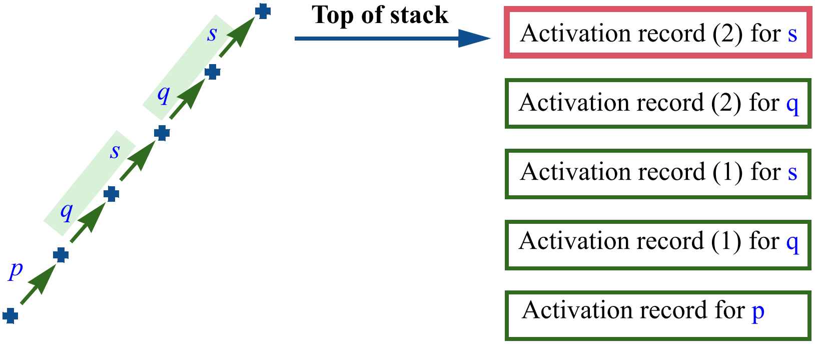
Like activation records, *objects* are created dynamically, as a result of **create** instructions. The program memory area devoted to dynamically allocated objects is known as the **heap**. But for activation records of routines we do not need to use the heap since the patterns of activation and deactivation are simple and predictable:

← *"Creating simple objects", 6.4, page 118.*

- A call to a routine requires a new activation record.
- On returning from that call, we may forget this activation record (it will never be useful again, since any new call will need its own values), and we must restore the caller's activation record.

This is a *last-in, first-out* pattern for which we have a ready-made data structure: stacks. The stack of activation records will reflect the call chain, pictured here going up:

← "Stacks", 13.11, page 420.


s
q
s
q
p
Top of stack
Activation record (2) for s
Activation record (2) for q
Activation record (1) for s
Activation record (1) for q
Activation record for p

Call chain and the corresponding call stack

We have encountered the stack of activation records before: it is the **call stack** which keeps track of routine calls during execution. If you are programming in a language supporting recursion, the call stack is the responsibility of the code generated by the compiler. Here we are looking at how to manage it ourselves.

← "Using stacks", page 421.

You can use an explicit stack of activation records to produce an iterative equivalent of a recursive routine:

"Iterative", defined on page 486, means non-recursive.

- To access local variables and arguments of the current routine: always use the corresponding positions in the activation record at the top of the stack.
- Instead of a recursive call: create a new activation record; initialize it with the value of the call's arguments and the position of the call; push it on the stack; and branch back (**goto**) to the beginning of the routine's code.
- Instead of a return: return only if the stack is empty (no suspended call is pending); otherwise, restore the arguments and local variables from the activation record at the top of the stack, pop the stack, and branch to the appropriate instruction based on the call position information found in the activation record.

Note that both translation schemes involve **goto** instructions. That is fine if we are talking about the machine code to be generated by a compiler; but when it is a manual simulation of recursion in a high-level language we have learned to avoid the **goto** and in fact Eiffel has no such instruction. We will have to write **goto**s temporarily, then replace them by appropriate control structures.

← "The goto instruction", page 183.

Recursion elimination essentials

Let us see how the scheme works for the body of *hanoi* with its two recursive calls. We use a stack of activation records, called just *stack*:

```
stack: STACK [RECORD]
```

with a small auxiliary class *RECORD* to describe activation records:

```
note
    description: "Data associated with a routine instance"
class RECORD create
    make
feature -- Initialization
    make (n: INTEGER; c: INTEGER; s, t, o: CHARACTER)
            -- Initialize from count n, call c, source s, target t, intermediary o.
        do
            count := n ; call := c; source := s ; target := t ; other := o
        end
feature -- Access
    count: INTEGER.
        -- Number of disks.

    call: INTEGER
        -- Identifies a recursive call: 1 for the first, 2 for the second.

    source, target, other: CHARACTER -- First call
        -- Needles.
end
```

(Instead of a full-fledged class we could also just use tuples.) An instance of the class represents the context of a call: the number of disks being moved (*count*), the three needles in the order used by the call, and *call* telling us whether this execution, if coming from a recursive call, came from the first or second call in

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
    do
        if n > 0 then
            hanoi (n–1, source, other, target)      -- First call
            move (source, target)
            hanoi (n–1, other, target, source)      -- Second call
        end
    end
```

We use the *stack* of activation records to provide a non-recursive version of the procedure, temporarily relying on **goto**s, as shown on the following page.

Warning: *because of the* **goto** *instructions and labels this is not legal Eiffel. The gotos will be removed next.*

```
iterative_hanoi (n: INTEGER; source, target, other: CHARACTER)
    local  -- We need locals representing arguments to successive calls:
        count: INTEGER
        x, y, z, t: CHARACTER
        call: INTEGER
        top: RECORD
    do     -- Initialize locals to values of arguments in original call:
        count := n; x := source; y := target; z := other

start:  if count > 0 then
                    -- Translation of hanoi (n–1, source, other, target):
            stack.put (create {RECORD}. make (count, 1, x, y, z))
            count := count – 1
            t := y ; y := z ; z := t
            goto start

after_1:    move (x , y)

                    -- Translation of hanoi (n–1, other, target, source):
            stack.put (create {RECORD}. make (count, 2, x, y, z))
            count := count – 1
            t := x ; x := z ; z := t
            goto start
        end

                    -- Translation of routine return:
after_2:if not stack.is_empty then
            top := stack.item    -- Top of stack
            count := top.count
            x := top.source ; y := top.target ; z := top.other
            call := top.call ; stack.remove
            if call = 2 then
                goto after_2
            else
                goto after_1
            end
        end
                -- No else clause: the routine terminates when
                -- (and only when) the stack is empty.
    end
```

This block is referred to below as SAVE_AND_ADAPT_1

Referred to below as MOVE

Referred to below as SAVE_AND_ADAPT_2

Referred to below as RETRIEVE

The body of *iterative_hanoi* derives from *hanoi* through systematic application of recursion elimination techniques:

D1 For every argument, introduce a local variable. The example uses a simple naming convention: *x* for *source* and so on.

D2 Assign on entry the value of the argument to the local variable, then work exclusively on that variable. This is necessary because a routine may not change the value of its arguments (*n* := *some_new_value* is invalid).

D3 Give a label, here start, to the routine's original first instruction (past the local variable initializations added by D2).

D4 Introduce another local variable, here *call*, with values identifying the different recursive calls in the body. Here there are two recursive calls, so *call* will have two possible values, arbitrarily chosen as **1** and **2**.

D5 Give a label, here after_**1** and after_**2**, to the instructions immediately following each recursive call.

D6 Replace every recursive call by instructions which:

- Push onto the stack an activation record containing the values of the local variables.
- Set the values of the locals representing arguments to the values of the call's actual arguments; here the recursive call replaces *n* by *n* – 1 and swaps the values of *other* and *target*, using the local variable *swap* for that purpose.
- Branch to the first instruction.

D7 At the end of the routine, add instructions which terminate the routines' execution only if the stack is empty, and otherwise:

- Restore the values of all local variables from the activation record at the top of the stack.
- Also from that record, obtain the call identification
- Branch to the appropriate post-recursive-call label among those set in D5.

This is the general scheme applicable to the derecursification of any recursive routine, whether a programmer is carrying it out manually, as we are now doing, or — the more common situation — compilers include it in the code they generate for routine calls.

We will see next how to simplify it — including **goto** removal — with the help of some deeper understanding of the program structure; in the meantime, make sure you fully understand this example of brute-force derecursification.

If, as I hope, you do find the transformation (if not the result) simple and clear, you may enjoy, as a historical aside, an anecdote reminding us that what is standard today was not always obvious. It is told by Jim Horning, a computer scientist well known for his own contributions, in particular to the area of formal methods:

***Naur & Horning* (*2006*)**

Touch of History: When recursion was thought impossible (as told by Jim Horning)

In the summer of 1961 I attended a lecture in Los Angeles by a little-known Danish computer scientist. His name was Peter Naur and his topic was the new language Algol 60. In the question period, the man next to me stood up. "It seems to me that there is an error in one of your slides."

Peter was puzzled, "No, I don't think so. Which slide?"

"The one that shows a routine calling itself. That's impossible to implement."

Peter was even more puzzled: "But we have implemented the whole language, and run all the examples through our compiler."

The man sat down, still muttering to himself, "Impossible! Impossible!". *I suspect that much of the audience agreed with him.*

At the time it was fairly common practice to allocate statically the memory for a routine's code, its local variables and its return address. The call stack had been independently invented at least twice in Europe, under different names, but was still not widely understood in America.

Slightly abridged from Jim Horning's blog at horningtales.blogspot.com/2006/07/recursion.html. Reproduced with permission.

The reference to independent inventions of the notion of call stack is probably to Friedrich Bauer from Munich, who used the term *Keller* (cellar), and Edsger Dijkstra from Holland, when implementing his own Algol 60 compiler.

Bauer (2005)

Simplifying the iterative version

The code given above looks formidable, especially against the simplicity of the original recursive version. Indeed, with a truly recursive algorithm like this one an iterative version will never reach the same elegance. But we can get close by reviewing the sources of complication:

- We may replace the **goto**s by structured programming constructs.
- By identifying invertible operations, we may limit the amount of information to be stored into and retrieved from the stack. ← *See I2, page 489.*
- In some cases (tail recursion) we may bypass the stack altogether.

The last two kinds of simplification can also be important for performance, since all this pushing and popping takes time, as well as space on the stack.

On the Hanoi example let us start by getting rid of the **goto** eyesores. To abstract from the details of the code we express the body of *iterative_hanoi* as

← *From page 492.*

```
INIT
start:    if count > 0 then
              SAVE_AND_ADAPT_1
              goto start
after_1:      MOVE
              SAVE_AND_ADAPT_2
              goto start
          end
after_2:if not stack.is_empty then
              RETRIEVE
              if call = 2 then goto after_2 else goto after_1 end
          end
```

count is an integer variable; the instructions I0, I1 and I2 can change its value.

with *SAVE_AND_ADAPT_1* representing the storing of information into the stack and change of values before the first call, *SAVE_AND_ADAPT_2* the same for the second call, *RETRIEVE* the retrieval from the stack of local variables including *call*, *MOVE* the basic move operation, and *INIT* the initialization of local variables from the arguments.

This is the example of **goto** structure that served (with abstract names for the instructions and conditions, I1, C1 etc.) as illustration in the discussion of **goto** removal. The result was

*← "Appendix: an example of goto removal", page 205. The resulting **goto**-less structure appears on page 206. The local variable over is initialized to **False**.*

```
from INIT until over loop
    from until count <= 0 loop
        SAVE_AND_ADAPT_1
    end
    from stop := stack.is_empty until stop loop
        RETRIEVE
        stop := (stack.is_empty or (call /= 2))
    end
    over := (stack.is_empty and (call = 2))
    if not over then MOVE ; SAVE_AND_ADAPT_2 end
end
```

which we can immediately simplify, getting rid in particular of the *stop* boolean variable:

```
from INIT until over loop
    from until count = 0 loop SAVE_AND_ADAPT_1 end
    from call := 2 until stack.is_empty or call = 1 loop RETRIEVE end
    over := (stack.is_empty and (call = 0))
    if not over then MOVE ; SAVE_AND_ADAPT_2 end
end
```

The simplifications result from an analysis of possible changes to the values of the variables:

- Since *count* can never become negative because of the precondition of *hanoi* and the test conditioning recursive calls, it is legitimate to replace that test, *count* <= 0, by *count* = 0.

- To get rid of *stop* we note that any value *call* gets out of *RETRIEVE* can only be 1 or 2, since these are the possible values stored onto the stack; so we can replace *call* /= 2 by *call* = 1, then set *call* to 2 the first time around so that this particular condition is only taken into account for the second and later iterations if any.

Tail recursion

A standard technique that helps reduce the overhead of stack pushing and popping relies on the observation that it is not necessary to store context information, and later retrieve it, if the algorithm does not need this information any more; this is the case in particular for a recursive call that is the *last* operation executed by an instance of the recursive routine.

This simplification applies to the *hanoi* example. The second recursive call is the last instruction executed by an activation of the routine. This means that *SAVE_AND_ADAPT_2* is not necessary, or more precisely that the only information it must preserve is *call*, since in getting back from a call you need to know whether it was an instance of the first or the second one: in the first case you need to pop the other values (*count*, *x*, *y*, *z*), in the second you don't.

A good compiler can detect tail recursion and apply this optimization to improve the performance of a recursive algorithm.

In the *hanoi* case it is superseded by another optimization, which almost entirely gets rid of the stack and which we will now see. You should, however, practice tail recursion elimination by implementing the above algorithm and removing the unneeded push operations.

→ *Exercise 14-E.14, page 504.*

Taking advantage of invertible functions

Using a stack to store the values before a call and retrieve them afterwards is the default technique and always works, but we saw earlier that an alternative exists: reverting the transformation of arguments. In the *hanoi* case this turns out to be possible for *all* arguments:

← *I2, page 489*.

- The transformation of *count* prior to each call, *count* := *count* – 1, has an obvious inverse: *count* := *count* + 1.
- For the other arguments, representing needles, the transformation is $swap_{23}$ for the first call and $swap_{12}$ for the second, if we call $swap_{ij}$ the operation that swaps the variables representing the i-th and j-th needles (for example $swap_{23}$ is $t := y$; $y := z$; $z := t$). But every $swap_{ij}$ is its own inverse: applying it a second time restores the original values.

So we do not actually need to store any of *count*, x, y and z on the stack: it suffices, at the time of a *RETRIEVE*, to apply the appropriate inverse operation. Specifically, *RETRIEVE* becomes:

```
"Retrieve the value of call"
count := count + 1
if call = 1 then swap23 else swap13 end
```

A stack remains necessary, but only to record and retrieve the values of *call*. The simplification becomes even more dramatic if we notice that *call* only has two possible values, 1 and 2, which were just a convention to identify the two recursive calls. Let us instead call them 1 and 0. There is a simple representation for a stack of 0/1 (or boolean) values: if you know for certain that the stack's height plus one cannot exceed the bit size of an integer — typically 64 on modern computers, until recently 32 —, just use a *single integer*, say s, for the stack. It is a matter of considering the 0s and the 1s of the binary representation, even if you do not know the details of number representation on your computer. The operations are:

```
s = 1           -- Is the stack empty?
s := 1          -- Initialize to an empty stack
s := 2 * s      -- Push a 0
s := 2 * s + 1  -- Push a 1
b := s \\ 2     -- Obtain (into b) the top of the stack (\\ is remainder)
s:= s // 2      -- Pop the stack (// is integer division)
```

The first is a boolean expression, the others are instructions.

Here is the result of a typical sequence of such instructions:

Instruction	*Goal*	*Result*	*Binary representation of s (leftmost zeroes omitted)*
$s := 1$	-- Start empty	$s = 1$	1
$s := 2 * s$	-- Push a 0	$s = 2$	1 0
$s := 2 * s + 1$	-- Push a 1	$s = 5$	1 0 1
$s := 2 * s + 1$	-- Push a 1	$s = 11$	1 0 1 1
$s := 2 * s$	-- Push a 0	$s = 22$	1 0 1 1 0
$s := s \mathbin{//} 2$	-- Pop	$s = 11$	1 0 1 1
$b := s \mathbin{\backslash\backslash} 2$	-- Get top	$b = 1$	

The binary representation of integers, shown in the last column, has the largest weights on the left ("*big-endian*" convention). The top of a non-empty stack is 0 if the number is even, 1 if it is odd.

This technique of using a single integer to represent a stack of boolean values can be used safely whenever you have a guaranteed limit on the stack size. In the *hanoi* example this is not a problem since 2^{63} or even 2^{31} are more moves than can be handled in any reasonable time.

Combining the previous observations leads to a simpler and more efficient form of the *iterative_hanoi* algorithm with arguments *n*, *source*, *target*, *other*:

```
from
    $count := n$ ; $x := source$ ; $y := target$ ; $z := other$ ; $s := 1$
until $over$ loop
    from until $count = 0$ loop
        $swap_{23}$ ; $s := 2 * s + 1$ ; $count := count - 1$
    end

    from $call := 0$ until $s = 1$ or $call = 1$ loop
        $call := s \backslash\backslash 2$ ; $s := s // 2$ ; $count := count + 1$
        if $call = 1$ then $swap_{23}$ else $swap_{13}$ end
    end
    $over := ((s = 1)$ and $(call = 0))$
    if not $over$ then
        $move (x, y)$
        $swap_{13}$ ; $s := 2 * s$ ; $count := count - 1$
    end
end
```

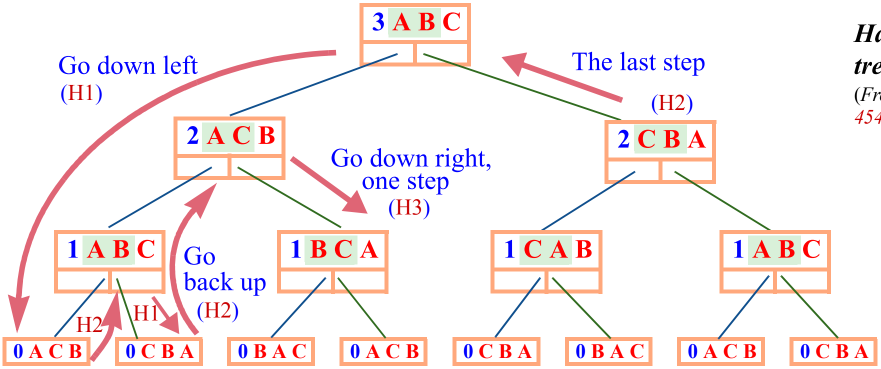
over is initialized to **False** *as usual.*

-- Go down left
-- (H1, see next page)

-- Go back up
-- (H2)

-- Visit node,
-- go down right
-- (H3)

Although this is the result of a systematic transformation and not the kind of program that you would normally write (recursion is simpler and clearer), it is interesting to follow the execution on this form too, relating it to the original recursive version and specifically to the binary *execution tree* from the beginning of this chapter, showing the execution as an inorder traversal:



Hanoi binary tree traversal
(*From pages 450 and 454*)

As noted next to the algorithm, it has three components:

H1 Go down left, as far as possible, until you reach a leaf. The leaves are at $n = 0$ (*count* = 0 in this version), although earlier figures showing this tree stopped at 1 since nothing visible from the outside happens at level 0.

H2 Go back up. As long as you are coming from the right just continue going up, since this corresponds to the second recursive call and there is nothing more to do with this instance of the routine.

H3 Having gone up one left branch, perform the visiting operation (move one disk from x to y, and go down *one* right branch).

This is repeated until, coming up from the right (H2), you find the stack empty.

When going down (H1, H3), you decrement *count* and swap y and z if going left (H1), x and z if going right (H3); when coming back up (H2), you restore the original values by incrementing *count* and doing the appropriate swap depending on whether you are coming back from the left or from the right — which you find out by looking at the top of the stack, meaning the parity of s as given by *call*.

14.10 KEY CONCEPTS LEARNED IN THIS CHAPTER

- It is often convenient to define a concept recursively, meaning that the definition uses one or more other instances of the concept itself.
- For the definition to be useful, any occurrence of the concept in its definition must apply it to a smaller target, and there must be at least one case for which the definition is non-recursive, so that any application of the definition reduces in the end to a combination of elementary cases.
- Recursive definitions can be useful in particular for routines, data structures and grammars.
- Any loop can be expressed in an equivalent recursive form, through a simple transformation.
- The other way around, any recursive algorithm has a recursion-free equivalent, but the transformation is more delicate; it requires changing the control flow, and recording the value of local information prior to every recursive call so as to retrieve it later, either by using a stack or by spotting invertible transformations.

New vocabulary

Activation
Activation record
Alpha-beta
Backtracking
Binary tree
Call chain
Depth-first
Direct recursion
Indirect recursion
Inorder
Iterative
Instance (of a routine)
Minimax
Non-creative
Postorder
Preorder
Recursion
Recursive
Recursive definition
Traversal

14-E EXERCISES

14-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

14-E.2 Too much recursion?

Is the definition of "recursive definition" a recursive definition? ← *Page 435.*

14-E.3 Binary search trees with repetitions

For every binary search tree routine in this chapter, rewrite the declaration (if needed) to permit multiple occurrences of a given *item* value in a tree as discussed after the initial definition. ← *Page 455.*

14-E.4 A programming language without program texts

This exercise addresses language processing techniques seen in an earlier chapter; the solution requires recursion.

→ See also exercises 16-E.4, page 617 to 16-E.6, page 618 in the inheritance chapter.

The goal is to write an interpreter and a compiler for an elementary programming language. To avoid dealing with concrete syntax, the tools will directly manipulate data structures rather than texts.

Our little language is called WASO (acronym for With Abstract Syntax Only) and has the following properties:

- The only data type is “integer”.
- Variables, all of integer type, do not need to be declared. A variable name is an arbitrary string.
- Integer constants can be used, such as 1.
- Integer expressions can be formed with addition, subtraction, multiplication and integer division.
- There are two kinds of instruction: assigning an expression to a variable, and printing the value of a variable.
- A WASO program consists of a sequence of assignments and a sequence of print instructions, either or both of which can be empty.
- The execution of a program consists of initializing all variables to zero, executing the assignments in sequence, and executing the print instructions in sequence.

So a typical program — written out here as if WASO had a textual representation (concrete syntax), although this is not part of the language definition — is:

```
assign
    x := 3
    y := 5
    x := 2 ∗ (x + (y // 3))
then
    print x
    print z
end
```

This is only one possible concrete syntax.

The execution of this program prints the single value 8.

The concrete syntax is only one of many possible choices. Another would use the keyword **print** instead of **then** and in the second clause list only the variables to be printed, without repeating **print**.

The assignment:

1 Write a set of classes, including *PROGRAM*, *ASSIGNMENT*, *PRINT* and *EXPRESSION*, with the associated features including creation procedures, to build abstract syntax trees representing WASO programs.

2 Add a class with a procedure that uses these classes and features to create an abstract syntax tree representing the above example program.

3 Add to class *PROGRAM* a procedure *write_out* that produces a textual (concrete) representation of a WASO program, as given out for the example. Run it on the example tree from step 2 and check that the output is the above text. *Hint*: you need a recursive procedure performing a traversal, similar to those introduced for binary trees in this chapter.

4 Write a WASO interpreter, in the form of a procedure *interpret* in class *PROGRAM* which executes the program and produces the expected output. Run it on the example and check the result (which as noted should be the single value 8).

5 Write a WASO-to-Eiffel compiler, in the form of a procedure *compile* in class *PROGRAM* which produces an Eiffel system implementing the semantics of the source WASO program: a root class with an appropriate creation procedure, and any other classes needed. Run it on the example; use Eiffel Studio to Eiffel-compile the output; run it on the example and check the result.

Terminology note: the result of step 5 is an *unparser*, producing a text representation from an internal representation such as an abstract syntax tree — the reverse of what a *parser* does.

14-E.5 Non-recursive insertion

Write a version of *put* for binary search trees using a loop rather than recursion. (**Hint**: you may use for inspiration the non-recursive version of the search function *has*.)

← *Page 458.*

14-E.6 Recursive reversal

Retaining the same assumptions (a list of stops is known through its first cell, of type *STOP*, giving access to the rest through repeated application of *next*), rewrite the function *reversed* from the discussion of references so that it uses recursion rather than a loop. (See also the next exercise.)

← *Page 261.*

14-E.7 Reversing a list, functional style

Write a recursive function that produces the reverse of a linked list (the argument and the result should be of type *LINKED_LIST* [*G*], from EiffelBase). Keep pointer manipulations to a minimum and remain as close as possible to the style of the *reversed* function given as an example of Haskell programming. Analyze the time and space complexity of your solution.

← *"Functional programming and functional languages", page 324.*

14-E.8 Backtracking curtailed

Adapt the general backtracking algorithm so that it keeps track of previously explored positions and discards any path leading to such a position. You may assume that *PATH* has a query *position* defining a path's terminal position.

← *Page 461.*

14-E.9 Cycles despised

Adapt the general backtracking algorithm so that it does not explore paths longer than *path_cutoff*, a given integer value.

← *Page 461.*

14-E.10 Properties of a function graph

(This exercise calls for mathematical analysis, not a programming solution.) In the successive approximations H_i of the graph of the Towers of Hanoi function, assuming three needles 'A', 'B', 'C':

← *"The towers, bottom-up", page 483.*

1 What is the number of pairs in H_i?

2 Give a mathematical formula for H_i.

14-E.11 Programming a function graph bottom-up

1 Devise a class of which every instance represents an arguments-result pair, of the form [(n, s, t, o], <…>], for the Towers of Hanoi function graph.

← *"The towers, bottom-up", page 483.*

2 Based on the preceding class, devise another to represent the function graph as a whole.

3 From this class and the rules [5] and [6] defining the function graph in the bottom-up interpretation of recursion, write a program that produces the i-th approximation of the graph, H_i, for any i. The algorithm may use loops, but it may not use recursion.

4 Use this program to print out sequences of moves (with source 'A' and target 'B') for a few values of i; check that the results coincide with those of the recursive procedure.

14-E.12 Bottom-up view of binary tree algorithms

Consider a recursive algorithm for binary tree traversal; you may choose preorder, inorder or postorder.

1 Taking inspiration from the bottom-up analysis of the Towers of Hanoi solution, devise a model to interpret the traversal as a function returning a sequence of nodes.

← *"The towers, bottom-up", page 483.*

2 Write a recursive "definition" of this function.

3 Express this "definition" as a fixpoint equation on the function graph, using T_i as the name of the graph for binary trees of height i.

4 Use the definition to produce (either manually or by writing a small program) T_5 for the example binary tree, and the resulting traversal order.

← *From the figure on page 447.*

14-E.13 Recursion without optimization

(This exercise requires access to a compiler for a programming language such as C or C++ with support for **goto** instructions.) Implement and test the direct iterative translation of the *hanoi* procedure, in its initial version using **goto**s and a stack without optimization.

← *iterative_hanoi*, *page 492.*

14-E.14 Saving on stack saving

1 Implement and test the **goto**-free iterative, stack-based version of the Tower of Hanoi problem.

← *Algorithm on page 495.*

2 Improve the solution through tail recursion optimization, avoiding unnecessary saves in the second call.

3 (Only if you have solved the previous exercise.) Apply the same optimization to the version using **goto** instructions.

14-E.15 Traversal without a stack

We saw that implementing recursion only requires a technique to invert the transformation of arguments in recursive calls; a stack is just one possible way to satisfy this requirement. Using a suitable inversion technique, implement binary tree traversal, for example inorder, non-recursively and without any stacks except possibly a stack of boolean values (or, equivalently, a bit in every node).

← *"Implementation of recursive routines", 14.9, page 486.*

Hint: temporarily overwrite tree links to remember where you came from.

Counter-hint: you could find a solution by running Web searches for the words *Deutsch*, *Schorr* and *Waite* (names of authors of a famous algorithm based on this idea). Don't; rather, design an algorithm, then look up existing references if you wish.

14-E.16 Transitive closure

(This exercise refers to a later chapter.) Restate the definition of transitive closure as a recursive definition.

→ *Page 513.*

14-E.17 Matrix algebra on BNF productions

(This exercise requires basic knowledge of linear algebra.) Consider a BNF production, such as the small example used in this chapter, or more extensive ones from earlier chapters, involving only Concatenation and Choice productions (no Repetition, as it can be replaced by combinations of the other two).

1 Treating concatenation of tokens as "multiplication" and alternative choices as "addition", show that it is possible to express the grammar as a matrix equation $X = A * X + B$, where X is the vector of nonterminals, A is a matrix of terminals and nonterminals, and B is a vector.

2 Discuss ways of solving this equation by following the model discussed for fixpoint equations.

15

Devising and engineering an algorithm: Topological Sort

One of the pleasures of learning computer science is to discover beautiful algorithms. In this chapter we explore an algorithm scheme with several claims to our attention: it is useful in many practical cases; it has a simple mathematical basis; it is particularly elegant; and it illustrates problem-solving techniques that you will find applicable in many other contexts.

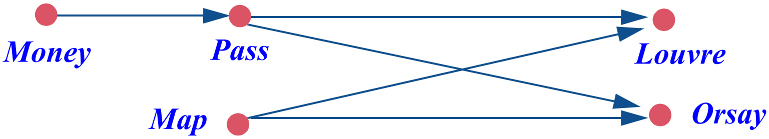
I will not throw a ready-made answer at you; instead we will develop the solution step by step from the description of the problem, starting with a mathematical analysis and continuing with a search for data structures ensuring both correctness and efficiency. We will not just devise an algorithm but strive for a complete, properly *engineered* solution that can satisfy practical needs. At the end of the chapter, we will draw the lessons of this example for both algorithm development and general software engineering.

15.1 The problem

Today is for culture: you want to visit the Louvre and the Orsay museum, in any order. Before visiting either you must get a map; you must also get a metro pass because your old one expired yesterday, but you cannot get a pass until you have gone to the bank or an Automatic Teller Machine to get some money. We may express these constraints as

[*Map*, *Louvre*], [*Map*, *Orsay*], [*Pass*, *Louvre*], [*Pass*, *Orsay*], [*Money*, *Pass*]

where [*x*, *y*] means "*x* must occur before *y*"; or we may represent them graphically



Ordering constraints

A **topological sort** of such a set of elements governed by ordering constraints is an enumeration of all the elements in an order that respects the constraints. Possible topological sorts in this example include

There are two more possibilities

Money, *Pass*, *Map*, *Louvre*, *Orsay*
Map, *Money*, *Pass*, *Orsay*, *Louvre*
Money, *Map*, *Pass*, *Louvre*, *Orsay*

but *Pass*, *Money*, *Map*, *Louvre*, *Orsay*, for example, would be incorrect since it violates the constraint [*Money*, *Pass*].

A topological sort problem may have:

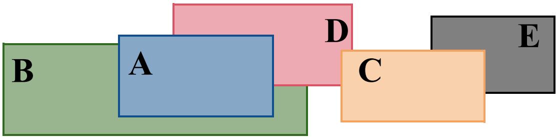
- Several solutions, as here.
- Exactly one solution.
- No solution, as will be the case if — and only if — the constraints include a *cycle*: a set of constraints of the form $[e_1, e_2]$, $[e_2, e_3]$, ..., $[e_n, e_1]$ for some $n \geq 1$. If we add [*Orsay*, *Money*] to the example, creating such a cycle, there can no longer be any solution since the constraints require both that *Money* occur before *Orsay* and the reverse.

If there is more than one solution, the problem is to produce one of them. Usually, any solution has an associated *cost*; then the goal will be to produce the solution with minimal cost. We will see where, in the algorithm, we can apply this criterion to choose between alternative solutions. Another variant of the problem would be to produce *all* solutions.

Example applications

The topological sort problem arises whenever we want to order a number of elements in conformance to some ordering constraints. This is a frequent problem; here are some examples.

- In a graphical display, consider rectangles that may overlap. Some are "above" others, as illustrated. You need an algorithm that will display them in an order respecting these constraints, so that in the end the figure appears as intended. This is a topological sort problem. In the illustrated example, the constraints are [B, A], [D, A], [D, C], [B, D], [E, C] (where [x, y] means "x must not hide any part of y"); a possible solution is the order B D E A C.



Rectangles with precedence constraints

- When an industrial installation such as a power plant or an airplane undergoes *maintenance*, the schedule is determined from a set of tasks to be performed and a set of ordering constraints between them; for example structural work on an element must come before repainting it. A topological sort yields a schedule of tasks compatible with these constraints.
- Another application occurs in *project management*, especially software project management. If the problem domain is technical, the project should produce and maintain a *glossary* of the technical terms involved. (Misunderstandings between application area experts and software developers are a major source of errors and deficiencies in software systems.) The definition of any of these terms may involve other terms that have their own entries. The entries might appear in alphabetical order, as in a dictionary, but it may also be useful to have a version of the glossary that can be read in sequence, with the definition of any term appearing before any definition that uses the term. Producing such a list is a topological sort problem.

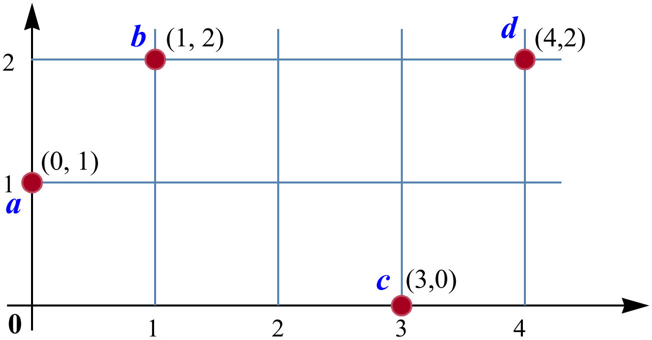
→ The requirements document of a software project should include such a glossary: "The glossary", page 722.

- You might want to see a list of the *features in a class* that shows the features not in the order listed (grouped, by default, into feature categories) but in one that facilitates sequential reading by guaranteeing that no call to a feature occurs before the feature's declaration.
- Compiling object-oriented programs efficiently can take advantage of topological sort for implementing *inheritance*, specifically *dynamic binding* as discussed in the next chapter. The issue is to number the classes of a possibly large program so that the number assigned to a class is close to those of its *descendants*, classes that inherit from it directly or indirectly. Using "inherits from" as the ordering relation, the EiffelStudio compiler relies on topological sort to achieve dramatic space optimizations, essential to the viability of the O-O approach.

→ "A peek at the implementation", 16.8, page 575

Points in a plane

Another example provides a convenient visualization of the problem. Consider points in a plane:



A finite set of points

We introduce a relation $\ll$ by stating that $p_1 \ll p_2$ holds for any two points p_1 of coordinates (x_1, y_1) and p_2 of coordinates (x_2, y_2) if they satisfy all of:

- $x_1 \leq x_2$
- $y_1 \leq y_2$
- $p_1 \neq p_2$ (the two points are not the same).

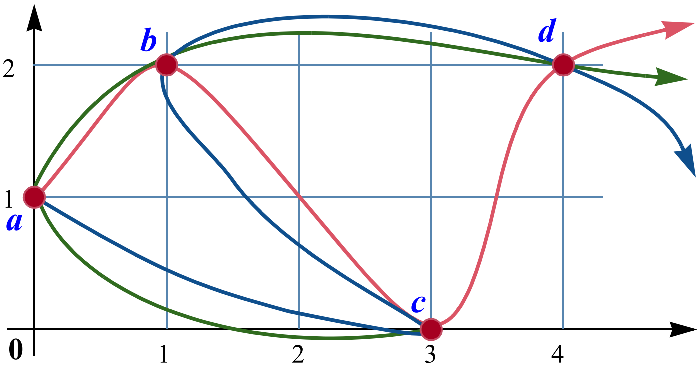
For the four points shown in the figure, the following hold:

$a \ll b \qquad a \ll d \qquad b \ll d \qquad c \ll d$

A topological sort for this relation is any enumeration of the points that lists p before q for any two points such that $p \ll q$. For our four points there are three such enumerations:

a, b, c, d
a, c, b, d
c, a, b, d

which we may visualize as three different traversals of the set of points in the preceding figure:



Three topological sorts of a set of points

On the other hand, the enumeration a, d, b, c is not compatible with the $\ll$ relation since the property $c \ll d$ requires c to appear before d.

15.2 THE BASIS FOR TOPOLOGICAL SORT

The problem discussed in this chapter has a concise mathematical statement:

> ***Definition*: The topological sort problem**
>
> Given an acyclic relation r on a finite set, find a total order relation of which r is a subset.

This definition is made possible by simple mathematical notions — relations as sets, acyclic relation, order relation (total or not) — which we will now review.

Binary relations

> ***Definition*: Relation**
>
> A **relation** over a set A (short for *binary* relation) is a set of pairs of the form $[x, y]$ where both elements of the pair, x and y, are members of A.

More general relations, as in relational databases, are sets of tuples of n elements belonging to any given sets, for any n. The relations of this chapter are binary ($n = 2$) and over a single set A.

An example relation over the set $\{1, 2, 3\}$ is

$\{[1, 2], [1, 3], [2, 3]\}$

which we may call "<" since it represents "less than" (meaning that it contains all the pairs $[x, y]$, with x and y both in $\{1, 2, 3\}$, such that x is less than y).

We may use relations to describe the earlier examples:

- A relation *below* over a set of rectangles, containing all rectangle pairs $[x, y]$ such that the display must show points of y rather than x in any area where they overlap.
- A relation *before*, the set of pairs $\{[Map, Louvre], [Map, Orsay], \ldots\}$; it is a relation over the set $\{Money, Pass, Map, Louvre, Orsay\}$, containing all pairs $[x, y]$ for which x must happen before y.
- A relation *used_in* over a set of glossary terms, containing all pairs $[x, y]$ such that the definition of the term y uses the term x.
- A relation *called_by* over the features of a class, containing all pairs $[x, y]$ such that the body of feature y contains a call to feature x.
- A relation $\ll$ over points, the set of pairs $\{[a, b], [a, d], [b, d], [c, d]\}$.

Acyclic relations

Our examples so far are all *acyclic relation*s, a notion defined as follows:

Definition: Acyclic relation

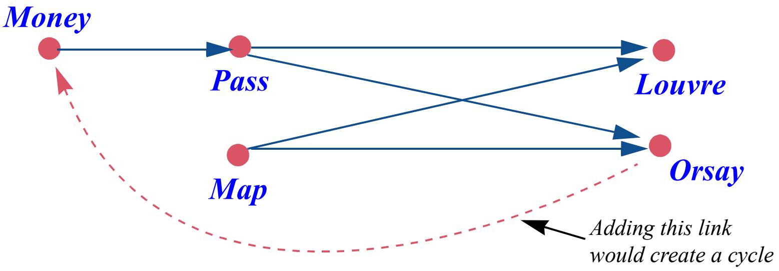
A relation is **acyclic** if it has no cycle.

with:

Definition: Cycle in a relation

A **cycle** for a relation r over a set A is a sequence $x_1, \dots x_m$ ($m \geq 2$) of elements of A such that all successive pairs $[x_i, x_{i+1}]$ for $1 \leq i < m$ belong to r, and $x_m = x_1$.

The relation *before* as given has no cycles:



A relation describing ordering constraints

Adding the pair [*Orsay*, *Money*] would create a cycle: *Money*, *Pass*, *Orsay*, *Money*.

The simplest case of a cycle for a relation r occurs (with $m = 2$) for an element x such that $[x, x] \in r$.

To succeed, topological sort requires an acyclic relation, although for practical considerations we will look for an algorithm that can partially process constraints involving a cycle.

If the underlying set is finite, acyclic relations have an important property, crucial to the topological sort algorithm:

No-Predecessor theorem

For any acyclic relation r over a non-empty finite set A, there exists an element x of A with no predecessors for r.

relying on a notion of "predecessor":

Definition: Predecessor

A **predecessor** of an element y for a relation r is an element x such that the pair $[x, y]$ belongs to r.

The proof of the No-Predecessor theorem is by contradiction. Assume the theorem does not hold; then every element in A has at least one predecessor. Let x_1 be some element in A (we may indeed find such an x_1 since the theorem assumes A to be non-empty). By the hypothesis, x_1 has at least one predecessor; let us pick one and call it x_2. By the same reason x_2 also has at least one predecessor, so we may again pick x_3 such that $[x_3, x_2]$ is in r. Continuing this way gives an infinite sequence such that $[x_{i+1}, x_i]$ belongs to r for every $i \geq 1$. Because A is a finite set, the sequence has to repeat elements. More precisely: the elements x_1, x_2, … x_{n+1}, where n is the number of elements of A, cannot all be different; there must be integers i and j, with $1 \leq i < j \leq n + 1$, such that $x_i = x_j$. But then x_j, $x_{j\text{-}1,}$ … x_i is a cycle for r, which is impossible.

This is a *constructive* proof, which we will directly use in devising the topological sort algorithm: to produce an enumeration of the elements, the algorithm will pick, at every iteration, an element that has no predecessor in the remaining order relation.

The condition that A is finite is essential to the proof. The theorem does not apply to infinite sets; for example, the relation "less than" on mathematical integers is acyclic, but every element has predecessors.

Order relations

The idea of topological sort is to embed a given acyclic relation in a *total order* relation. To define this notion we must first consider plain *order* relations.

Definition: Order relation (strict, possibly partial)

A relation is an **order** relation if it satisfies the following properties for any elements x, y, z of the underlying set X:

O1 **Irreflexive**: the relation has no pair of the form $[x, x]$.

O2 **Transitive**: whenever the relation contains a pair $[x, y]$ and a pair $[y, z]$ (whose first element is the same as the second element of the first pair), it also contains the pair $[x, z]$.

Such an order relation is also:

O3 **Asymmetric**: whenever it contains a pair $[x, y]$, it does *not* contain the pair $[y, x]$. (Proof: if it contained both, transitivity implies that it would also contain $[x, x]$, violating irreflexivity.)

The full name for order relations as defined above is: *strict and possibly partial* order relation. Our order relations (also the "total" ones seen next) are *strict*, in the same sense that "<" denotes "*strictly* less than". It is also possible to work with the nonstrict versions, such as "≤", less than or equal.

→ See "Appendix: terminology note on order relations", 15.7, page 546. Also, do exercise 15-E.3, page 547 to explore the relationship between strict and nonstrict versions.

The relation "<" on $\{1, 2, 3\}$ (or any other set of integers) is an order relation. So is the relation $\ll$ on points. Our other acyclic relations — *before* between tasks, *used_in* between dictionary entries, *called_by* between features — are irreflexive and asymmetric, but not necessarily transitive, so they are not order relations; we will see next how to obtain transitive versions.

Order relations *vs* acyclic relations

Order relations are closely connected with acyclic relations. In one direction the connection is straightforward:

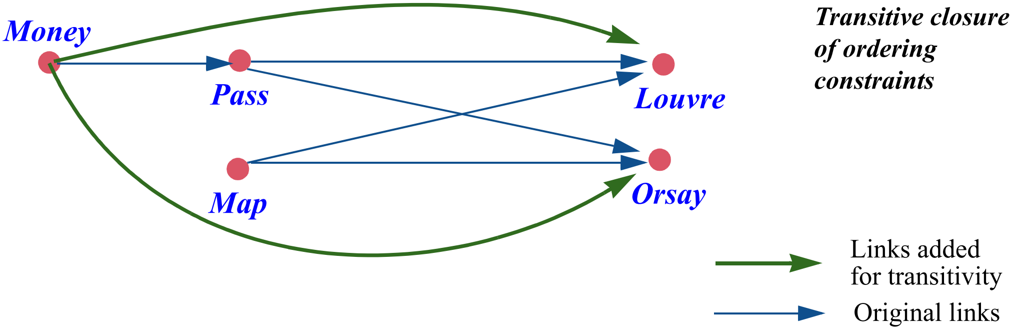
> ***Theorem*: Acyclic and order relations (1)**
>
> Any order relation (and more generally any subset of an order relation) is acyclic.

The proof is by contradiction. Assume a cycle $x_1, x_2, x_3, \dots x_m$ where x_m is the same as x_1. By transitivity (O2) this implies that $[x_1, x_1]$ is also in the relation; that is impossible because of irreflexivity (O1).

This generalizes the above proof that asymmetry — the impossibility of having both pairs $[x, y]$ and $[y, x]$ (O3) — follows from O1 and O2. Such a case is indeed a cycle with just two elements. Similarly, the pair $[x, x]$, ruled out by irreflexivity, would be a cycle with just one element.

There is also an interesting property the other way around: the *transitive closure* of an acyclic relation is an order relation. Informally, the transitive closure of a relation is a version of the relation made transitive by following the original relation's links as many times as possible.

This can be illustrated on the relation *before* expressing ordering constraints between tasks. The relation is irreflexive, asymmetric and acyclic; it is not transitive since it contains the pairs [*Money*, *Pass*] and [*Pass*, *Louvre*] but not [*Money*, *Louvre*]. We can make such a relation transitive by adding all pairs of the form $[x, z]$ for which the original includes both $[x, y]$ and $[y, z]$ for some y, repeatedly until there are no more pairs to be added. The result of this process is the transitive closure of the original relation. In the example it adds just two links:



Transitive closure of ordering constraints

For the relation *called_by* between features, the transitive closure is the relation that holds between x and y if y calls x directly or indirectly. For a relation *child* among persons, denoting the set of pairs $[x, y]$ such that person x is a child of y, the transitive closure is the relation connecting any two persons x and y such that y is a descendant, direct or indirect, of x.

The transitive closure of a relation r is written r^+, so we may state that $child^+ = descendant$. Here is a precise definition:

← Same use of + as in language theory; see "Repetition", page 301

←Exercise 14-E.16, page 504 requests a recursive variant of this definition.

Definition: Transitive closure of a relation

The **transitive closure** r^+ of a relation r over a set A is the relation containing all pairs of the form $[x_1, x_m]$ for some sequence of elements $x_1, \ldots x_m$ $(m \geq 2)$ such that all $[x_i, x_{i+1}]$ pairs for $1 \leq i < m$ belong to r.

Transitive closure gives us the other side of the relation between acyclic relations and order relations:

Theorem: Acyclic and order relations (2)

The transitive closure of any acyclic relation is an order relation.

Proof: the transitive closure of any relation r is obviously transitive, so all we have to show is that it is irreflexive for an acyclic r. Assume it is not. This means that there exists an element x such that $[x, x]$ belongs to r^+. By the definition of transitive closure, there must be a sequence of elements $x_1, \dots x_m$ $(m \geq 2)$ such that all $[x_i, x_{i+1}]$ pairs for $1 \leq i < m$ belong to r and that both x_1 and x_m are x. But this is a cycle for r, and hence impossible.

This result shows that we may view an acyclic relation as the "germ" of an order relation. Taking its transitive closure gives us a true order relation. This corresponds to the intuition behind relations such as *before* between tasks: if the constraints specify that task *Money* must precede *Pass*, and also that *Pass* must precede *Louvre*, we naturally understand that x must precede z. In other words, we instinctively take the transitive closure. But when it comes to preparing the input data for a scheduling program, or another program that will perform a topological sort, we will want to list basic constraints only, not their full transitive closure. That is why topological sort can use an acyclic relation as its input. (Many presentations of topological sort start from an order relation, but this is more specific than required.)

Computing a transitive closure is a computationally expensive operation, but we do not need to perform it explicitly; the topological sort algorithm will work directly from the acyclic relation.

Total orders

To describe the output of topological sorting we need a specialization of the notion of order relation: *total* order relation. For a finite set we may view a total order simply as an enumeration of the underlying set's elements, each appearing once, such as *Money*, *Pass*, *Map*, *Louvre*, *Orsay*; but the concept is more general:

← Briefly encountered in the study of recursion: "Binary search trees", page 454.

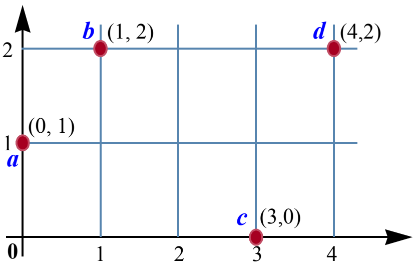
Definition: total order relation (strict)

A total order is an order relation that additionally is:

O4 **Total**: for any a and b, one of the following holds: $[a, b]$ is in the relation; $[b, a]$ is in the relation; $a = b$.

← Meaning a relation that's irreflexive (O1, page 511) and transitive (O2, page 511); it's asymmetric as a result (O3, page 512).

To understand condition O4, note that we know from asymmetry (O3) that at most one of the first two possibilities may hold, and from irreflexivity (O1) that the last possibility is exclusive of the other two. So *at most one* of the three *may* hold. What the new condition adds is that one of the three *does* hold.



(*Figure from page 507.*)

The relation "<" on integers is also total. But not every order is total. Our $\ll$ relation on points in a plane is an order relation, as we have seen; but it is not total since this would mean that for any two different points p_1 and p_2 either $p_1 \ll p_2$ or $p_2 \ll p_1$ holds. That's not the case for the pair $[a, c]$ since neither $a \ll c$ nor $c \ll a$ holds. There is another counter-example, the pair $[b, c]$.

Many total orders exist on this set of four points; in fact, any enumeration of them — that is to say, any ordered list that includes each of them exactly once — yields a total order t, defined as follows: the pair $[p, q]$ is in t if and only if p appears before q in the enumeration. For example the enumeration $[a, b, c, d]$ defines the total order

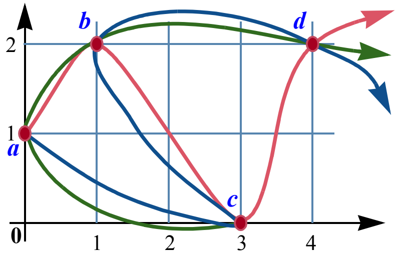
$$\{[a, b], [a, c], [a, d], \\ [b, c], [b, d], \\ [c, d]\} \quad [1]$$

Conversely, any total order on a finite set defines a single enumeration.

→ The proof is exercise 15-E.3, page 547.

Such a total order is a topological sort of the original relation — in our example, the relation $\ll$ — if and only if it is *compatible* with it, meaning that whenever $p \ll q$ the element p appears before q in the total order. We have seen that three total orders satisfy this requirement for the example: expressed as enumerations they are a, b, c, d (given in [1] as a set of pairs); a, c, b, d; and c, a, b, d.

What does "*compatible*" precisely mean? This notion is actually easy to specify thanks to the definition of relations as sets of pairs. To say that a total order such as the enumeration a, b, c, d, is compatible with a given (acyclic) relation is simply to say that the set of pairs of that relation is a **subset** of the total order's set of pairs: every pair in the order relation is also a pair of the total order relation. In our example the relation $\ll$ is the set of pairs



(*Figure from page 508.*)

$$\{[a, b], [a, d], [b, d], [c, d]\} \quad [2]$$

and is indeed a subset of the set of pairs [1] of the total order. The subset property expresses that whenever the given constraints specify a certain order between two elements, the output of the algorithm must list these elements in that order.

This yields the definition of "topological sort", describing our task as finding a total order of which the given order is a subset.

← Page 509.

Acyclic relations have a topological sort

The absence of cycles is clearly a necessary condition for the existence of a topological sort (a total order that includes the original relation). What about the other way around: if we have an acyclic relation, can we always produce a topological sort — a total order that includes it?

Indeed we can:

Topological Sort theorem

For any acyclic relation r over a finite set A, there exists a total order relation t over A such that $r \subseteq t$.

← *"Theorem: Acyclic and order relations (2)", page 513.*

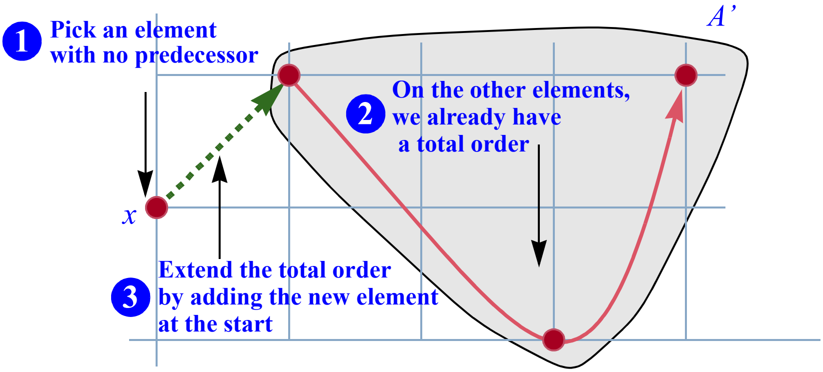
→ *See exercise 15-E.5, page 548.*

To prove this theorem we may use the observation that $r \subseteq r^+$, where r^+ is the transitive closure of r, and the previously proved property that r^+ is an order relation; it suffices to extend r^+ to a *total* order relation. But another proof is more interesting for our purposes. It is a *constructive* proof (relying on the No-Predecessor theorem) and will allow us directly to deduce an algorithm scheme.

This proof is by induction on the number of elements n in the set A. If $n = 0$, the set is empty; the only possible relation is the empty relation (the empty set of pairs of elements of A), which is a total order. This proves the base step.

If you prefer, you can use as base step the case $n = 1$, for which A consists of a single element x; even though A is not empty then, the only acyclic relation in A is the empty relation again, since if a relation has at least one pair that pair must be $[x, x]$, which would create a cycle.

For the induction step assume that the theorem holds for sets of n elements and consider an acyclic relation on a set A of $n + 1$ elements. The figure gives the idea of the proof:


A'
1 Pick an element with no predecessor
2 On the other elements, we already have a total order
3 Extend the total order by adding the new element at the start
x

Extending an incomplete topological sort

The No-Predecessor theorem tells us that *A* has at least one element without predecessors. Let *x* be such an element. Let *A'* be the set consisting of all elements of *A* except *x*, and *r'* the relation on *A'* consisting of all pairs of *r* except those involving *x*. Clearly, *r'* is an acyclic relation over *A'*. By the induction hypothesis, since *A'* has *n* elements, there exists a total order *t'* over *A'* that is compatible with *r'* (that is to say, $r' \subseteq t'$). Now consider the relation *t* over *A* consisting of the following pairs:

← Page 510.

- All the pairs in *t'*.
- All pairs of the form [*x*, *y*] where *y* is an element of *A'*.

  If you prefer to think of a total order as an enumeration, you may just view *t* as the enumeration of the elements of *A* that starts with *x* and continues with the enumeration of the elements of *A'* given by *t'*.

It is easy to see that *t* is a total order, and that $r \subseteq t$; this gives us a total order compatible with *r*, and proves the theorem.

The Topological Sort theorem is the mathematical justification for the program that we are now going to build; better yet, its proof directly suggests the algorithm's basic idea.

15.3 PRACTICAL CONSIDERATIONS

With the theoretical basis clear, we can start looking for a software solution. The core is a topological sort *algorithm*, but first we must examine performance constraints and define a software engineering framework.

Performance requirements

What can we expect to achieve in time and space complexity?

The inputs to the algorithm are a set of elements and a set of constraints. Let *n* be the number of elements and *m* the number of constraints.

The algorithm must (in the case of an acyclic relation) perform:

- At least one operation for every constraint (since ignoring any single constraint might make any particular output order wrong).
- At least one operation for every element, if only to add it to the output.

So the best time complexity that we may hope for is $\mathbf{O}\ (m + n)$.

The more surprising result is that the topological algorithm developed below will actually achieve this theoretical ideal, both in time and in space.

Class framework

A purely algorithmic solution would use a function of the form

```
topologically_sorted (elements: …; constraints: …): LIST […]
          -- Enumeration of the members of elements,
          -- in an order compatible with the constraints.
```

with appropriate input types to represent the sets *elements* and *constraints* (corresponding to our earlier *A* and *r*). It is better — as the remaining development will progressively show — to use an object-oriented approach with a class *TOPOLOGICAL_SORTER*, any instance of which represents an instance of the topological sort problem. The data structures representing the elements and constraints will be attributes of the class, set up through initialization procedures such as *record_element* and *record_constraint*.

Instead of a function *topologically_sorted* as above we will have:

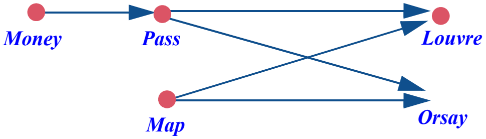
- A procedure *process*, performing the topological sort process.
- A query *sorted*, returning a list of elements, as computed by *process*.

This framework gives us more flexibility, and will accommodate many useful additional features.

Input and output

The sets of elements *A* and constraints *r* might come from many different sources. For example we might have a file listing the constraints, one per line:

```
Map    Louvre
Map    Orsay
Pass   Louvre
Pass   Orsay
Money  Pass
```



(*From the figure on page 505.*)

It may be useful to have a separate file listing all elements, or at least elements *not* involved in any constraint (we cannot guess such elements from the constraints, but they should still be part of the output).

In another setup, the input might have been entered interactively using a program or a Web form. In examples such as ordering rectangles on a screen, terms in a glossary or features of a class, the format will again be different.

To ensure generality we make our basic class generic: it becomes *TOPOLOGICAL_SORTER* [*G*], where the parameter *G* represents the type of the elements. Then the result of the query *sorted* — denoting the topologically sorted list of elements computed by *process* —is of type *LIST* [*G*]; the two initialization procedures cited have signatures

```
record_element (e: G)
record_constraint (e, f : G)
```

Overall form of the algorithm

Consider an acyclic relation *r* over a set of elements *A*; since we need program names, let us assume — without prejudging its implementation choices — that the class *TOPOLOGICAL_SORTER* has them available through queries *constraints* and *elements*. The general scheme for the topological sort algorithm in procedure *process* is:

```
from … until elements.is_empty loop
    "Let x be an element without predecessors for the constraints"
    "Produce x as the next element of sorted"
    "Remove x from the set of elements"
    "Remove all pairs starting with x from the set of constraints"
end
```

Suitably refined, this form will work if we indeed start from an acyclic relation (or, as a special case, an order relation).

> The four pseudocode instructions shown above recur throughout the rest of this chapter; to avoid monopolizing attention, they will no longer use the special color marking pseudocode (except in one case where the attention will be required). The normal pseudocode convention resumes in the next chapter.

← *"Touch of Style: Highlighting pseudocode", page 109. The exception is in the final form of the core loop, page 536.*

The No-Predecessor and Topological Sort theorems give the justification, which we should express through a loop invariant and variant (did the spectacle of an invariant-less loop make you scream? — I hope it did):

← *Pages 510 and 516.*

```
process
        -- Produce in sorted an enumeration of the members of elements,
        -- in an order compatible with constraints.
    require
          -- "constraints describes an acyclic relation on elements"
    do
     from
        create {...} sorted.make
     invariant
          -- "constraints describes an acyclic relation on elements"
     until
        elements.is_empty
     loop -- As before, except for explicit use of the result list sorted:
        "Let x be a member of elements without predecessors for constraints"
        sorted.extend (x)
        "Remove x from elements"
        "Remove from constraints all pairs starting with x"
     variant
          elements.count
     end
    ensure
     -- "sorted is a topological sort of elements according to constraints"
    end
```

Cycles in the constraints

This version of the algorithm scheme is correct in principle but not suitable for most real-life applications of topological sorting.

The problem is the precondition, which requires *r* to be an acyclic relation. A topological sort program gets its input in the form of individual ordering constraints, for example [*Map*, *Louvre*], [*Map*, *Orsay*] as above. Such input may have been prepared by humans, and we cannot be sure that it is error-free. (In industrial plant maintenance, there may be thousands of tasks and tens of thousands of constraints between them.)

In the glossary example, we may hope that no set of two or more terms all reference each other in their definitions (thereby creating a cycle), but we have no way to enforce this rule on glossary authors. The expectation is in fact the other way around: the glossary's author will expect a program that can be told: "*Order these entries so that definition always comes before first use — and by the way, if you find any mutually referential entries, tell me what they are, so that I can improve the definitions*".

Similarly, the task of ordering the features of a class so that declarations appear before calls is impossible in the case of indirect recursion (direct recursion is fine), even though this is not an error. It may still be interesting to apply a topological sort algorithm to the non-cyclic part of the call graph, and report any remaining cycles.

These considerations suggest that *process* renounce its above contract

```
    require
                    -- "constraints describes an acyclic relation on elements"
    ensure
                    -- "sorted is a topological sort of elements,
                    -- according to constraints"
```

in favor of a more realistic one:

```
    -- (No precondition.)
    ensure
        -- "sorted is a topological sort, according to constraints, of all the
        -- members of elements not involved in a cycle"
```

This is not enough yet, since the class should be able to report to its clients that the input contains a cycle — and to say what elements are involved. One way to provide this functionality would be through a boolean-valued function

```
has_cycle: BOOLEAN
        -- Is the relation represented by elements and constraints
        -- not acyclic?
    do … end
```

or, more informatively, a function that returns the list of elements involved in a cycle (void if and only if *has_cycle* is false). This is conceptually sound, but not the best approach because it is computationally too expensive. Finding cycles — the job of a function *has_cycle* — is essentially as hard, in time and space complexity, as topological sort proper; but if we *attempt* to do a topological sort without the precondition, we may at hardly any extra cost find any cycles in the process.

The No-Predecessor theorem tells us indeed how we can find cycles as a side bonus of a topological sort process: ← *Pages 510.*

- As shown in the loop above, we look at each stage, as long as the set of elements is not empty, for an element without predecessors.
- The theorem indicates that if the relation is acyclic we will always find such an element.
- If we *cannot* find an element that has no predecessors and the set *elements* is not empty, we know — from the theorem — that the remaining elements are all involved in at least one cycle. We can terminate the algorithm and report that a full topological sort is impossible. This is a graceful form of termination, since we will have topologically sorted the elements that are not in cycles, and will be able (from the remaining *elements* and *constraints*) to tell the client which elements and constraints cause the problem.

In this scheme, used in the rest of this chapter, the topological sort routine has no precondition and the loop invariant, instead of

-- "*constraints* describes an acyclic relation on *elements*"

gets weakened to:

-- "*constraints* describes a subset of the original relation on *elements*

with the consequences that

-- "Any cycle in *constraints* was present in the original relation"

and

-- "*constraints* describes an acyclic relation if the original was acyclic"

This is the basis we should retain. As a consequence, we can no longer use the loop exit condition *elements*.*is_empty* as above, since a non-empty *elements* does not guarantee any more that we may correctly execute the instruction

"Let *x* be a member of *elements* without predecessors for *constraints*"

As the new exit condition, we will simply have

"No member of *elements* is without predecessors for *constraints*"

whose negation — there is at least an element without predecessors — guarantees that the loop body can find the next candidate element for output.

Overall class organization

We can now define the overall form of the class that will serve as the framework for the solution:

```
class
    TOPOLOGICAL_SORTER [G –> HASHABLE]

feature {NONE} -- Internal data structures

        … See next sections …

feature -- Initialization

    record_element (a: G)
            -- Include a in the set of elements.
        require
            not_sorted: not done
        do
             … See next sections …
        end

    record_constraint (a, b: G)
        -- Include [a, b] in the constraints.
        require
            not_sorted: not done
        do
             … See next sections …
        end
feature -- Status report
    done: BOOLEAN
            -- Has topological sort been performed?

feature -- Element change

    process
            -- Perform a topological sort over all applicable elements.
            -- Results accessible through sorted, cycle_found and cyclists.
        require
            not_sorted: not done
        do
             … See next sections…
        ensure
            sorted: done
        end
```

→ *The routine body appears on page 526 (revised, page 536).*

```
feature -- Access

    cycle_found: BOOLEAN
            -- Did the original constraint imply a cycle?

    cyclists: LIST [G]
            -- Elements involved in any cycle.

    sorted: LIST [G]
            -- List, in an order respecting the constraints, of all
            -- the elements that can be ordered in that way

feature -- Status setting

    reset
        -- Allow further updates of the elements and constraints.
        do
            done := False
            cycle_found := False ; cyclists := Void ; processed_count := 0
        ensure
            fresh: not done
        end

invariant
    elements_exist: elements /= Void
    constraints_exist: constraints /= Void
    cyclists_only_if_cycle: done implies (cycle_found = (cyclists /= Void))
end
```

The feature clauses have been listed in an order facilitating sequential reading rather than the recommended standard order, which a final version should respect.

The class is generic; the generic parameter *G* represents the type of elements. The notation *G –> HASHABLE* means that we are "constraining" *G* (a notion introduced in the next chapter) by type *HASHABLE*; the reason for the constraint will emerge a little later.

→ "Constrained genericity", page 596. See "Numbering the elements", page 531.

The algorithm will rely on internal data structures, which we will devise in the next sections; the corresponding features do not need to be available to clients, so they will all be declared under **feature** {*NONE*}.

← Meaning they are secret; see "Information hiding: modifying fields", page 240.

Once *process* has done its job, it will make its results available to clients through several related queries:

- The boolean *done*, enabling clients to find out whether a topological sort has indeed been performed; it is false after initialization.
- The list *sorted*, giving an order compatible with the constraints for elements not involved in a cycle.

- The boolean *cycle_found*, to indicate whether any elements were determined to participate in one or more cycles.

- The list of all these cycle-involved elements, which we accordingly call *cyclists*. The invariant clause cyclists_only_if_cycle tells us that it is only meaningful if *cycle_found* is true.

So a typical use of the class by a client wishing to perform a topological sort is:

```
your_structure: TOPOLOGICAL_SORTER [YOUR_ELEMENT_TYPE]
…
create your_structure

… Calls of the form your_structure.record_element (x) to record elements
… and your_structure.record_constraint (x, y) to record constraints …

your_structure.process

-- The topologically sorted elements are now available,
-- in the correct order, as your_structure.sorted.

if your_structure.cycle_found then
    -- The elements involved in cycles are now available
    -- through your_structure.cyclists …
end
```

It would be desirable for consistency to equip the queries *sorted*, *cycle_found* and *cyclists* with the precondition *done*, but we omit this for the moment.

There is, however, a precondition **not** *done* for the initialization procedures *record_element* and *record_constraint*, as well as for the topological sort procedure *process* which has the postcondition *done*. This enforces the rule that as a client you should first set up the elements and constraints, then call *process*. As a result, it is an error to call *process* several times in succession on the same class instance; since the constraints have not changed, this would make no sense (although you may of course reuse the query results as many times as you wish). The procedure *reset* is there in case you explicitly want to add elements and constraints after a call to *process*, in preparation for a new call to *process*.

Procedure *reset* simply sets *done* to false, without clearing the previous elements and constraints. We might add a procedure *forget* that calls *reset* and clears all data structures. But it is just as reasonable to assume that, in this case, the client will create a new instance of *TOPOLOGICAL_SORTER*.

15.4 BASIC ALGORITHM

We can now start to provide a full implementation of the key part of the solution, procedure *process*.

The loop

We already had a general algorithm for *process*; adapted in light of all subsequent observations (loosening the invariant, using the feature *sorted* which represents the result in *TOPOLOGICAL_SORTER*), it reduces to this:

← *Page* 520.

```
from
        create {…} sorted.make
until
     "No element is without predecessors"
loop
     "Let x be an element without predecessors"
        sorted.extend (x)
     "Remove x from the set of elements"
     "Remove all constraints starting with x"
end

if "Any elements remain" then -- Report cycle:
    cycle_found := True
    "Insert these elements into cyclists"
end
```

All that remains — do not rejoice too soon, major decisions still lie ahead — is to refine the pseudocode elements into actual program text. The final part (reporting cycles) will be a straightforward consequence of the rest; this leaves the four highlighted operations, in fact just three since we can treat the first two (finding out if there is an element without predecessors, and if so get one such element) as a single operation. They will be the focus of our search for an efficient algorithm:

Topological sort: the basic operations

T1 Find an element without predecessors — or report there isn't any.

T2 Given an element *x*, remove it from the set of elements.

T3 Given an element *x*, remove from the set of constraints all that start with it (that is to say, all pairs of the form [*x*, *y*] for some *y*).

We must find a representation for the elements and constraints that makes these operations as efficient as possible. The data structures will appear as secret features in the section reserved for that purpose in the class text.

← The clause starting with **feature** *{NONE} on page 523.*

That class sketch left two other routine bodies unfilled: *add_element* and *add_constraint*. We will need to complete them based on the data structures that we devise.

A "natural" choice of data structures

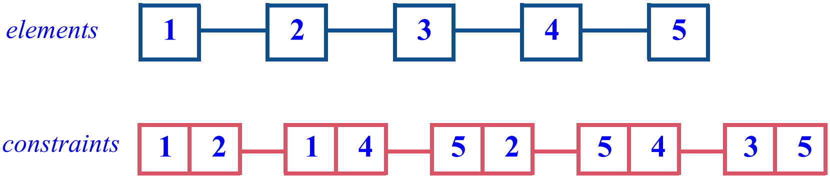
For our first attempt at data structures, it is natural to choose a representation that directly models the problem's input as it comes to us. (One thing you may have already learned about programming is to become suspicious when you hear a solution presented as *natural*. What is natural to me may not be natural to you; and what is natural to you and me may turn out to be silly.)

We most likely get our data as a list of elements and a list of constraints. We can use attributes that directly reflect that structure (declared secret, like all data structures that follow, by appearing, in the class text, in a section of the form **feature** {*NONE*} -- Internal data structures):

←Page 523.

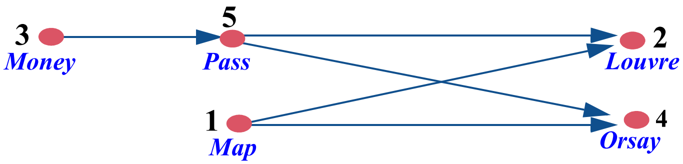
```
elements: LINKED_LIST [G]
constraints: LINKED_LIST [TUPLE [G, G]]
```

In our example the data structures will look like this:



Elements and constraints

assuming for convenience that we have assigned numbers to the example's elements, as follows:



Numbering the elements

(From the figure on page 505.)

Performance analysis of the natural solution

Can we implement what we need — the operations T1, T2, T3 and the procedures *record_element* and *record_constraint* — with this representation, and if so what is the time and space cost?

The two procedures are straightforward. For example *record_constraint* (x, y) will just perform

```
constraints.extend ([x, y])
```

adding the tuple $[x, y]$ at the end of the list of constraints. Similarly, *record_element* (x) will perform *elements*.*extend* (x).

> We must make sure that *constraints* and *elements* are non-void for such instructions; the corresponding **create** instructions may either appear in a *default_create* for the class, or be performed on demand on first need. This will also apply to other data structures introduced below.

← *"Creation procedures", 6.5, page 122.*

We can also use this representation to perform the other operations; let us examine the cost, for n elements and m constraints:

- To find if there is an element without predecessors (T1), we can traverse the list of *constraints* and count predecessors for every element, then traverse the list of elements to find those for which the count is zero; but the first part is an $\mathbf{O}\ (m)$ operation and the second is $\mathbf{O}\ (n)$. If we do this at every step, the total cost is $\mathbf{O}\ (m * n + n^2)$.
- Removing an element (T2) can be as bad as $\mathbf{O}\ (n)$ each time (meaning $\mathbf{O}\ (n^2)$ for the whole process) with a linked list, although we could bring it down to $\mathbf{O}\ (1)$ (total $\mathbf{O}\ (n)$) by tuning the data structure.
- Removing a set of constraints (T3) can again be as bad as $\mathbf{O}\ (m)$ each time, meaning $\mathbf{O}\ (m * n)$ altogether, if all we have to represent constraints is the global list *constraints*, which does not enable us to find all the constraints starting with a given x without traversing the whole structure.

Anything that is $\mathbf{O}\ (m * n)$ or $\mathbf{O}\ (n^2)$ is bad. In particular, we may in a practical application expect most elements to be involved in at least one constraint — often many more in the average, e.g. ten or so in a typical scheduling problem — so that $m > n$, implying that $\mathbf{O}\ (m * n)$ is worse than $\mathbf{O}\ (n^2)$. Anything in n^2, growing like the square of the number of elements, will be out of reach for large practical applications, as the number of elements may be large.

So with this first choice of data structures we do have a solution, but performance-wise it does not scale up to large practical problems.

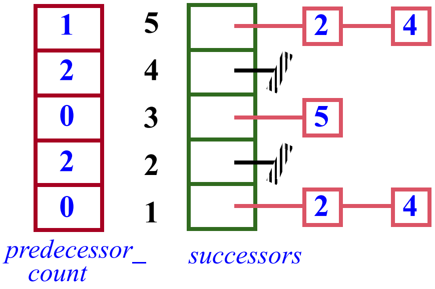
Duplicating the information

Fortunately we can do better than the "natural" solution. The observation is that we do not have to use the data structures as they are given to us. The lists *elements* and *constraints* express the data in a form that directly mirrors how things appear to the external world, for example to the person who inputs a set of tasks and a set of associated constraints. What is clear and "natural" to describe the input to the outside world is not necessarily the best form for an *algorithm* that will process the data for a specific purpose. Rather than following the original form blindly, the algorithm may start with an initialization phase that turns it into the format best suited to that processing.

The following data structures help make the job of topological sort — tasks T1 to T3 — convenient and fast:

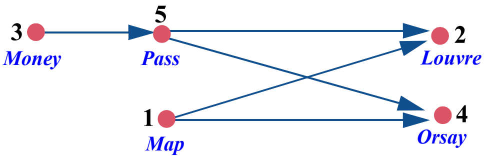
```
successors: ARRAY [LINKED_LIST [INTEGER]]
            -- Indexed by element numbers; for each element x, gives the list of
            -- its successors: the elements y such that there is a constraint [x, y].

predecessor_count: ARRAY [INTEGER]
            -- Indexed by element numbers; for each, says how many
            -- predecessors the element has.
```

Here is how they will initially look for our working example:



Number of predecessors, and lists of successors

This expresses the constraints between elements, repeated on the side figure. For example, the explanation for the entries of indexes 1 and 2 is that task 1 (*Map*) has no predecessor and the successors 2 and 4 (*Louvre* and *Orsay*), and that task 2 (*Louvre*) has two predecessors and no successors.



(*From the figure on page 527.*)

What's interesting in this representation is that the array *predecessor_count* is conceptually redundant: we could always reconstruct the information it provides by exploring the array *successors*, which includes all there is to know about the constraints. But there is nothing wrong with storing information in two (or more) different ways if — as we are going to see — it brings us a significant improvement in computation time.

Such **space-time tradeoffs** are a key ingredient of good algorithm design. Of course the tradeoff has to be acceptable. Here our goal is to have $\mathbf{O}\ (m + n)$ time complexity. In space complexity, *successors* is $\mathbf{O}\ (m + n)$ (one array entry per element, one tuple and reference per constraint); adding the $\mathbf{O}\ (n)$ array *predecessor_count* does not change the picture.

The original data structures, *elements* and *constraints*, already took up $\mathbf{O}\ (m + n)$ space.

Spicing up the class invariant

It is convenient for clarity to add a query

```
count: INTEGER
        -- Number of elements
```

which we can make public. It is also useful, if only for readability, to add the following invariant clauses, the last two expressed informally:

```
elements.count = count
predecessor_count.count = count
successors.count = count

-- For every i in 1..count, predecessor_count [i] is the number of
-- predecessors of i according to the constraints.

-- For every i in 1..count, successors [i] contains all the successors
-- of i as implied by the constraints, or is void if i has no such successors.
```

count represents n in the program.

Numbering the elements

To use an array we need to associate an integer with every element. I sneakily introduced such a numbering a while ago but now it is not just a useful convention; it is required by our choice of data structures.

←"A "natural" choice of data structures", page 527.

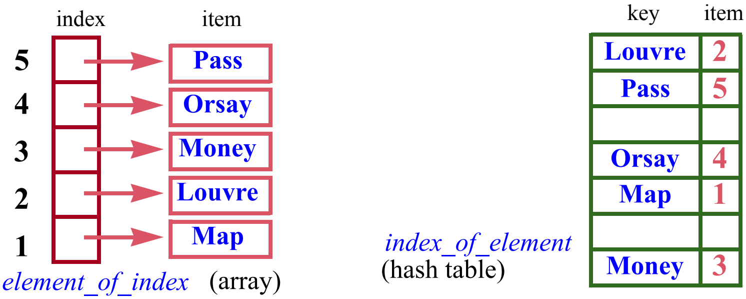
Does this mean that we should renounce the generic parameter *G* of our class *TOPOLOGICAL_SORTER* [*G*] since all manipulations of elements will now use their integer numbers? Absolutely not. It remains necessary, for expressiveness, to produce a mechanism applicable to elements of any type. All we need in practice is a hash table and an array:

```
index_of_element: HASH_TABLE [INTEGER, G]
        -- For every element, gives its index

element_of_index: ARRAY [G]
        -- For every assigned index, gives the associated element
```

The integer *index_of_element* [*e*] will be the number *x* assigned to an element *e*, of type *G*. Then *element_of_index* [*x*] will be *e*.

Both of these use bracket notation: *index_of_element* [*e*] is the item of key *e* in the hash table, and *element_of_index* [*i*] is the item of index *i* in the array.

←"Bracket notation and assigner commands", page 384.



Two-way mapping between elements and numbers

Subject to a proper implementation of hash tables, these structures are both **O** (*n*) in space.

To define a hash table of elements of type *G* requires that *G* conform (through inheritance) to class *HASHABLE*. This was taken care of by declaring the class as *TOPOLOGICAL_SORTER* [*G* –> *HASHABLE*] (using "constrained genericity" as introduced in the next chapter).

← Page 523.

→ "Constrained genericity", page 596.

TOPOLOGICAL_SORTER will not export *index_of_element* and *element_of_index*, since these features are for implementation only; but we must enable clients to find out if a certain element is part of the problem, so we export the following query:

```
has_element (e: G): BOOLEAN
        -- Is e one of the elements to be topologically sorted?
    do
        Result := index_of_element.has (e)
    ensure
        consistent: Result =index_of_element.has (e) and then
            index_of_element [e] >= 1 and then
            index_of_element [e] <= element_of_index.count and then
            element_of_index [index_of_element [e]] = e
    end
```

Make sure you understand the postcondition.

→ *Exercise 15-E.6, page 548.*

Let us see how our new data structures help reach the goal of **O** $(m + n)$ time. Two aspects are now involved: operations T1 to T3, but also initializing the data structures *predecessor_count* and *successors*. Both are equally relevant: it would not help to have **O** $(m + n)$ for the core of the algorithm (the loop iterating operations T1 to T3) if the initialization took — say — **O** $(m * n)$.

It does not seem too hard to initialize the data structures in **O** $(m + n)$: process all constraints in sequence; for every constraint $[x, y]$, increment the y entry in the array *predecessor_count*, and insert y into the list *successors* $[x]$. Both of these operations are **O** (1), so applying them to all constraints is **O** $(m + n)$. We will need to spell out the details, but for now let us indeed assume **O** $(m + n)$ initialization and concentrate on T1 to T3, the core operations of the algorithm.

→ *"Initializations and their time performance", page 538.*

Basic operations

We start with T3: "*Given an element* x*, remove all constraints of the form* $[x, y]$ *for any* y". If we know the number for x, this is straightforward:

L1 We will not need the list of successors of x any more. We could make it void through *successors* $[x]$:= **Void**. In practice, this is not necessary, as the algorithm will never visit the entry x of *successors* any more. But even if we had to perform this operation it would be **O** (1) — meaning **O** (n) globally if we apply it to all elements. Good!

L2 We must also update any relevant entry in the array *predecessor_count*. The effect on this array of "removing all constraints $[x, y]$" for given x means that we must decrease *predecessor_count* $[y]$ by 1 for every successor y of x. So it suffices to traverse the list *successors* $[i]$ (before you set it to void, of course, if you want to do that), and for each element encountered decrease the corresponding entry in *predecessor_count*. This is a straightforward loop, whose code appears below. This process will be done *at most once*, in the entire processing, for each constraint in the system. So it is **O** (m). Good again!

→ *Innermost loop on page 536.*

T3, then, is **O** $(m + n)$ at worst.

All the processing just described is there to maintain the invariant clauses expressing that the array *predecessor_count* and the array of lists *successors* faithfully reflect the structure of the remaining constraint relation.

← As introduced on page 530.

T2 is "*given an element x, remove it from the set of elements*". In fact, with our new data structures, we do not really need to do anything here. The information that really matters affects the *constraints* starting with *x*, and we have just taken care of these. Definitely good.

There remains T1, "*find an element without predecessors — or report there isn't any*". It suffices to traverse the array *predecessor_count* and look for zero values. But this is **O** (n) meaning, overall, **O** (n^2). *Not* good!

We are still missing one — our last — data structure.

The candidates

We will not avoid one **O** (n) traversal of the array *predecessor_count* upon initialization — the **from** clause of our main loop — to find out the initial "candidates" for immediate output: elements without predecessors in the original relation. Unless *every* element is involved in some cycle, a rather inauspicious initial situation for an attempt at even partial topological sorting, we will find one or more *x* for which *predecessor_count* $[x]$ is initially 0. This requires **O** (n), as noted, but paying **O** (n) once is not a problem.

→ In procedure find_initial_candidates, page 537.

After that, we will not ever need to traverse the array *predecessor_count* to look for new candidates. It suffices to notice that the operation labeled L2 on the previous page, which decrements one or more entries of the array *predecessor_count*, is the *only* one that can make an entry of the array zero if it wasn't zero initially. This means we can just extend the operation to make it watch for entries that become zero. Assuming it was written

```
        -- Decrease the y entry of predecessor_count by one:
predecessor_count [y] := predecessor_count [y] – 1
```

we replace it by

```
        -- Decrease the y entry of predecessor_count by one
        -- and check if this makes y an element without predecessors:
predecessor_count [y] := predecessor_count [y] – 1                        [3]
if predecessor_count [y] = 0 then
        "Record that y is now without predecessors"
end
```

To “Record that an element is without predecessors” it suffices to add it to a structure *candidates* which will, after initialization and after every iteration of the loop, contain all not yet processed elements that have no predecessor. What concrete data structure should we use for *candidates*? For the topological sort algorithm the precise choice does not matter as long as the structure supports the following five features:

count is there for completeness, but we will not actually need it.

```
feature -- Access

    item: G
            -- An element previously inserted.
        require
            not_empty: not is_empty

feature -- Measurement

    count: INTEGER
            -- Number of elements.
        ensure
            non_negative: Result >= 0

feature -- Status report

    is_empty: BOOLEAN
            -- Is there no element?
        ensure
            definition: Result = (count = 0)

feature -- Element change

    put (x: G)
            -- Insert x.
        ensure
            one_more: count = old count + 1

    remove: G
            -- Remove the element given by item.
        require
            not_empty: not is_empty
        ensure
            one_fewer: count = old count – 1
```

The general name for such a structure is **dispenser**, by analogy with a machine into which you may deposit elements (*put*) and also, by pressing a button, getting a previously deposited element (*item* and *remove*), assuming there is still at least one (**not** *is_empty*):

← *"Dispensers", 13.10, page 418.*

A dispenser

As you remember, the basic idea is that you do not choose the element to get and remove: the dispenser chooses for you. Stacks, with a LIFO policy, and queues, with a FIFO policy, are dispensers; so are priority queues.

For topological sort, any dispenser will do the job. Choosing a particular kind affects the actual order — among those compatible with the constraining relation — in which the algorithm outputs elements. This is the lever that you can apply to select a specific policy, for example to ensure that the result will optimize a certain criterion. It is also the reason for describing topological sorting as an algorithm *family* rather than a single algorithm.

→ *As explored in exercise "Parameterizing topological sort", 15-E.9, page 548*

We may declare the candidate dispenser as

```
candidates: PRIORITY_QUEUE [INTEGER]
          -- Elements without predecessors, ready to be released

-- Additional clause for the invariant:
     -- For every item x of candidates, predecessor_count [x] = 0
```

An implementation by a *STACK* or *QUEUE* would also do; a "priority queue" is a more general kind of dispenser where every element may be given a priority, with the rule that *item* yields (and *remove* takes away) the element with the highest priority. *STACK* and *QUEUE* are the special cases of priorities set as an increasing and decreasing function of the order of insertions. A general *PRIORITY_QUEUE* allows you, by playing with the priorities, to enforce any selection policy of interest.

The loop, final form

We may now write the main loop of the topological sort algorithm — the body of the procedure *process* — with all its details. The pseudocode instructions of the previous version have been left as comments (in red) for comparison. The routine must declare local variables *x* and *y* of type *INTEGER* and *x_successors* of type *LIST* [*INTEGER*], recording the successors of a particular element. We also add an integer variable *processed_count* — used next — to keep track of how many elements we have processed.

← The basic form appeared on page 526. For the context, including procedure process, see page 523.

```
from
    create sorted.make
    find_initial_candidates -- See next
invariant
    -- "The data structures represent a subset of the original elements,
    -- and the corresponding subset of the original relation"
until
    candidates.is_empty
loop
                        -- "Let x be a member of elements with no
                        -- predecessor for constraints"
    x := candidates.item ; candidates.remove

    sorted.extend (element_of_index [x])

                        -- "Remove x from elements and
                        -- all pairs starting with x from constraints"
    x_successors := successors [x]  -- A list
    from x_successors.start until x_successors.after loop
        y := x_successors.item

                            -- Next few lines are from [3], page 533:
        predecessor_count [y] := predecessor_count [y] – 1
        if predecessor_count [y] = 0 then
                            -- "Record that y is now without predecessors"
            candidates.put (y)
        end

        x_successors.forth
    end
    processed_count := processed_count + 1
variant
    count – processed_count
end
report_cycles           -- See next
done := True
```

This algorithm assumes that the arrays *predecessor_count* and *successors* have been properly set up, as must be the case before any call to *process*. The details of the initializations are coming next.

→ "Initializations and their time performance", page 538 below.

Operations on *candidates* are highlighted to emphasize how critical this structure has now become to the algorithm.

The procedure *find_initial_candidates* must set up the *candidates* dispenser with the elements initially without predecessors. It is straightforward:

```
find_initial_candidates
        -- Insert into candidates any elements without predecessors.
    local
        x: INTEGER
    do
        if candidates = Void then create candidates end
        from x := 1 until x > count loop
            if predecessor_count [x] = 0 then
                candidates.put (x)
            end
            x := x + 1
        end
    end
```

This is the **O** (n) traversal that without *candidates* we would have had to perform at each step of the loop; now we just do it once at the beginning.

← As noted at the end of "Basic operations", page 533.

It is not an error for the procedure to find *no* elements satisfying *predecessor_count* [x] = 0; this case will simply result in an empty *candidates* structure, causing the loop to terminate immediately, as every element is involved in a cycle.

Procedure *process* must do one more thing after the loop: set up the information enabling a client to find out about any cycles in the input. This is the task of procedure *report_cycles*. To implement it, we note that the loop terminates when there are no more elements in *candidates*; if the original relation was acyclic we will have processed all elements, so we use *processed_count* to find out whether there are any left:

```
report_cycles
        -- Make information about cycles available to clients.
    do
        if processed_count < count then
                        -- There was a cycle in the original relation!
            cycle_found := True
            create {LINKED_LIST [G]} cyclists.make
            from x := 1 until x > count loop
                if predecessor_count [x] /= 0 then
                        -- x was involved in a cycle
                cyclists.extend (element_of_index [x])
                x := x + 1
            end
        end
    end
```

Initializations and their time performance

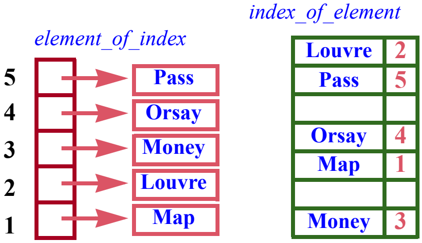
We now have an efficient — **O** (*m* + *n*) — implementation of the core of the topological sort loop, thanks to three data structures chosen directly to fit its needs: the arrays *predecessor_count* and *successors*, and the dispenser *candidates*. To complete the job we must spell out their initialization, making sure they do not exceed the performance constraints.

The initialization will have to perform:

- *record_element* (*e*) for every element: *n* times altogether.
- *record_constraint* (*e*, *f*) for every constraint: *m* times altogether.

The job of *record_element* (*e*) is to assign a number to *e*, so that the rest of the processing can deal with integers, rather than actual elements of *G*.

This is done by filling in twin entries in the array *element_of_index* and the hash table *index_of_element*:



(*From the figure on page 531.*)

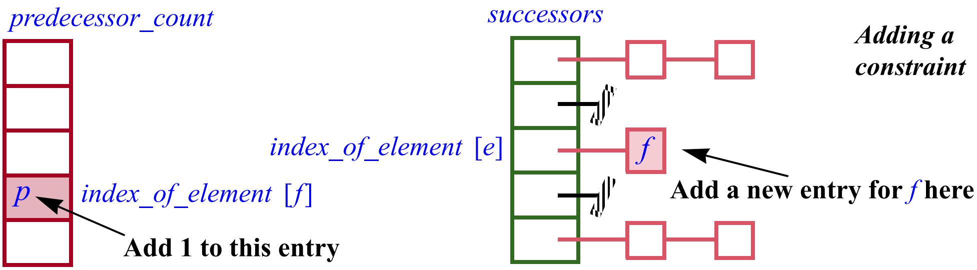
```
record_element (e: G)
        -- Add e to the set of elements, unless already present.
    require
        not_sorted: not done
    do
        if not has_element (e) then
            count := count + 1

            index_of_element.extend (count, e)
            element_of_index.force (e, count)
        -- extend and force expand the structures if necessary; this means
        -- we do not need to know the number of elements in advance.
        end
    ensure
        inserted: has_element (e)
        one_more: not (old has_element (e)) implies (count = old count + 1)
    end
```

> The initial test ensures that the procedure ignores a second attempt to insert a given element. This policy allows *record_constraint* (*e*, *f*) to start by calling *record_element* on both *e* and *f* just to make sure the elements are properly inserted. An exercise asks you for a way to avoid the duplication of work between *has_element* and *extend*.

→ *Exercise 15-E.7, page 548.*

With appropriate implementations of *extend* and *force*, the code of *record_element* is **O** (1); executed for all elements, it will contribute **O** (*n*) to the algorithm. This is in line with our requirements.

The remaining initialization mechanism is the procedure for entering constraints. A call to *record_constraint* (*e*, *f*) must increase by 1 the number of predecessors of *f* in the array *predecessor_count*, and add *f* to the list of successors of *e*. That list is one of the entries of the array *successors*:



Adding a constraint

The routine will read

```
record_constraint (e, f : G)
        -- Add the constraint [e, f ].
    require
        not_sorted: not done
        exist: e /= Void and f /= Void
    local
        x, y: INTEGER
    do
            -- Ensure e and f are inserted (no effect if they already were):
        record_element (e); record_element (f)

        x := index_of_element [e]
        y := index_of_element  [f ]
        predecessor_count  [y] := predecessor_count  [ y] + 1
        add_successor (x, y)
    ensure
        both_there: has_element (e) and has_element (f)
    end
```

with an auxiliary procedure (which does not need to be exported):

```
add_successor (x, y: INTEGER)
        -- Record y as successor of x.
    require
        1 <= x ; x <= count
        1 <= y ; y <= count
    local
        x_successors: LINKED_LIST [INTEGER]
    do
        x _successors := successors [x]

            -- The successor list for x may not have been created yet:
        if x_successors = Void then
            create x_successors.make
            successors [x] := x_successors
        end

        x_successors.extend (y)

    end
```

As suggested earlier, *record_constraint* starts by calling *record_element* on the constraint's two elements; because of the way we have designed *record_element*, this has no effect if they were already there. This policy makes it possible for a client application to start from just a list of constraints, never having to call *record_elements* explicitly.

> We cannot, however, assume this will always be the case and remove *record_element* from the public interface of the class. An instance of the problem may, as noted, include elements that are not involved in any constraint but should still be listed as part of the output. In such a setup, the input must include, separate from the list of constraints, a list of elements.

On the subject of duplication, the procedure *record_constraint* does not attempt to determine if a constraint has already been entered. Indeed, as you are invited to check, our topological sort algorithm will work as expected if a constraint appears twice. This may well happen with manually entered data and the algorithm does not consider it an error; there is nothing contradictory in saying twice "*e* must come before *f*". To apply a different policy is the responsibility of the client application, as part of input validation.

Now what about efficiency? The code for each of the two auxiliary procedures is **O** (1): one array access plus, in the second case, insertion at the end of a list (with a good implementation ensuring that the list cursor stays at the end) and, once for each applicable element, an object creation. Then *record_constraint* as a whole is **O** (1); as it is executed once for each constraint, its contribution to the algorithm is **O** (m). We have achieved our goal of **O** ($m + n$) time for the initialization as well as the main part of the algorithm.

Putting everything together

You have now seen all the program elements needed to implement topological sort. A class built directly from this discussion is available in EiffelBase and used in Traffic, but I suggest that independently of this existing implementation you check your understanding of the concepts by writing yourself an implementation that brings them all together:

Programming time!
Usable implementation of topological sort

From the elements of this chapter, write a class *TOPOLOGICAL_SORTER* providing a general, usable implementation of topological sort.

→ Exercises 15-E.8, page 548 and 15-E.9, page 548.

Make sure to engineer the solution properly by providing not just the algorithm but also the initialization procedures (*record_element*, *record_constraint*).

To test your solution, you will find at the page for this course a file listing a few hundred example constraints, and all its possible topological sorts.

15.5 LESSONS

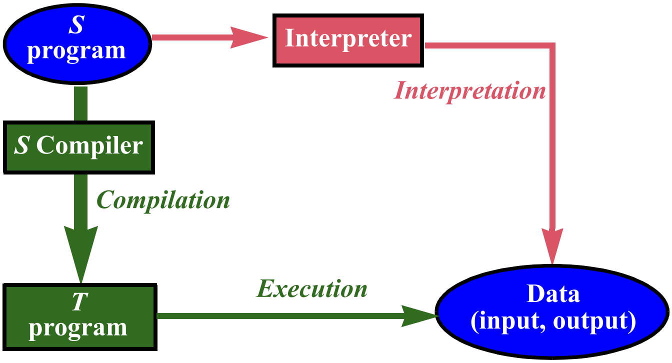
The topological sort algorithm has important consequences to teach us for both algorithm design and software engineering.

Interpretation vs compilation

We have seen the two solution styles for executing programs written in some source programming language *S*:

← *"Compilation vs interpretation", 12.2, page 330.*

- Interpretation; write a program, called an **interpreter**, that can directly execute an arbitrary *S* program applied to an arbitrary input.
- Compilation: write a program, called a **compiler**, that transforms any *S* program into a program with equivalent semantics expressed in a target language *T*. If *T* is machine language for the desired platform, the result can be directly executed; alternatively, *T* can be further compiled or interpreted.



Interpretation and compilation of programs

(Simplified version of figure on page 331.)

Practical language implementations, as noted, often combine the two approaches.

← *"Combining compilation and interpretation", page 332.*

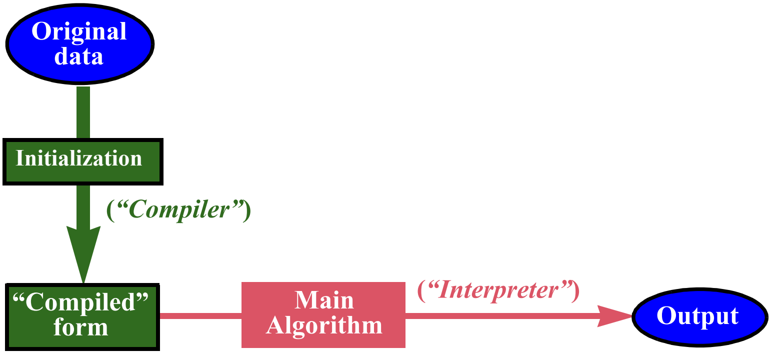
These concepts generalize to many application domains other than the execution of programs in a high-level language. To perform a certain processing on a certain input, we may use data structures that directly mirror the input; or we may proceed in two steps:

- *Compilation*: transform the data into a form more suitable for the algorithm's needs.
- *Interpretation*: apply the needed operations to the resulting structure.

This technique (which, as for language processing, may in the general case involve several iterations of the process) is exactly what we have applied for topological sort. We first looked at an interpretive solution using the seemingly natural data structures, directly deduced from the statement of the problem; but they turned out to yield bad performance. "Compiling" them into a representation tailored to our goals led to a solution with excellent performance.

← *"A "natural" choice of data structures", page 527.*

In such a two-step solution, the "compilation" step, which initializes the data structures, may be as delicate to devise as the actual processing based on its results, and it may account for as much time, sometimes more. That is fine as long as the overall performance meets your goals — but of course you must not jump to conclusions about performance until you have taken into consideration the initialization as well as the later processing.



"Interpretation" and "compilation" of data

The approach can be summed up as a *heuristics* — a general strategy, similar to a "design pattern" but of a more abstract nature, that is known to help devise good solutions in suitable cases:

→ *"About design patterns", page 678.*

Touch of Heuristics: Compile the data first!

Good algorithms are often obtained through a two-step strategy where:

- The first step turns the input, from its given form, into internal data structures carefully devised to suit the algorithms' goals.
- The second step processes the resulting form to attain these goals.

We will see another application of this idea in the discussion of architectures for event-driven design.

→ *"Invest then enjoy", page 694.*

Time-space tradeoffs

Closely tied to the "Compile the data first!" heuristics is the observation that the ideal data structure — the one best helping the second step — is often not the most economical representation possible for the underlying information. In the topological sort solution, information about constraints may end up in *three* different parts of the data structure: a constraint [*x*, *y*] causes *y* to appear in the list of *x*'s successors in the array *successors*; it adds one to *predecessor_count* [*y*]; and the absence of any such constraint for a given *y* leads to inserting *y* into *candidates*. Such replication sacrifices some space to ensure a considerable gain in execution time. Tradeoffs of this kind, in this direction or the reverse one, are among the keys to efficient algorithm design.

← For the terminology see "Information and data", page 8.

Algorithms vs systems and components

It is possible to give a description of topological sort that ignores many of the aspects studied in this chapter, concentrating only on the final algorithm. For a usable solution, however, one must take into account the practical needs of applications. The object-oriented approach allows us to meet this goal: instead of writing a "topological sort program" we have devised a **class**, *TOPOLOGICAL_SORTER*. An instance of this class describes an instance of the topological sort problem, equipped with not only the algorithm (procedure *process*) but also with all the apparatus enabling clients to:

- Set up the problem instance, by recording elements and constraints in a convenient way.
- Apply *process* to produce a topological sort of the applicable elements, satisfying the constraints.
- Query the resulting state, to discover whether any cycles were found, and if so what elements they involve.

The difference between this approach and a mere algorithm is part of the difference between **software engineering** and mere programming. In software engineering it is not enough to devise clever algorithmic solutions and the associated data structures; the goal is to provide *solutions* that can be integrated into successful systems.

→ Chapter 19 describe the non-programming aspects of software engineering.

In addition we should make these solutions **reusable**, so that they are not just design patterns, which programmers can integrate into their systems by buying and reading books (especially excellent books such as the present one), but *components* that can be made available, once and for all, for reuse directly off the shelf.

You will also have noted how, in this process, the **contracts** enable us at every step to know exactly what we are doing — what we expect, what we guarantee and what we maintain.

15.6 KEY CONCEPTS LEARNED IN THIS CHAPTER

- A topological sort is an enumeration of a set of elements compatible with a set of ordering constraints on these elements.

- The problem has a simple mathematical description: given a (strict) order relation, find a total order that is a subset of it.

- In practice the relation is usually not an order relation but at best acyclic. Taking its transitive closure gives an order relation.

- A realistic, well-engineered software solution must accept possibly erroneous input in which the relation has cycles. It should then produce a topological sort of the acyclic part, and report remaining cycles.

- Such a solution must provide not just the topological sort algorithm but also mechanisms to build a problem instance by entering individual elements and constraints.

- With n elements and m constraints, it is possible to perform topological sorting in $\mathrm{O}\,(m + n)$ time and space.

- The key to the efficiency of the algorithm is that it works from data structures specifically adapted to the problem: two arrays giving, for each element, the list of its successors and the number of its predecessors; and a dispenser (stack, list or priority queue) containing the set of elements without predecessors.

- As this example illustrates, good algorithmic solutions are often obtained by first “compiling” the problem’s data into a specially designed data structure, which can then be “interpreted” efficiently.

New vocabulary

Acyclic	**Antisymmetric**	**Binary relation**
Cycle	**Irreflexive**	**Order relation**
Partial order	**Relation**	**Strict order**
Topological sort	**Total order**	**Transitive closure**

15.7 APPENDIX: TERMINOLOGY NOTE ON ORDER RELATIONS

To discuss topological sort it is convenient — as this chapter has shown — to use *strict* order relations, as in "*strictly* less than"; the "<" relation on numbers is an example of strict order. For other problems it may be more useful to deal with the non-strict versions, such as "≤" on numbers. The two are closely related: $x \leq y$ holds if and only if $x < y$ **or** $x = y$. The common convention is for "order relation" to mean the nonstrict version. In this chapter, since we have used only strict order relations, the word "strict" has usually been omitted, so that "order" means "strict order".

For a strict order relation (irreflexive and transitive), some of the literature uses the term *quasi-order*. Of course one may pick any name for a notion as long as one provides a precise definition, but this particular name is unfortunate since there is nothing "quasi" about such orders; if anything they are "more" ordered than nonstrict variants — those usually called "order relations" — since they do not hold between an element and itself (irreflexivity). To make things worse, other authors use "quasi-order" for relations that are transitive and *reflexive* (rather than irreflexive). So it is better to stay away from this term and instead qualify order relations as "strict" when needed.

The next issue is whether the relation is total or not. Totality means that for any two non-equal elements x and y one of $x < y$ and $y < x$ will hold. An order relation that satisfies this property is a *total order*. One that does not satisfy the property — meaning that there is at least one pair of distinct elements for which neither $[x, y]$ nor $[y, x]$ is in the relation — should be called a *partial* order. But that is not what "partial order relation" means in most of the literature: it means an order relation that we do not know to be total. In other words, it is a *possibly partial* order relation. That is confusing, since now a total order relation is also partial! It is better to write, as in this chapter:

- *Total order* for an order relation that is known to be total.
- *Partial order* for an order relation that is known to be non-total.
- If we do not know, or want to include both cases, just *order*. In case of possible ambiguity, use "possibly partial order".

15-E EXERCISES

15-E.1 Vocabulary

Give a precise definition of each of the terms in the vocabulary list on the preceding page.

15-E.2 Irreflexivity and asymmetric

Order relations were defined as irreflexive and transitive, and proved asymmetric as a consequence. Prove that it is equivalent to define them as asymmetric and transitive, with irreflexivity a consequence.

← *Page 511*.

15-E.3 Total order and enumeration

Prove that if a relation r is a total strict order on a finite set, there exists a single enumeration of the elements such that, for any elements x and y, x appears before y in the enumeration if and only if the pair $[x, y]$ is in r.

15-E.4 Strict vs. nonstrict orders

The discussion in this chapter has relied on *strict* order relations (partial or total), such as "<" on integers, "less than". It is also possible to use *nonstrict* order relations, such as "≤", "less than or equal to". The definition of a partial strict order — which we will call "<" although it does not have to be the usual relation on numbers — was that it must be:

← *"Binary relations", page 509.*

O1 **Irreflexive**: $x < x$ holds for no x.

O2 **Transitive**: whenever $x < y$ and $y < z$ hold, so does $x < z$.

That the relation is also

O3 **Asymmetric**: $x < y$ and $y < x$ may not both hold.

is a consequence of the previous two properties.

For a partial *nonstrict* order relation "≤", there are three independent conditions:

N1 **Reflexive**: $x \leq x$ for any x.

N2 **Transitive**: whenever $x \leq y$ and $y \leq z$ hold, so does $x \leq z$.

N3 **Antisymmetric**: whenever $x \leq y$ and $y \leq x$ both hold, then $x = y$.

For any partial strict order relation "<" there is an associated relation "≤", defined by

$$x \leq y \text{ if and only if: } x < y \textbf{ or } x = y \qquad \textbf{[4]}$$

Conversely, given a partial nonstrict order relation "≤", we may define an associated relation "<" by

$$x < y \text{ if and only if: } x \leq y \textbf{ and } x \neq y \qquad \textbf{[5]}$$

This exercise explores the relationship between these associated strict "<" and nonstrict "≤" variants.

1 Prove that if "<" is a partial strict order relation, then "≤", as defined by [4], is a partial nonstrict order relation.

2 Prove that if "≤" is a partial nonstrict order relation, then "<", as defined by [5], is a partial strict order relation.

3 In the strict case, the definition imposes only two conditions: irreflexivity and transitivity; condition O3, asymmetry, is a consequence. In the nonstrict case, there are three conditions. Prove that antisymmetry, N3 does not necessarily follow from the other two, reflexivity and transitivity. (In other words, find an example that satisfies N1 and N2 but not N3.)

4 Prove that replacing "reflexive" by "irreflexive" in the definition of a nonstrict order yields the definition of a strict order.

5 Does replacing "irreflexive" by "reflexive" in the definition of a strict order yield the definition of a nonstrict order?

15-E.5 Acyclic and total order relations

Prove the Topological Sort theorem on the basis of the second theorem on acyclic and order relations.

← Page 516; page 513.

15-E.6 An interesting postcondition

Explain the postcondition of the function *has_element*.

← Page 532.

15-E.7 Optimizing hash table usage

The algorithm for procedure *record_element* tests whether an element *e* is already present in the hash table *index_of_element*, and inserts it only if not. This causes two search operations, one for *has* (called by *has_element*) and one for *extend*. Examining the contract form of *HASH_TABLE* to find the appropriate features, rewrite the procedure to avoid this small inefficiency.

← Page 540.

15-E.8 Programming topological sort

Implement the class *TOPOLOGICAL_SORTER* according to the discussion of this chapter.

← See "Programming time! Usable implementation of topological sort", page 541.

15-E.9 Parameterizing topological sort

(This exercise assumes you have done the preceding one.) Extend the implementation of topological sort to enable clients to select a specific policy for choosing between competing "candidates" ready for output.

→ As discussed in "The candidates", page 533.

PART IV:

Object-Oriented Techniques

In this fourth part we move to the forefront of modern programming technology and see how we can benefit from the full power of object-oriented ideas.

The first of its three chapters describes the many facets of inheritance, including polymorphism, dynamic binding, multiple inheritance and the notion of constrained genericity. The second one explores newer developments that add considerable expressive power to the object-oriented framework: agents (also known as closures), with a detour through the theory of lambda calculus. The last chapter of this part presents a flexible type of software architecture, complementing our earlier control structures: event-driven design.

16 Inheritance

The world (are you ready, this early in the chapter, for deep revelations about life?) is a mess. Nature perhaps abhors a mess, perhaps not, I am not sure, it depends on whom you read — Plato, Aristotle, Kant — but science definitely does. Sane reasoning about the world demands order.

Order is what science provides: an idealized, organized version of reality. We'll leave it to our philosopher friends to debate whether the order was really present in the first place, behind a disorderly appearance, and all science has to do is uncover it; or if the world *really* is a mess and science stencils an artificial order onto the natural mishmash. What matters, to get us started on understanding inheritance, is that science, and with it engineering, seek systematic, well-structured descriptions. One of the principal tools in this quest is hierarchical classification, also known as taxonomy.

To become sciences, botany and zoology needed Linné, in the 18th century, to devise an effective classification of living beings. Biological taxonomy tells us that the common dolphin species, which it calls *Delphinius delphis*, is included in the genus *Delphinius*, itself part — I am skipping levels — of the order of Cetaceans, which belongs to the class of Mammals, included in the superclass of Tetrapodes, attached to the sub-phylum of Vertebrates, member beyond any doubt of the kingdom of Animals. The objects in this case are natural; the order is artificial.

For the full line see for example the Wikispecies entry: species.wikimedia.org/wiki/Delphinus_delphis.

> It is indeed one of many possible human choices; Aristotle, for example, did not associate dolphins with land mammals, although he recognized they were not fish as others had previously classified them.

In mathematics, the objects themselves — numbers, functions, sequences… — are artificial, a creation of human minds. Évariste Galois in the early 19th century, then Georg Cantor and others came up with abstract mathematical structures to group such mathematical objects — however different they may superficially look — into categories, themselves organized in hierarchical classifications. With one operation that is associative and has an identity element (think of concatenation for strings, with the empty string as identity, or addition for non-negative integers with 0) you get a *monoid* structure; add the notion of inverse, and you have a *group* (think of the set of all integers, with *–x* as the inverse of *x*); add a second operation with adequate properties (on integers, multiplication) and you get a *ring*; bring in some more properties and you get a *field* (think of real numbers).

Like mathematicians, we programmers deal with abstract objects — figments of our imagination — and cannot blame their disorder on anyone but ourselves. Like everyone, we need to organize these disorders into a semblance of order. Perhaps we need this *more* than anyone else, because we are arguably the world champions in entropy creation, apt through our programs to create bigger messes than was ever thought possible. "*To err is human, but to mess up for good takes a computer*", the saying goes, to which we may add "*or a computer programmer*".

To fight this we can, like other sciences, use taxonomy. We organize our objects into categories, and examine the hierarchical relations between those categories. As a dolphin is a mammal and a mammal a vertebrate, as a field is a ring, a taxi as modeled in our programs is a vehicle and a vehicle is a moving city object. A pedestrian is a moving city object too, but not a vehicle. Inheritance will enable us to reason about such "*is-a*" relationships and use the resulting taxonomies to structure our software.

We have encountered inheritance and the associated **inherit** keyword before — actually in our very first example — as a way to let a class benefit from the work done for previous classes. This was only one of the aspects of inheritance; it applies to classes viewed as *modules* (collections of useful features). Inheritance becomes even more interesting — with new techniques such as polymorphism and dynamic binding — in its application to the other role of classes: as *types*, each describing a collection of run-time objects, such as metro lines or taxis.

← PREVIEW inheriting from TOURISM: page 16.

16.1 Taxis are vehicles

Yes, taxis. We have done our plebeian bit enough in previous chapters, taking the metro with everyone else; now we move up one rung, economically if not environmentally, and commandeer our own vehicles.

Inheriting features

Traffic's class *TAXI* provides, as you can check, such features as

```
take (from_location,to_location: LOCATION)
          -- Bring passengers from from_location to to_location
```

The actual class name is TRAFFIC_TAXI; as usual you should prefix the names of all Traffic classes in this chapter by TRAFFIC_.

and *office*, representing the taxi dispatching office.

When reading the class text you will only see a few other features. And yet taxis have more properties. For example:

- A taxi has passengers (otherwise the comment of *take* would make no sense: what passengers are being taken from one place to another?); the class must include a command to load passengers and a query to find out the current number of passengers.
- At any time a taxi has a current position.

Where are the corresponding features? You can find the answer by noticing the beginning of the class declaration:

```
note
    …
class
    TAXI
inherit
    VEHICLE
feature
    … Rest of class …
```

TAXI inherits from *VEHICLE* and indeed if you look up class *VEHICLE* you will find the commands *load*, to load passengers into a vehicle, as well as *unload* and a query *count* giving the current number of passengers. Now look up *VEHICLE* itself and you will see

```
note
    …
deferred class
    VEHICLE
inherit
    MOVING
feature
    … Rest of class …
```

where the class *MOVING*, describing any moving objects that can be tracked in Traffic, has queries such as *position*.

VEHICLE, and *MOVING* as well, are introduced not just by the usual **class** but by **deferred class**; we will soon study in more detail the concept of deferred class, used to specify that a class does not fully describe all its features, leaving the implementation of some or all of them to classes that inherit from it.

Looking at these three classes as describing types of run-time objects — taxis, vehicles, anything that moves — we see what the inheritance clauses tell us: that in Traffic any taxi can be handled as a vehicle, and any vehicle as a moving object. In particular, all the features of class *MOVING* are applicable to targets of types *VEHICLE* and *TAXI*, and all the features of *VEHICLE* to targets of type *TAXI*.

Inheritance terms

As usual, precise terminology helps:

Definitions: heir, parent, (proper) descendant and ancestor

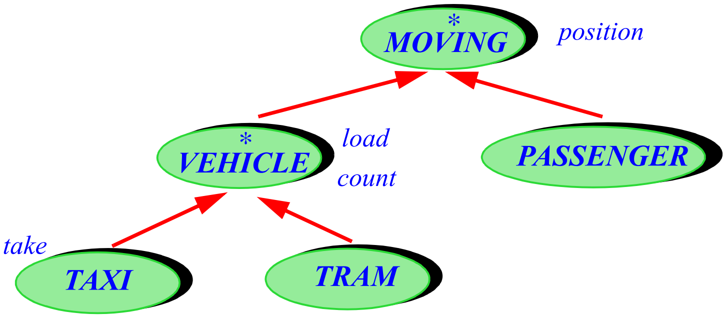
If *B* inherits from *A* (in Eiffel by listing *A* in its **inherit** clause), *B* is an **heir** of *A* and *A* a **parent** of *B*.

The **descendants** of a class are the class itself and (recursively) the descendants of its heirs; **proper** descendants exclude the class itself. "**Ancestor**" and "**proper ancestor**" are the reverse notions.

The definition of descendant is recursive, but by now this should cause no mystery for you; another, more informal way to express it is that descendants of *A* include *A* itself, its heirs, their own heirs and so on.

In the literature you may encounter the term "subclass", sometimes meaning heir and sometimes proper descendant, as well as "superclass".

In our example as illustrated below: the descendants of *MOVING* are all the classes shown; its proper descendants are all these classes except *MOVING* itself; the proper ancestors of *TAXI* are *VEHICLE* and *MOVING*.

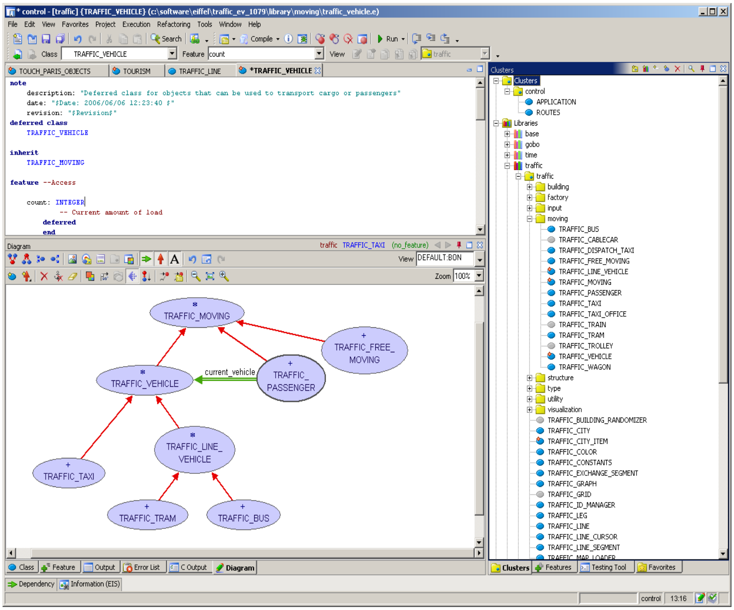


An inheritance hierarchy

Next to each class, the figure shows a feature or two that it introduces. Note the convention for representing inheritance on such class diagrams: a single arrow, red in our figures; remember that the other relation between classes, "client", is represented by double arrows. The inheritance arrow always goes from heir to parent, not the other way around, reflecting an important property: the designer of a class knows about its parents (and hence its ancestors), but should not have to know about its proper descendants.

You do not have to produce class structure diagrams such as the above manually; if the classes exist and have been compiled the Diagram Tool of EiffelStudio will produce them for you. Just click the Diagram tab in the bottom subwindow and drop classes there:

← *"Text, program and design editors", 12.5, page 342.*



* control - [traffic] {TRAFFIC_VEHICLE} (c:\software\eiffel\traffic_ev_1079\library\moving\traffic_vehicle.e)

Class TRAFFIC_VEHICLE Feature count

```
note
    description: "Deferred class for objects that can be used to transport cargo or passengers"
    date: "$Date: 2006/06/06 12:23:40 $"
    revision: "$Revision$"
deferred class
    TRAFFIC_VEHICLE

inherit
    TRAFFIC_MOVING

feature --Access

    count: INTEGER
            -- Current amount of load
        deferred
        end
```

An inheritance hierarchy in the Diagram Tool

As the diagram shows, the previous figures omitted some classes for simplicity; in particular TRAM inherits from VEHICLE not directly but through the intermediate class LINE_VEHICLE, of which another heir is BUS. Also note the client link fromPASSENGER to VEHICLE.

If the class does *not* exist yet and you are designing the class structure. you can also use the Diagram Tool for that purpose, to create classes and describe their client and inheritance links graphically; in that mode the texts will be generated from the diagrams rather than the other way around.

Features from a higher authority

With examples such as the *MOVING* hierarchy, we can appreciate the novelty introduced by inheritance: the "*features of a class*" no longer mean just the features declared in the class itself, but also those inherited from a parent. So with *m*: *MOVING*, *v*: *VEHICLE* and *t*: *TAXI* you may write, along with

```
v.load (…)
t.take (…)
v.count            -- An expression
```

other feature calls such as

```
t.count
t.load
t.position

v.position
```

which rely on features inherited from a parent or, more generally, a proper ancestor. We may distinguish between "immediate" and "inherited" features:

Definitions: features of a class, immediate, inherited, introduce

A "**feature of a class**" is one of:

- An **inherited** feature if it is a feature of one of the parents of the class.
- An **immediate** feature if it is declared in the class, and not inherited. In this case the class is said to **introduce** the feature.

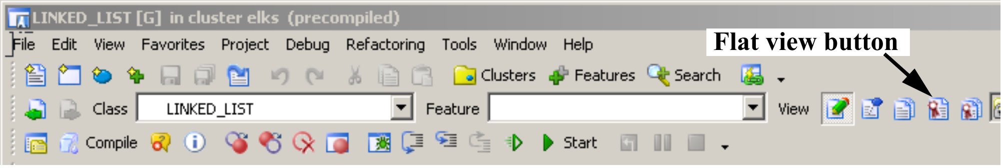
Note that the definition involves recursion.

From now on, then, there is more to a class *A* than what you see in its class text: "features of *A*" means not only the features declared there, but also those it gets from its parents, if any.

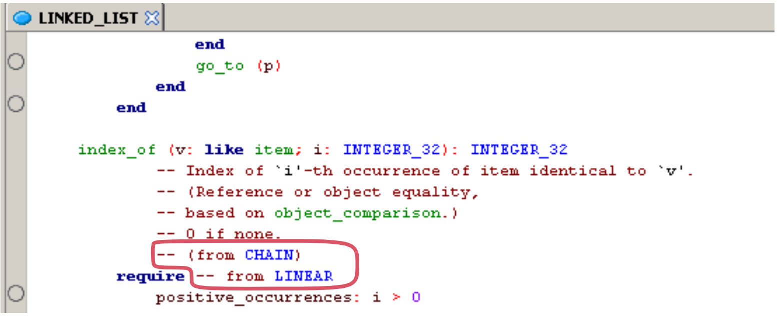
The flat view

How then can you get the full picture? It is called the **flat view** of a class: an artificially reconstructed version that has all the features, immediate and inherited. It is not something you write but just a view, like the *contract view* which, as we saw, gives you the implementation-free version of a class. EiffelStudio will produce it for you; just bring up any class and click the "Flat view" icon in the top right (let the tooltip help you find it):

← *"What characterizes a metro line", page 53.*


LINKED_LIST [G] in cluster elks (precompiled)
File Edit View Favorites Project Debug Refactoring Tools Window Help
Clusters Features Search
Class LINKED_LIST Feature View
Compile Start
Flat view button

The result looks like a normal class text (with some new notations that will become clear as we go along); note a new kind of comment, here for a feature of *LINKED_LIST*:



```
                    end
                    go_to (p)
                end
            end

        index_of (v: like item; i: INTEGER_32): INTEGER_32
                -- Index of `i'-th occurrence of item identical to `v'.
                -- (Reference or object equality,
                -- based on object_comparison.)
                -- 0 if none.
                -- (from CHAIN)
            require -- from LINEAR
                positive_occurrences: i > 0
```

Flat view

These are not in the original class text but added by EiffelStudio when it produces the flat view; they indicate that the feature is inherited, coming here from the proper ancestor *CHAIN*, and that its precondition was defined in another ancestor, *LINEAR*. We will soon see how contracts are transmitted through inheritance.

The *Contract View* of a class is deduced (by removing secret information as explained in the earlier discussion) not from the original class text but from the flat view: what generally matters to clients is to know the exported features and their abstract properties, not whether the class introduced or inherited them. You can limit yourself to immediate features through the *Interface* View.

← *"What characterizes a metro line", page 53.*

16.2 POLYMORPHISM

Accumulating features is not the only purpose of inheriting from a class. After all, we were already doing this when we wrote *PREVIEW* as an heir to *TOURISM*. We should take advantage of inheritance as a relationship between types, not just modules: the dolphins-are-mammals and rings-are-groups story.

← *Page 16.*

Translating into programming terms the notion that "*any taxi is also a vehicle*" means accepting assignments between variables and expressions of these two types. With

```
my_vehicle: VEHICLE
cab_at_corner: TAXI
```

the inheritance structure described above makes this assignment valid:

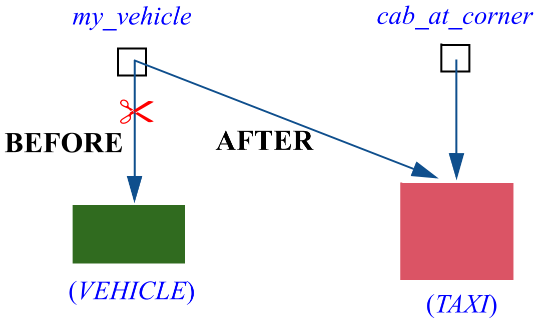
```
my_vehicle := cab_at_corner
```

It differs from the assignments we saw so far, starting with the chapter on this notion: until now the source expression and the target variable were of the exact same type, but here they are different. Specifically, the source type is a descendant of the target type.

← *Chapter 9 introduced assignment.*

The *effect* of the assignment is nothing new. Other than type considerations, all we learned about assignment still applies. Assuming the two variables were initially both attached to objects, the before-and-after picture is the same as with reference assignments seen earlier:

← *"Reference assignment", 9.5, page 252.*



Polymorphic assignment

It is a plain reference assignment; the attached objects — here the *VEHICLE* and *TAXI* objects — are not affected. The novelty is that a variable declared of a certain type, such as *VEHICLE*, may now become attached during execution not only to an object of that exact type, but also to one of any descendant type, such as *TAXI*.

Definitions

Here too we need appropriate terminology:

Definitions: Polymorphism

An *attachment* (assignment or argument passing) is **polymorphic** if its target variable and source expressions have different types.

An *entity* or *expression* is polymorphic if — as a result of polymorphic attachments — it may at run time become attached to objects of different types.

A *container data structure* is polymorphic if it may contain references to objects of different types.

← *Chapter 13 discussed containers.*

Remember that "entities" include variables (attributes, local variables) but also formal arguments of routines and **Result**.

"Polymorphism" is the existence of these possibilities. It is often confused with *dynamic binding*, to be studied next. Dynamic binding is related to polymorphism, and only makes sense because of polymorphism, but it is a separate concept.

As the definition notes, polymorphism results not only from assignment but also from argument passing. If some arbitrary class, say *DAILY_SCHEDULE*, has a routine

```
register_trip (v: VEHICLE)
```

then the following call is valid:

```
register_trip (cab_at_corner)
```

where the actual argument is of a proper descendant type. Most interesting here is the ability of a routine such as *register_trip* to deal with its formal argument on the basis of partial knowledge: all it knows is that at run time the argument's value will denote (be attached to) an object representing some kind of *VEHICLE*; that object could be a *TAXI*, a *TRAM* or any other kind of vehicle, but the routine does not know which, and the answer may change from one execution to the next.

Some of the corresponding classes may even have been written **after** the release of routine *register_trip*, as library designers concoct new kinds of *VEHICLE* to extend the existing framework. Think of what this means if you are asked to write such a routine: you may be dealing, through its arguments, with objects of types that have not yet been devised! The answer to this challenge will come from dynamic binding.

Polymorphism is not conversion

In spite of its name — from Greek words meaning "several shapes" — polymorphism does not cause any object to get a new "shape" (a new type) at run time. Polymorphic attachments are only applicable to reference types, with the effect described in the last figure: a reference reattachment. *No object changes type*.

You may occasionally need, aside from polymorphism, a way to transform objects; a simple example is the assignment of an integer value to a *REAL* target, whose internal representation is different. The appropriate mechanism here is not polymorphic reattachment but **conversion**.

Eiffel provides a general conversion mechanism, applicable to both reference and expanded types; it is the one that stands behind your ability to assign an integer to a real, but you can define conversions for your own types as well. If you devise a class *DATE* with attributes *day*, *month* and *year*, you can include a conversion routine from *DATE* to *STRING*, making it possible to assign a date to a string variable (with a result such as "3 Jun 2009" or any other format defined by the conversion procedure).

We will not explore the conversion mechanism further; if you need to use it, just look up the beginning of class *REAL_32* in EiffelBase (see the **convert** clause); this will be enough to get the basic ideas.

What matters for the present discussion is that conversion and polymorphism are exclusive mechanisms; if either of them is applicable, the other is not. So when you see an assignment *a* := *b*, or the passing of an argument to a routine, there is never any ambiguity as to what they mean.

Polymorphic data structures

In the definition of polymorphism the last case, polymorphic data structures (also called polymorphic containers), is particularly interesting. Consider a list — a typical container — intended to collect vehicles:

← *Page* 558.

```
fleet: LIST [VEHICLE]
```

and a call such as *fleet*.*extend* (…), adding an item to the list. What kind of argument can we use to replace the "…"? If you look up the declaration of *extend* in *LIST* [*G*] you see

```
extend (v: G )
          -- Add a new occurrence of v at end.
     …
```

In the case of *fleet* the actual generic parameter corresponding to *G* is *VEHICLE*, so the call expects an argument of type *VEHICLE*, as in

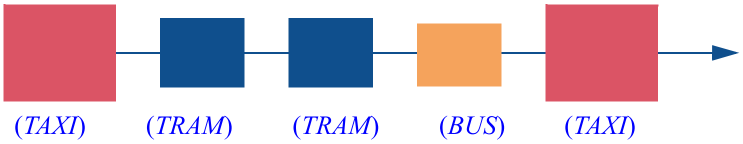
```
fleet.extend (my_vehicle)
```

But then polymorphism implies that wherever you need a *VEHICLE* any instance of a descendant type is fine too, so that — aside from the possibility for *my_vehicle* itself to be polymorphic — it is valid to use

```
fleet.extend ( cab_at_corner )
```

and in general any actual argument whose type is a proper descendant of *VEHICLE*, such as *TAXI*, *TRAM* and others.

A polymorphic container is the result of such a sequence of insertions, using possibly different actual types in each case. After a few such calls to *extend*, our *fleet* list might look like this:



A polymorphic list

with a mix of objects of different types, all descendants of *VEHICLE* (including *BUS*, not listed on the earlier figure).

The possibility of building such polymorphic data structures results from the combination of two fundamental object-oriented mechanisms: inheritance and genericity. It gives us a new level of flexibility. Consider for example the query *last*, which yields the last element of a list. It is declared in class *LIST* [*G*] as returning a result of type *G*. Since *fleet*: *LIST* [*VEHICLE*] uses *VEHICLE* as the actual generic parameter for *G*, the expression

```
fleet.last
```

is of type *VEHICLE*. In any particular call, the resulting object may be of any descendant type. If the list is in the state illustrated above, the object is a *TAXI*; but it might be of any other vehicle type, and you don't know. Nor do you *need* to know, since you can apply to that result any *VEHICLE* feature: after *v* := *fleet*.*last*, with *v*: *VEHICLE*, calls such as *v*.*load* (…) and *v*.*count* are valid — although not, of course, *v*.*take* (…) since *take* expects not just a *VEHICLE* argument but a *TAXI*.

I can hear you: it's unfair! Just look at the picture — the last element is a taxi! Why can't I use a perfectly respectable taxi operation like *take*?

Just calm down.

One, life is unfair, you might just as well get used to it.

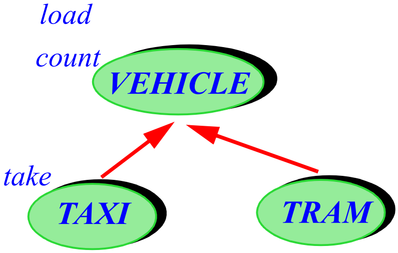
Two, how can you be so sure the last element *is* a taxi? A picture is just a picture; what really matters is the argument that this particular execution passed to *extend* or another list command in the last call that modified *fleet*.

And three — all right, this is the real answer — there does exist a technique to check that the last item is indeed a taxi in that particular execution, and, if it is, to subject it to specific *TAXI* features such as *take*. But for that you have to wait for almost forty pages. Did I mention that life is unfair?

→ *"Uncovering the actual type", 16.13, page 599*.

What is more immediately important is to understand the effect of the valid calls — calls to vehicle features, such as *v*.*load* (…) and *v*.*count* — in the case of a polymorphic target. The answer brings us to another fundamental object-oriented concept.

16.3 DYNAMIC BINDING



(*Original figure on page 554*.)

In a call such as *v*.*load* (...), with the declaration *v*: *VEHICLE*, the target *v* may be polymorphic; so at run time it may be attached to an object of type *TAXI* or *TRAM* or any other *VEHICLE* descendant. Now if you look up these classes you will note that each of them has a different implementation of the feature *load* (we will see below how to declare them without creating any ambiguity or conflict). Which version of the feature should the call actually solicit?

Only one answer really makes sense: we want to call *the right feature*. Right in the sense of the version that most closely matches the type of the *run-time* object that *v* denotes (is attached to) in that particular execution. When that object is an instance of *TAXI* you want the *TAXI* version of *load*; when it is a *TRAM* you want the *TRAM* version; and so on. Any other solution would be incorrect: not just the obviously foolish idea of applying a tram operation to a taxi object, but even applying to such an object the default version from class *VEHICLE* if there is one. Surely, if the author of class *TAXI* took the trouble to provide a special version of *VEHICLE* features directly adapted to *TAXI* objects, it was with the expectation that any call to the feature on a *TAXI* target will use that version.

The declaration of the target, *v*: *VEHICLE* in our example, has no effect here, and should not have any. Its purpose is different: to make the variable *v* general enough that in various executions it may sometimes denote a taxi object, sometimes a tram, sometimes other kinds of vehicles. But in any particular call it is not some abstract, non-denominational vehicle: it is either a taxi or a tram or something else just as specific. The features that execution applies to it should be specific too.

This policy, a cornerstone of the object-oriented form of software construction, has a name:

Definition: Dynamic binding

Dynamic binding (a semantic rule) is the property that any execution of a feature call will use the version of the feature best adapted to the type of the target object.

The reverse policy is called *static binding*. Surprising as it sounds, it does exist in some languages, notably C++; there static binding is the default and you only get dynamic binding if you request it explicitly by declaring the features ("functions" or "methods" in C++) **virtual**.

→ For details of the C++ policy: "Static and dynamic binding", page 826 (part of the C++ appendix).

16.4 TYPING AND INHERITANCE

As you have probably guessed from the examples, there are limits to the typing flexibility provided by polymorphism. It is OK to make a variable of type *VEHICLE* denote a *TAXI* or *TRAM* object, but it would not be right to let it become attached to an object representing a passenger or a city.

The rule is straightforward: in a polymorphic reattachment, the type of the source must be a descendant of the type of the target. This is not quite the proper terminology — we will see below how to get it right — but conveys the basic idea. It means that our earlier example assignment *my_vehicle* := *cab_at_corner* is valid, but not, for example, *my_vehicle* := *Paris*. The city of Paris is not a vehicle (only, since Rastignac, a vehicle for people's dreams).

This rule stands behind the various possibilities discussed so far: if we allow *x* := *y* with *x* of type *T* and *y* of type *U*, we must be sure that a call *x*.*f* will make sense at run time not only if the target object is of type *T* but also if it is of type *U*, as long as the call was accepted as **valid** at compile time. That validity condition is the usual one, based on the declaration of *x*: it states that *f* must be a feature of *T*. We need the guarantee that *f* is then also a feature of *U*; this is the case, by definition, if *U* is a descendant of *T*.

← *"Definitions: features of a class, immediate, inherited, introduce", page 556.*

The following terms help in understanding these concepts:

Static type, dynamic type

The **static type** of an entity or expression *e* is the type used in its declaration in the corresponding class text.

If the value of *e*, during a particular execution, is attached to an object, the type of that object is *e*'s **dynamic type**.

After the assignment *my_vehicle* := *cab_at_corner*, the entity *my_vehicle* has dynamic type *TAXI*. Its static type is the type used in its declaration: *VEHICLE*.

These notions apply to entities and expressions. For *objects* there is no need for the distinction: an object only has a dynamic type; it is a *TAXI* or a *TRAM* or something else, but only one of these, and will not change its type.

A consequence of the basic type rule, expressed with these notions, is that **any dynamic type for an entity or expression must conform to its static type**.

The term "conform" used here generalizes to types what "descendant" means for classes. As you remember, the difference between classes and types comes from genericity. A non-generic class is also a type (which has enabled this discussion to use just "descendant" so far); but if a class is generic you need to provide actual generic parameters — types themselves — to turn it into a type. *Conformance* holds, for reference types, if *descendance* holds between the underlying classes and the actual generic parameters if any are the same:

← *"Classes vs types", page 369.*

- *TAXI* (no generic parameters) conforms to *VEHICLE* since it is a descendant.
- *LINKED_LIST* [*TAXI*] conforms to *LIST* [*TAXI*] since *LINKED_LIST* is a descendant of *LIST* and the actual generic parameter is the same.

Here is a more precise statement of this property:

Definition: conformance

If a class *D* is a descendant of a class *C*, both non-expanded, then types derived from *D* conform to those derived from *C* as follows:

- If the classes are not generic, then *D* (as a type) conforms to *C*.
- If they are generic, then *D* [*T*, *U*, …] conforms to *C* [*T*, *U*, …] (with the same generic parameters).

An expanded type conforms only to itself.

> The full definition also accepts conformance if the actual parameters are not identical but conform; for example *LINKED_LIST* [*TAXI*] conforms to *LINKED_LIST* [*VEHICLE*] (and hence to *LIST* [*VEHICLE*]). This case requires special precautions and we will not need it. As usual with language issues, consult the language standard if you want to know the full story.

www.ecma-international.org/publications/standards/Ecma-367.htm.

All this just to have a correct version of the polymorphism rule. The precise rule must mention conformance rather than descendance:

Polymorphism type rule

For a polymorphic attachment to be valid, the type of the source must conform to the type of the target.

This confirms that polymorphism is only applicable to reference types since, per the above definition, an expanded type only conforms to itself.

The typing rules give us safety, by ensuring that in the fundamental operation of object-oriented programming

```
x.f (…)
```

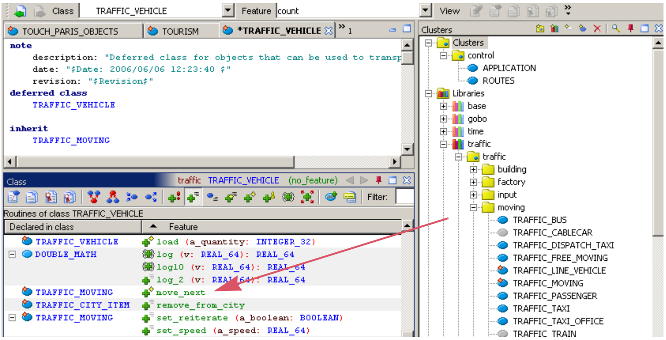
there will always be a feature *f* applicable to the object to which *x* is attached at run time, whatever its type may be as a result of polymorphic attachments to *x*. It is interesting to note the combination of properties that characterizes most recent object-oriented languages:

- **Static typing**, guaranteeing that there is *at least one* feature for *f*.
- **Dynamic binding**, guaranteeing use of *the best* feature — the one directly appropriate for the object's type — if more than one feature variant would be available.

The Smalltalk language departs from this policy by combining dynamic binding with dynamic typing: invalid feature applications, say *Paris*.*load* (…) applying a *VEHICLE* feature to a *CITY* target, are not flagged statically but detected only at run time, where they typically cause the program to terminate abnormally. The goal is to avoid the overhead of compile-time checks and to provide more flexibility. Subsequent object-oriented languages — their task made easier by advances in performance — have tended to apply static typing.

16.5 DEFERRED CLASSES AND FEATURES

Looking up the list of features in class *VEHICLE*:



```
note
    description: "Deferred class for objects that can be used to transp
    date: "$Date: 2006/06/06 12:23:40 $"
    revision: "$Revision$"
deferred class
    TRAFFIC_VEHICLE

inherit
    TRAFFIC_MOVING
```

Routines of class TRAFFIC_VEHICLE

Declared in class	Feature
TRAFFIC_VEHICLE	load (a_quantity: INTEGER_32)
DOUBLE_MATH	log (v: REAL_64): REAL_64
	log10 (v: REAL_64): REAL_64
	log_2 (v: REAL_64): REAL_64
TRAFFIC_MOVING	move_next
TRAFFIC_CITY_ITEM	remove_from_city
TRAFFIC_MOVING	set_reiterate (a_boolean: BOOLEAN)
	set_speed (a_speed: REAL_64)

```
Clusters
    control
        APPLICATION
        ROUTES
Libraries
    base
    gobo
    time
    traffic
        traffic
            building
            factory
            input
            moving
                TRAFFIC_BUS
                TRAFFIC_CABLECAR
                TRAFFIC_DISPATCH_TAXI
                TRAFFIC_FREE_MOVING
                TRAFFIC_LINE_VEHICLE
                TRAFFIC_MOVING
                TRAFFIC_PASSENGER
                TRAFFIC_TAXI
                TRAFFIC_TAXI_OFFICE
                TRAFFIC_TRAIN
```

Features of VEHICLE in EiffelStudio

you see a feature *move_next*. In the class text it does not appear; indeed *move_next* is inherited from *MOVING*, but in the text of *MOVING* you will find

```
move_next
        -- Move to following position as determined by schedule.
    deferred
    end
```

This is a new form of declaration for us; previous routines had a body starting with **do** and continuing with instructions. Declaring a feature **deferred** means that it has a specification — signature and contract — but no implementation. The implementation is deferred (hence the name) to descendant classes. Indeed in class *TAXI* you will see

We may also say "deferred routine", but the standard term is "deferred feature" since the specification does not prescribe eventual implementation as routine or attribute.

```
move_next
        -- Move to following position as determined by schedule.
    do
     … A sequence of instructions …
    end
```

providing an actual implementation corresponding to the needs of *TAXI* moving objects. Class *TRAM* provides another implementation.

We say that these classes **effect** the feature — make it **effective**. From now on we have two kinds of feature: effective (specification plus implementation) and deferred (specification only). From features these terms carry over to classes and then types:

This is known as an "effecting".

Deferred classes and types

If a class *C* has at least one deferred feature (either introduced as deferred in *C* itself, or inherited as deferred and not effected), it is a **deferred class**.

A type based on a deferred class is a **deferred type**.

A type or class is **effective** if it is not deferred (all its features are effective.)

As usual, the distinction between classes and types is due to genericity. You can see for example that *LIST* is a deferred class (because it describes the general notion of list, with implementations in effective descendants such as *LINKED_LIST*); *LIST* [*TAXI*], and more generally *LIST* [*T*] for any *T*, are deferred types.

Deferred classes are also called "abstract", and effective classes "concrete".

The reason why *move_next* is deferred in class *MOVING* is that at this level:

- We know that the feature applies to all moving objects, and want to ensure that all actual vehicle objects have a version of it.
- Because *MOVING* describes the general concept of moving object, not a specific kind, it is not possible to provide a default implementation.

MOVING specifies the existence of the feature, but it is the responsibility of proper descendants, such as *TAXI*, to offer their own implementations.

The absence of a default implementation in *MOVING* raises an obvious issue: what would be the effect of *m*.*move_next* if the object attached to *m* is of type *MOVING* or *VEHICLE*, both of which have *move_next* deferred, hence not implemented? The answer is simple: there are no such objects.

Non-Deferred Creation rule

The target type of a creation instruction may not be deferred.

This prohibits us from creating an object of type *MOVING* or *VEHICLE*, for example in **create** *my_vehicle* (with the earlier declaration): it would be impossible to carry out a call to a deferred feature such as *move_next* on the resulting object.

So if a type is deferred there are no *objects* of that type. But you may still have *variables* (and other entities, and expressions) of the type, such as *my_vehicle*. They will become attached to instances of effective conforming types, such as *TAXI* in this example. Indeed the whole idea of deferred features, classes and types only makes sense thanks to polymorphism and dynamic binding.

Because creation requires an effective type, you must know, when you want to use a class, whether it is effective or deferred; this tells you whether you can use **create** *x* on variables of the corresponding types. To find out, you should not have to look up the class text (or, because deferred features can be inherited, the text of ancestors or the flat view); this would defeat information hiding. Being deferred or not is a key property of the class and should be advertised visibly. In fact it is (after the **note** clause which provides general information about the class) the first thing the class text shouts to the world:

← "The flat view", page 556.

Deferred Class rule

The declaration of a deferred class must start with **deferred class** (instead of just **class** for an effective class).

Subject to variations of syntax and terminology, the previous rules and definitions were independent of the programming language. This one is an Eiffel rule.

"A deferred class" in this rule has the meaning of the preceding definition: a class that has at least one deferred feature, immediate or inherited. The rule states that this should not just remain insider knowledge but becomes part of the interface of the class, as reflected in its contract view. In our example you can check that *MOVING* and *VEHICLE* start with **deferred class**.

← "What characterizes a metro line", page 53.

Eiffel allows you to declare a class **deferred class** even if it does not have any deferred features. The class is then considered deferred, as an extension to the above definitions. This is useful if you want to force the use of the class as an ancestor in inheritance hierarchies and to prohibit instantiating it.

To be more precise, declaring a class **deferred** prohibits *direct* instantiation. The concepts of this chapter call for revisiting the notion of "instance". So far an instance of a type (and of the underlying class) was just a run-time object matching the description given statically by the class. This remains correct but we need to distinguish between direct and indirect cases:

Instance, direct instance

An object obtained through a creation instruction is a **direct instance** of the instruction's target type.

An **instance** of a type is a **direct instance** of any conforming type.

Including the type itself.

A consequence of the Non-deferred creation rule is that a deferred type has no *direct* instances; this is the case with *VEHICLE* and *MOVING*. Both have *instances*, though: the direct instances of effective descendants such as *TAXI*.

These definitions directly reflect the concept of polymorphism: declaring *x* of type *T* means you want *x* at run time to denote instances of that type; with polymorphism we understand this as covering not only objects deduced exactly from the declaration of *T* — its direct instances — but, recursively, instances of any conforming type.

Deferred classes are a major part of the effectiveness of inheritance as a taxonomy mechanism. Often, the higher levels of an inheritance hierarchy consist mostly of deferred classes, introducing abstract concepts whose implementation is relegated to the effective descendants. You can readily look up two extensively developed examples:

- The EiffelBase library of data structures and algorithms. The library introduces a global taxonomy of the fundamental structures of computer science; towards the top, you will see classes such as *LINEAR* (structures that can be traversed one-way) and *FINITE* (finite structures); more specific structures are covered by effective classes descending from these general abstractions.
- The EiffelVision graphics library. Look up for example the "figures" cluster, with a general notion of *FIGURE*, represented by a deferred class, and its progressive specialization all the way down to effective classes representing concrete kinds of figure such as circles and squares. The taxonomy of geometric figures is a standard example in textbook presentations of inheritance, but there is nothing academic about its use in EiffelVision.

Some programming languages, notably Java and C#, offer a language construct known as the **interface**, presented in the appendices covering these languages. An interface is similar to a class but its features have no implementation whatsoever. It is like a class where all features are deferred (and do not have any

← Not to be confused with the general software engineering concept of interface introduced in chapter 4.

contracts). Deferred classes are more general: as the definition indicates, a class is deferred as soon as it has *one* deferred feature, but it can have a mix of deferred and effective features. This enables the deferred class mechanism to support an incremental mode of development: you start out by identifying key abstractions, which you define through classes with most features deferred ("very deferred" classes, close in spirit to interfaces, but with contracts); then, as you explore the problem in more detail, you introduce more and more effective descendants. This is the object-oriented variant of the general technique of **refinement**.

← "Deferred classes and types", page 566. On how interfaces compare to deferred classes, see also "Using multiple inheritance", page 588 below.

← Refinement was discussed in "Overall setup", 6.1, page 108.

A particularly powerful technique involving partially deferred classes is the **Program with Holes** design pattern, which has effective routines calling deferred ones. It captures a general scheme whose details are left for descendants to implement, hence combining reusability with adaptability. As a typical example, you have seen many times the standard scheme for repeating (*iterating*) an operation over all the elements of a sequential structure such as a list; applied to the search operation, it reads

```
search (v: G)
        -- Move to first position (at or after current position) where
        -- v appears. If no such position ensure that exhausted will be true.
    do
            from until v = item loop forth end
    end
```

EiffelBase classes representing lists and other such structures all inherit from a deferred class *LINEAR* where this code appears (in a more complete version taking into account the distinction between reference and object equality — for details bring up the class text in EiffelStudio). The implementation of *forth*, however, depends on the representation chosen for lists: linked, or using an array etc. So in *LINEAR* the feature is deferred:

```
forth
        -- Advance cursor by one position
    require
        in_range: not after
    deferred
    ensure
        increased: index = old index + 1
    end
```

Note the precondition and postcondition; as discussed below, contracts are fully applicable to deferred features and classes.

← "What happens to contracts?", 16.9, page 580.

Then every effective descendant of *LINEAR* effects *forth*; for example in an arrayed list, where the cursor is simply represented by the integer attribute *index*, the implementation is just *index* := *index* + 1; in a linked list (look up the feature in *LINKED_LIST* and *TWO_WAY_LIST*) the details are more complicated. What matters is that *search* does not have to know such details; all it needs is the ability to rely on a feature *forth*, of which it knows the specification, as expressed by the contract, but not the implementation.

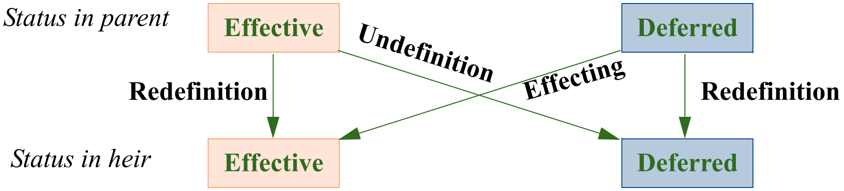
The name "Program with Holes" reflects this approach to incremental software construction: at each level of abstraction, capture all the relevant information, including those that can be fully implemented (effective features with no calls to deferred ones), those that can only be specified (deferred features) and those that can be implemented by relying on deferred features, such as *search* in *LINEAR*. We may view the result as a partially constructed program, with holes to be filled through refinement in descendants.

This is one of the most important contributions of object-oriented methodology to the orderly construction of systems, in particular large ones.

16.6 REDEFINITION

When a class effects a deferred feature, it provides the first implementation of a mechanism that until then (in ancestors) had a specification but no implementation. To take further advantage of dynamic binding and make the architecture even more flexible, you may provide a new implementation for a feature that *already* had one in the parent from which a class inherits it.

We will say that the class **redefines** the feature; another term, used in particular with the C++ language, is **overriding**. Redefinition complements the just studied mechanism of effecting:



Forms of redeclaration

The rightmost case of the diagram shows that when inheriting a deferred feature you can not only make it effective (per the branch labeled "effecting") but also change it while leaving it deferred; this can only be to change its signature or contract, and is considered a form of redefinition, like the leftmost case. To complete the list of cases we also have *undefinition*, through which you forget the implementation of an effective feature by making it deferred again, ready to embark on a new life of passionate effectings.

The term *redeclaration* covers all these cases:

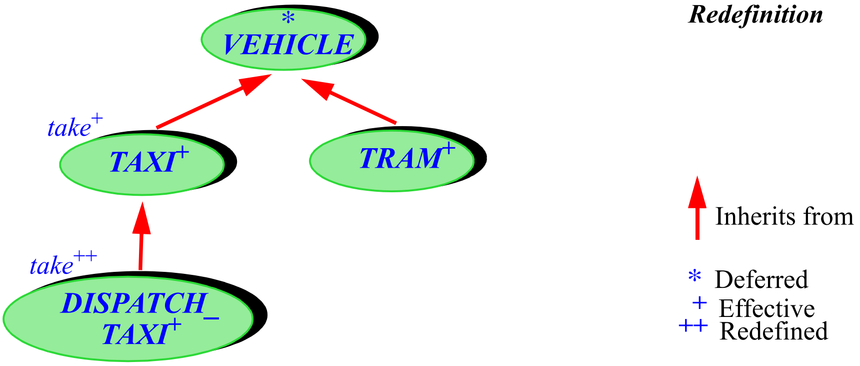
Definition: redeclaration

To **redeclare** an inherited feature is to change any or all of its signature, contract and implementation, or remove its implementation. Variants include effecting, redefinition and undefinition.

We do not need to study undefinition here as it is a fairly specialized operation (although not particularly hard or mysterious — you may find examples in EiffelBase by looking for the keyword **undefine**). Redefinition, on the other hand, is a widely used mechanism.

As an example, consider the Traffic class *DISPATCH_TAXI*, which inherits from *TAXI* and represents the notion of a taxi that is under the control of a dispatching office, as opposed to just cruising the streets and finding customers on its own. The procedure *take* has a different implementation for *DISPATCH_TAXI*, since when a dispatch taxi picks up a customer it must communicate this to the dispatching office and perform a number of auxiliary operations. On inheritance diagrams we represent a redefined feature through the symbol ++ (the idea being that one + represents effecting, the process of getting an implementation for the first time, and two mean changing that implementation status — becoming, as it were, even more effective):

Redefinition



The absence of any symbol means “effective” by default. Figures generally omit the single “+” except to emphasize that a particular feature or class is effective.

To redefine an inherited feature you provide a new declaration in the class text, but you must also announce the redeclaration to the world, loud and clear, through a **redefine** clause in the corresponding **inherit** part:

```
class
    DISPATCH_TAXI
inherit
    TAXI redefine take end
feature
    take (from_location,to_location: LOCATION)
            -- Bring passengers from from_location to to_location
        do
            … New implementation …
        end

    … Other features and rest of class …
end
```

If the class redefines several features it will list them all: **redefine** *f*, *g*, … **end**.

The purpose of the **redefine** clause is clarity and safety. It is an important rule of sound object-oriented programming that within a class a feature name should never denote two different features, a case known as **overloading** (permitted by some languages as discussed in the corresponding appendices, but causing a risk of confusion). The **redefine** clause clarifies, for a reader of the class text, the relationship of the feature to its namesake in the parent class: it is not a new feature with the same name but a re-implementation of the same feature.

If you omit the clause, you will get a compile-time validity error since your class is now considered to have two features with the same name, a forbidden case of overloading.

The requirement of listing an inherited feature under **redefine** applies to redefinition only. For undefinition, you will similarly use an **undefine** clause and provide a new, deferred declaration. For effecting, as in the earlier examples, there is no such clause; no conflict arises since the feature did not have an implementation before, so the new, effective feature naturally subsumes the version inherited in deferred form, giving it the implementation it had been waiting for all along in ancestors.

Redefinition, like effecting, works with dynamic binding: in

```
group_move (taxi_fleet: LIST [ TAXI ]-- See below about TARGET
        -- Make all taxis in taxi_fleet follow the same route.
    do
        from taxi_fleet.start until taxi_fleet.after loop
            taxi_fleet.item.take (…)
            taxi_fleet.forth
        end
    end
```

every element of the list will execute the version of *take* from either *TAXI* or *DISPATCH_TAXI* depending on whether it is a direct instance or one or of the other. This is like in our earlier examples of polymorphic variables and data structures; the only difference is that *TAXI* is effective and hence has direct instances, whereas *MOVING* and *VEHICLE* are deferred and have none.

If you redefine a feature, the parent version is known as the feature's **precursor**. Fairly often, the new implementation needs to rely on the precursor version. Rather than duplicating the code (did I ever mention that copy-pasting code is not a good idea?), you may use the keyword **Precursor**. The new declaration of *take* in *DISPATCH_TAXI* looks like this:

```
take (from_location,to_location: LOCATION)
        -- Bring passengers from from_location to to_location
    do
        Precursor (from_location, to_location)
        … Other operations, specific to dispatch taxis…
    end
```

This means that the implementation starts by doing whatever its precursor version in *TAXI* did, then adds its own operations specific to dispatch taxis.

In a feature redefinition like here, you may use **Precursor** just like a feature name; if the feature requires arguments you pass them — here *from_location*, *to_location* — as you would to the feature itself.

16.7 BEYOND INFORMATION HIDING

The combination of mechanisms described so far plays a key role in obtaining flexible, reusable and (thanks to static typing and also to the contract rules, yet to come) reliable systems. The particular contribution of polymorphism and dynamic binding is to take **information hiding** one step further. The basic idea of information hiding is to let clients use supplier mechanisms without having to know how they are implemented. The next advance is to protect the client from having to know *which of the possible suppliers* is being used in a particular case. When you write

← *"Information hiding", 8.5, page 218.*

```
my_vehicle.load (…)
```

you are asking for a certain abstract operation, *load*, to be applied to the target object; but since you do not know — with *my_vehicle* possibly polymorphic — what exact type the object will have, dynamic binding means that you also do not know what exact feature will be triggered.

This is the reason why contracts are so important. What you do know, captured by the original contract of *load* in the deferred class *VEHICLE*, is the essential semantics of *load*: that it adds *n* passengers to the vehicle. This is common to all the variants, although it is up to each of them to fill in the details.

Beware of choices bearing many cases

To appreciate the value of the object-oriented style permitted by polymorphism and dynamic binding, consider what you would do, without these mechanisms, to achieve the same goal of adapting an operation to the type of its target. It is of course not hard to come up with a solution:

Warning: non-OO style, not recommended.

```
load (v: VEHICLE; n: INTEGER)
        -- Add n passengers to v.
    do
        if "v is a tram object" then
            "Apply tram loading algorithm"
        elseif "v is a taxi object" then
            "Check that the capacity is 4 at most"
            "Apply taxi loading algorithm"
        elseif …
        end
    end
```

We just test for the type and apply the corresponding algorithm. The technique works, but with unpleasant consequences for software architecture:

- It yields long, complex conditional instructions (which could use a multi-branch rather than the **if** form, without affecting this discussion).
- It has to be repeated for every operation such as *load* that is conceptually applicable to any vehicle but carried out differently for different vehicles. There may be many such operations.

← "Multi-branch", page 195.

The problem with such verbose and repetitious code is that it damages the software's prospects of smooth evolution. The need to add a type variant — in the Traffic example, a new kind of vehicle — arises frequently in the evolution of a software system; every such addition requires going back to every routine that discriminates on types in the above style.

In contrast, the dynamic-binding solution requires you only to: add a class; give it a proper place in the inheritance hierarchy; and write an implementation for the features that need a specific variant for that class. In many cases there is no need to change anything in previously written software.

As a general rule, be wary of decision structures with many branches as above. They often reflect bad design, damaging extendibility:

Touch of Methodology: Fight the *Many Explicit Variants* syndrome

If your design leads you to a control structure involving conditional instructions with many branches, examine whether a simpler and more extendible solution is possible through dynamic binding.

This is a design guideline and not an absolute rule; not all conditional instructions should be replaced by dynamic binding. The most suspicious ones discriminate along the **type** of an object, as in the example. Then the solution is to reverse the architecture: instead of asking *algorithms* to choose between types of applicable *objects* — requiring them to know about all possible such types — equip each object type with the applicable algorithm variants.

What makes this solution better is inheritance, meaningful for types but with no equivalent for algorithms: when adding a class, once you have found its proper place in the inheritance hierarchy you only have to redeclare the features that need a special version for that class. The others you'll just inherit.

This discussion has assumed all along that we are dealing with a known set of operations and newly appearing types. It does not address the reverse situation, or the case of frequently adding both types and operations. "Visitor" techniques, studied at the end of this chapter, provide the complement.

→ *"Reversing the structure: visitors and agents", 16.14, page 606.*

16.8 A PEEK AT THE IMPLEMENTATION

In general this book presents programming concepts from the viewpoint of their use by application programmers, not their implementation by compiler writers — a fascinating topic, but for other books. We have already made a few exceptions; dynamic binding provides another one. Taking a little break away from pure concepts to look under the hood will give us a better grasp of the issues.

→ *As this is not critical to the discussion of programming principles you may prefer on first reading to skip to 16.9, page 580.*

For simplicity let us just consider routine calls (although the situation with attributes is similar). *Without* dynamic binding, the compiler knows, when it sees a call, what feature to specify in the generated code. For example static binding for the call

```
cab_at_corner. load (…)
```

means applying the *TAXI* version of *load*, as deduced from the type declared for the target *cab_at_corner*. To describe the generated code let us use C, which is low-level enough to be representative of assembly languages, but still understandable because it is not tied to a particular platform. (In addition, of course, the EiffelStudio compiler does generate C as one of its possible outputs, so this realistic.) Without dynamic binding, the generated code for the above call would be something like:

→ *Appendix C covers the C++ language and appendix D its C subset.*

```
C_TAXI_load  (C_cab, …);                                        [C1]
```

Note: *this is C code.*

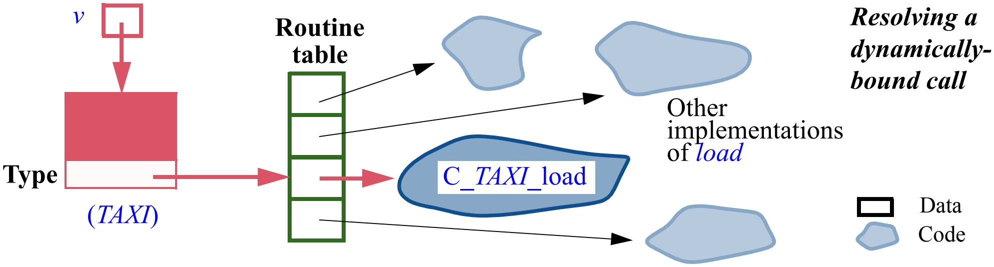
where C_*TAXI*_load is the C translation of the routine *load* in its version from class *TAXI*. Since C is not object-oriented and has no notion of a qualified feature call *x*.*f* (…), the C routine ("function") must take one more argument, corresponding to the target, here C_cab representing the original *cab_at_corner* from Eiffel.

In this setup the routine, C_*TAXI*_load, is known at compile time; it can appear explicitly, under its actual name, in the generated code.

With dynamic binding this is no longer the case. Of various *load* versions available as a result of the compilation, only at run time can it be determined which one a particular call must use. The generated code for an O-O call

```
v.load (…)
```

must rely on some appropriate data structure, such as one looking like this:



Resolving a dynamically-bound call

The key structure is a **routine table** (also known as a dispatch table, a virtual table or just a vtable). Entries in that table are references; unlike our usual references they point not to other data but to *code*. Specifically, each entry points to one of the versions of a routine, in this case *load*.

Such references (or "pointers") to code are not something that you can directly get in Eiffel and most other high-level languages, but at the machine level they are a direct result of the *stored-program computer* concept: since code, along with data, resides in memory, a table entry can contain the address of the start of a block of code, and the instruction set of any computer includes an instruction of the form "execute the code that starts at address *a*" for given *a*.

← "The stored-program computer", page 10.

High-level languages do not need such a facility because it is error-prone: what if the memory content starting at address *x* is not code, or is code but planted by a malicious hacker? They provide safer replacements, including, for O-O languages, dynamic binding. In Eiffel another higher-level mechanism is *agents*: an agent wraps some routine and can be passed around to various parts of the software,

→ Chapter 17.

> enabling them to call the associated routine. Some non-O-O languages offer, as another technique, the ability to pass a routine as argument to another routine. A section in the chapter of agents discusses these and other language mechanisms allowing programs to delay the selection of a routine until run time.

→ *"Other language constructs", 17.8, page 654.*

With dynamic binding, the routine selection depends on the type of the target object. In the figure that object, shown on the left, is an instance of *TAXI*. We cannot deduce this from the program text, where it is only known through the variable *v*, of the more general type *VEHICLE*; but if *v* is polymorphic, the corresponding reference may in a particular execution be attached to a *TAXI* object, and — this is the whole idea of dynamic binding — it's that type that counts, not the declaration of *v* which was only meant, in the program text, to preserve generality by permitting other kinds of object in other executions.

If we consider these properties with the eyes of a compiler writer, a clear consequence is that **every object must contain the identification of its own type**. Otherwise there would be no way to achieve dynamic binding, since the type determines the routine to select among all the possible variants. Implementations of object-oriented languages indeed include in the representation of any object, in addition to the fields denoting the object's attributes, one denoting its type; it is marked "**Type**" in the figure. Typically this is represented as an integer, filling up a word: 4 or 8 bytes — enough to cover all the possible types in any actual system. The EiffelStudio implementation adds yet another word to every object, for control information needed in particular by garbage collection. Such is the **space overhead** for O-O mechanisms in that implementation: two words per object. It is generally acceptable, but may become an issue if you have a very large number of very small objects.

You can trace in the figure, starting at the top left and following arrows, the machinery of dynamic binding for a call *v*.*load* (…). If the value of *v* is not void, following the corresponding reference leads us to an object. The "**Type**" field of that object gives us the integer representing the object's type, here *TAXI*. We use that integer to index into the routine table for *load*; the corresponding table entry yields the address of the program code for the appropriate routine variant.

Taken literally, this scheme would give the following C code, to be compared to the static binding implementation [C1]:

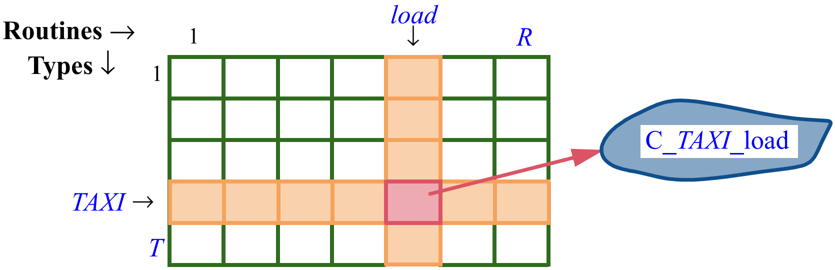
← *Page 576.*

```
(routine_table [ (*v).type]) (v, …);                    [C2]
```

Note: this is C code.

Explanation: * is the de-referencing operator, so *v is the object pointed to by v; then (*v).type (which can also be written v–>type) is the object's type field, which we use as an index into the array routine_table; this is the address of the corresponding C routine, which we apply directly to the arguments. As before, the argument list includes, in addition to the original arguments, the current object — here known simply through v, which plays the earlier role of C_cab.

From this basic scheme many variants and optimizations are possible. First, to understand the issue in full generality we note that the collection of all routine tables, each indexed by types, is a two-dimensional structure with *T* rows and *R* columns for a system with *T* types and *R* routines, as shown in the following figure.



Type-routine table

General scheme for dynamic binding implementation

▭ Data
⬡ Code

The solution so far, leading to the [C2] code style, splits this table along columns, each of them a routine table. We may instead split it by row, each a “type table”. Or we might have a two-dimensional array as suggested by the last figure. All these solutions satisfy a crucial requirement:

Efficient implementation of dynamic binding

A good implementation of dynamic binding should ensure that the time to find the appropriate routine version is **O** (1).

The time overhead of a dynamically-bound call over its statically-bound counterpart [C1] is the cost of finding the appropriate routine at run time. The flexibility brought by polymorphism and dynamic binding is worth a price, but not any price; in particular it is essential, as the principle states, to guarantee a constant upper bound on the execution time. Naïve implementations keep the inheritance structure at execution time and traverse it to find the applicable routine version. This technique is unacceptable as it introduces a direct conflict between the depth of the inheritance structure and program performance. (It becomes even worse with *multiple* inheritance.) All the array-based implementations discussed — routine-based, type-based, two-dimensional — satisfy the principle since finding the routine only involves indirections (following references) and array lookups, all constant-time operations.

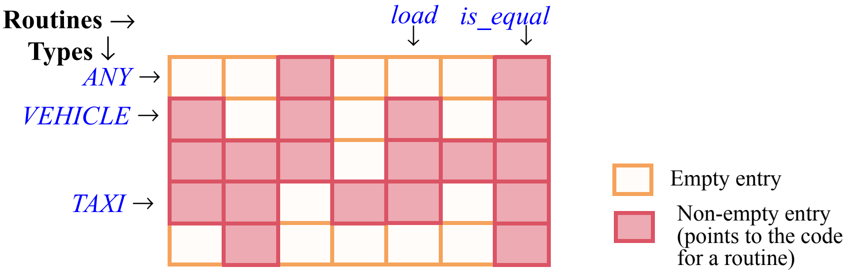
← “Using arrays”, page 388.

There is, however, a downside to this time efficiency: the added **space cost**. In all three variants, the above structure takes up *T* × *R* entries. This can be hard to justify; EiffelStudio itself, for example, needs over 6000 types and 50,000 routines. The table as pictured above is highly wasteful of space since in any practical case most of these entries will be empty; each routine is relevant for only a few types, for example *load* only for *VEHICLE* and its descendants.

For T types and R routines.

What counts is routines, not routine names! It does not matter how many unrelated routines elsewhere in your system happen to have the same name, here *load*.

In the variant that uses routine tables, we can trim each of these tables by removing, in each column, all entries before the first non-empty one (the "effective start" of the column) and after the last non-empty one:



General scheme for dynamic binding implementation

The above code [C2] will still work if a table entry routine_table [i] is indexed from the effective rather than physical starting point of the column; this is easy to achieve in machine code or C where an array is really nothing more than a starting address for a block of values, so that it suffices to offset that address by the index of the first non-empty entry in the column.

Such an optimization is not particularly interesting if the non-empty entries are scattered throughout a column representing a routine table. Assume for example that we have 6000 types and the type number is 1 for *VEHICLE*, 3000 for *TRAM* and 6000 for *TAXI*. Even if *load* exists only in these three classes, we still need the full column with 6000 entries, all but three of them empty. To improve this situation, noting that we have free choice for the numbers we assign to types, we can take advantage of the following property:

Feature Neighborhood theorem

The classes to which a feature belongs are the descendants of the class that introduces it.

This suggests choosing a type numbering scheme that gives neighboring numbers to descendants of any given class. By now you know what can help us here: **topological sort** as discussed in the previous chapter. "Descendant" is indeed a partial order since inheritance is acyclic.

Performing a topological sort based on the inheritance relation dramatically decreases the size of routine tables — by about 85% in EiffelStudio. This technique is essential; without it the space overhead would be hard to bear.

On the time side the overhead is, as required above, constant-bounded (even in the not yet discussed case of *multiple* inheritance) and actually quite small: essentially (see [C2]) an indirection, a field access and an array access. Better yet, **the overhead can disappear altogether** in certain cases: the compiler can find out that:

- A certain routine has only one version.
- A certain expression is not polymorphic

In these cases it can apply the static binding scheme [C1] and avoid any time penalty at all. This is, by the way, the reason why some older O-O languages offer static binding as an option — the default in C++, which reserves dynamic binding to "virtual" routines. The problem with this policy is that it is easy for a programmer to make the mistaken assumption that a call is static whereas it should be polymorphic; or maybe the decision was correct at some stage, but then you add a descendant class with a new version of the routine and forget to make the routine virtual.

The rule to remember is that **dynamic binding is always the correct semantics** for object-oriented calls. As a consequence static binding is only acceptable when it has the same semantics as dynamic binding, typically in one of the two cases cited. Because it is hard to detect such cases, static binding is better left as a **compiler optimization**.

In the EiffelStudio environment, such optimizations are indeed the responsibility of the compiler technology.

→ *"The melting ice technology", page 357.*

These observations complete our foray into language implementation techniques, which I hope will have given you a good grasp of what inheritance and associated techniques mean for the execution of O-O programs. There are of course many details to account for in practice; if you want to get to the bottom of things, the best place to start is an examination of the C code generated by the EiffelStudio compiler in "classic" mode. (The next step would be to look at the source code of EiffelStudio itself, all available as open source.) The two key points are that:

The other mode is ".NET", which does not generate C.

- The time overhead for dynamic binding can be constant-bounded and small; proper techniques reduce it to zero in applicable cases.
- The space overhead includes a field in every object, and tables that can be limited to an acceptable size through proper techniques.

16.9 WHAT HAPPENS TO CONTRACTS?

The definition of a feature does not just consist of a name, signature and (for an effective feature) implementation, but may also include a precondition and postcondition. For a class, we have the invariant. We know what they mean in the absence of inheritance. How does inheritance affect the picture?

Invariant accumulation

The first rule affects class invariants. It reflects the “is-a” view of inheritance and its role as a taxonomy mechanism. Writing *TAXI* as an heir of *VEHICLE* is not just a matter of convenience but makes a statement that polymorphism will work: when a vehicle is expected, for example as part of a *LIST* [*VEHICLE*], a taxi will do. This means that any constraint that has been defined for instances of the parent class must apply to those of the heir.

In *VEHICLE* we find

```
invariant
    not_too_small: count >= 0
    not_too_large: count <= capacity
```

expressing that the number of passengers (*count*) is non-negative and bound by the defined *capacity*. You will not find such clauses in *TAXI*; not because they have somehow stopped being applicable, but for the exact opposite reason: the clauses are automatically there. A class inherits from a parent not only its features but its class invariant.

You will see these inherited invariant clauses if you look up the **flat view** or the contract view of the class. This will also reveal that the heir may introduce supplementary constraints. Indeed the text of *TAXI* includes the clause

← *“The flat view”, page 556.*

```
invariant
    legal_limit: capacity = 4
    … Other clauses affecting taxi-specific features …
```

which comes in addition to those of *VEHICLE*; the flat view lists the parent clauses first, then the ones added by the heir.

A natural question is what would happen if these contradicted the parent properties, for example by stating *capacity* = –1. But this is nothing new compared to the possibility of including two contradictory invariant clauses in a *single* class, such as *a* >= 0 and *a* = –1. Such an invariant is simply wrong, and will be caught through testing (or in the future by static analysis).

The following definition captures the semantics:

Definition: Invariant of a class

The **invariant of a class** is the assertion (p_1 **and** … **and** p_n) **and then** i, where i is the assertion listed in the class’s own **invariant** clause (or **True** if it does not have one), and $p_1 \dots p_n$ are (recursively) the invariants of its parent classes if any.

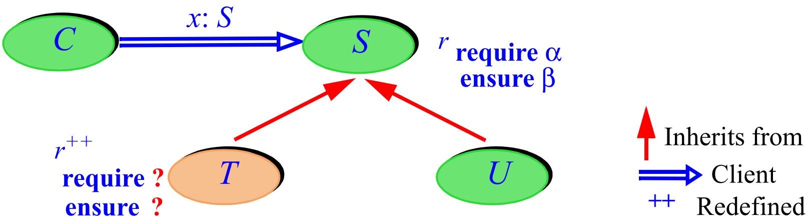
← *On* **and then** *see “Semistrict boolean operators”, 5.3, page 89.*

> The definition accounts for possible multiple parents, as studied later in this chapter (and including *ANY* if applicable, see next). The assertion in an **invariant** clause may consist of several subclauses, as in the above examples; in this case there already is an implicit **and then**.

In our example the invariant of *TAXI* is the conjunction of the clauses appearing in the texts of both *VEHICLE* and *TAXI*.

Precondition weakening and postcondition strengthening

The second issue is what happens to preconditions and postconditions and leads to an important rule of software development. To understand the issue we must consider it in the context of polymorphism and dynamic binding.



The contract adaptation context

Consider a routine *r* in a supplier class *S*, with a precondition and a postcondition (called α and β in the figure). A proper descendant *T* redeclares *r*; the figure shows this to be a redefinition, but it could be an effecting if *r* was deferred in *S*. The question is: to what extent can the redeclared version change the contract of the feature, here α and β?

← "Redeclaration" covers redefinition and effecting. See "Definition: redeclaration", page 571.

To devise the answer we must look at the perspective of a client class *C* that includes a call *x*.*r* (…) for *x* declared of type *S*. The contract gives the author of *C* a directive and a guarantee: make sure to satisfy the precondition when you call me, and you are entitled to assume the postcondition when I return. For example, the following scheme, which establishes the precondition in the simplest possible way — testing for it in a conditional — is guaranteed to work:

```
if x.α then
    x.r (…)                                          [3]
    -- Here x.β is guaranteed to hold
end
```

This is the direct application of Design by Contract, which we have used many times. But now we have polymorphism. Our *x*, declared of type *S*, is no longer certain for any particular run-time call to be attached to a *direct instance* of *S* (an object exactly of type *S*): it could denote a *T* object, or an instance of any other descendant of *S* such as *U* shown in the figure.

Because of dynamic binding the above call will, if the descendant has redeclared *r* as *T* does, use the redeclared version. But of course the author of a client class such as *C* does not necessarily know this. It's actually more devious: the scheme may well arise for a class *T* that did not even exist when *C* was written; in *C*, the above code may have been in a routine

```
do_something_with_an_S_object (x: S)
```

so that *x* is something of type *S* — meaning, *S or conforming* — that is passed to *C* by the outside world. Two years later *T* gets added to the class hierarchy, some other code gets written that knows about *T* and calls *c1*.*do_something_with_an_S_object* (*t1*) with *c1* of type *C* and *t1* of type *T*. Pity the original author of class *C*, who must write client code and guarantee its correctness even though it deals with objects of types yet to be conceived!

To ensure such correctness, *C* can only deal with the advertised properties of its suppliers such as *S* and their features such as *r*. This strictly limits how descendants such as *T* can fool around with the contract. Specifically:

- Making the precondition of *r stronger* in *T* would mean that client calls such as [3] can no longer be guaranteed correct treatment: if the dynamic type of *x* is *T*, all that the client guarantees is the original precondition α, which is not enough.
- Making the postcondition *weaker* would mean that the client may no longer rely on the assurance that the original postcondition β will hold after the call.

In other words, *T*, as a **subcontractor**, would be breaching the contract that binds the original contractor *S*, the only one that clients such as *C* know about.

In this discussion "stronger" means "implies" and "weaker" means "implied by". More precisely "a is stronger than b" means (*a* **implies** *b*) **and not** (*a* = *b*), and "weaker" is the inverse relation.

← *"Implication", 5.2, page 84.*

The rule follows from these observations:

Contract Redeclaration rule

The redeclared version of a feature may only: keep or weaken the precondition; keep or strengthen the postcondition.

It is not necessary to retain the contract exactly: weakening the precondition means that the descendant version accepts cases that would have been rejected by the original; strengthening the postcondition, that it delivers a better result — for example a better numerical approximation — than has been promised by the original. Both cases are harmless and, in fact, frequently useful.

As an example of precondition weakening, the routine *take* in class *TAXI* has a precondition clause stating that the customer must be within a certain distance (100 meters) of the current taxi position. In *DISPATCH_TAXI*, this is no longer necessary; the revised and relaxed condition, which the original one implies, is that the customer must be within the geographical area of the taxi.

How can the programming language enforce the contract redeclaration rule? The solution in Eiffel (also taken over by other notations applying Design by Contract ideas) is simple:

- You are **not** permitted to use the basic contract clauses, **require** and **ensure**, in a feature redeclaration.
- If you do not write a contract clause, the original clauses are retained. You will see them in the flat and contract views.
- To weaken an inherited precondition, use a contract clause of the form **require else** *new_pre*. The semantic effect is to equip the redeclared version with the precondition *old_pre* **or else** *new_pre*, where *old_pre* is the inherited precondition.
- To strengthen an inherited postcondition, use a contract clause of the form **ensure then** *new_post*. The semantic effect is to equip the redeclared version with the postcondition *old_post* **and then** *new_post*.

This satisfies the rule since — from the rules of logic — *a* implies *a* **or** *b*, and *a* **and** *b* implies *a*.

← *Exercise "Signs of strength", 5-E.8, page 103.*

The **and then** semantics for postconditions is a simplification of the actual rule, not affecting this discussion. For the full rule see the Eiffel language specification.

The flat and contract views will show the full reconstructed contracts, including inherited clauses.

The precondition of *take* in *DISPATCH_TAXI*, in application of this discussion, reads

```
require else
    in_zone: customer.is_in_zone (Current)
```

You can find many more examples by perusing libraries such as EiffelBase.

Contracts in deferred classes

The Contract Redeclaration rule gives its full meaning to the use of contracts in deferred classes, such as the contracts you may have noted in *forth*. Deferred features do not have an implementation, but they may have preconditions and postconditions; deferred classes, while not fully implemented, may have invariants. This is a big part of what makes the whole concept useful.

← *Page 569*.

When writing a deferred class and its features, you are providing a template with some elements to be filled in by descendants; remember the "Program with Holes" design pattern. While you are letting descendants provide their own implementations, you may and usually should *constrain* what such implementations may do. The contracts enable you to define such basic semantics, which descendants may refine but never contradict.

← *Page 569*.

The example of *forth* was typical:

```
forth
        -- Advance cursor by one position
    require
        in_range: not after
    deferred
    ensure
        increased: index = old index + 1
    end
```

This describes a routine that advances the cursor in a list. *How* it moves the cursor depends on the implementation; this is why the routine is deferred. But whatever a descendant implementation does, it must work correctly for any cursor position that is not *after* (not past the last element), and it must increase the cursor index by 1. Anything else is permitted as long as the implementation meets these conditions.

> As an analogy, think of a stereo system with various outlets where you may plug devices (a tuner, a CD player, speakers etc.). You can choose the device that you plug into each outlet — but only if it satisfies the corresponding electrical requirements. In the same way, *search* lets you "plug in" many possible variants of *forth* and other routines in descendants of *LINEAR*, but only if they satisfy the contracts defined for these features in *LINEAR*.

This gives us a closer grasp of what deferred classes and features are about. They do not just shirk implementation, but define an abstract semantic framework for future implementations. That is also why this notion is so useful in requirements analysis and high-level design: you can use deferred elements to define essential properties, not just of structure, but also of behavior. As the classes get progressively refined in descendants, the details will be filled, but always in accordance with the general framework that you have set at the start.

Contracts tame inheritance

Whether for deferred or effective classes, the rules on contract adaptation are essential to a proper use of inheritance. Polymorphism and dynamic binding are powerful techniques, but a bit scary too: since every type can adapt inherited features, how do you know that a call *my_vehicle*.*turn_left* will not — because of some redefinition in the applicable descendant class — make your vehicle turn right, stop, or make a U-turn? Here the flexibility that the combination of redefinition, polymorphism and dynamic binding brings to programming goes too far. As the designer of the original *turn_left* you want to allow every descendant to provide its own implementation, but you also want to freeze the essential semantic constraints that each must respect.

The Contract Redeclaration rule and associated language mechanisms (**require else** …) give you this control. You can specify, as broadly or narrowly as you wish, the boundaries of their freedom to implement your overall concept.

Inheritance and its associated techniques are not just a powerful form of reuse but a **subcontracting** technique. Classes use *redefinition* to subcontract certain operations to descendants. Because of *polymorphism* and *dynamic binding*, a client does not know which subcontractor will handle a particular call — just as, when you order an iPhone, you do not know which parts are made in Cupertino, Taiwan, Shanghai, Bangalore or Bucharest. The Contract Redeclaration rule could be called the "*Keeping Subcontractors Honest*" rule.

16.10 OVERALL INHERITANCE STRUCTURE

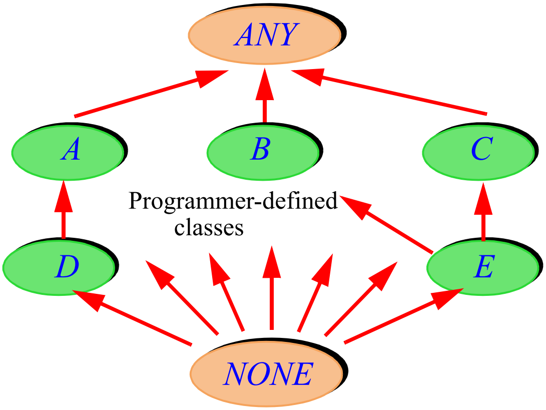
Inheritance allows us to provide a general framework where every software element has a clear place. Most object-oriented languages (a notable exception is C++) define a special class, sometimes called Object (Smalltalk, Java, C#). In Eiffel it is called *ANY* and is a part of the "Kernel Library", which also includes some fundamental classes closely connected with the language definitions (*ARRAY*, *STRING* and basic types such as *BOOLEAN*, *INTEGER*, *CHARACTER*, *REAL*). The overall structure appears in the figure on the facing page.

A, *B* etc. in that figure are any of the classes that you and I may write. The rule defining the role of *ANY* is simple: any such class to which you have not given an **inherit** clause, writing it instead as just

```
class A feature … end
```

is understood as if you had actually written

```
class A inherit ANY feature … end
```



Overall inheritance structure

ensuring the following property:

Universal inheritance and conformance theorem (Eiffel)

Every class is a descendant of *ANY*.
Every type conforms to *ANY*.

ANY is the place where general-purpose features, useful for all types, have their original declarations; they include *is_equal* and other general comparison features; in earlier examples we used its *print* procedure, which prints a default representation of any object. As a type, *ANY* is the most general of all; a variable declared as *v*: *ANY* can be used polymorphically to denote objects of any type.

← *"Traversals", page 453.*

The bottom of the figure shows another predefined class: *NONE*. As *ANY* is an ancestor to all classes, so is *NONE* a descendant of all classes. Unlike *ANY*, a flesh-and-bone class whose text you can bring up in EiffelStudio, revealing many useful features, *NONE* is a convenient fiction, closing off the type structure at the bottom, but without a meaningful class text. It serves two practical purposes:

- As a type, it enables us to give a type to **Void**, the predefined value representing the void reference (a value not attached to any actual object).
- As a class, it supports information hiding: as you know, we declare secret features in a clause starting with **feature** {*NONE*}. This means that they are exported only to *NONE*; so no actual class can use them.

← *"Information hiding: modifying fields", page 240.*

The syntax is a special case of **selective export**: the ability to start a feature clause with **feature** {*C*, *D*, …}, for any classes *C*, *D*, …, meaning that the features whose declarations follow are exported only to *C*, *D*, …, and their descendants.

It would make no sense to export a feature to *C* and not to its descendants, since they need the mechanisms available to *C*, for example to redeclare a feature of *C*. Class *NONE* has no proper descendants, so in this case it makes no difference.

16.11 MULTIPLE INHERITANCE

From the start, this discussion of inheritance has noted that a class may have two or more parents, a case known as *multiple* inheritance. This is an important possibility that can be put to great benefit. Here a warning is necessary if you have encountered some of the naïve literature on object-oriented programming:

> ***Touch of Methodology*:**
> **Dispelling urban legends about multiple inheritance**
>
> Multiple inheritance (while subject to misuse like any other programming construct) is an essential tool for the construction of reliable, extendible, reusable software systems. Do not be misled by blanket dismissals of this technique as difficult or problematic.

(If you are new to the field you do not need this warning — good for you.) The misconception about multiple inheritance comes from poor language design in early O-O approaches, which made multiple inheritance seem messy, and also from the limitations of early implementation techniques. As we will now see none of this is justified today, and once you have discovered the practical use of multiple inheritance you will not be able to live without it.

Using multiple inheritance

Inheritance is specialization: vehicles specialize the notion of a moving object, taxis specialize the notion of a vehicle. Sometimes a notion is a specialization of *two* or more other notions; then multiple inheritance is necessary. Without it you would end up choosing one of the parents as the principal one, and duplicate the code of the other.

To inherit from several classes just list them successively in the **inherit** part, each with its **redefine** and other inheritance-adaptation clauses if any:

```
class TROLLEY inherit
    TRAM
        redefine add_station, remove_station end
    BUS
feature
    …
end
```

A simple example can be found in basic libraries. The class *NUMERIC* covers objects equipped with the standard mathematical operations:

```
deferred class NUMERIC feature
    plus alias "+" (other: NUMERIC): NUMERIC deferred end
    minus alias "–" (other: NUMERIC): NUMERIC deferred end
    times alias "*" (other: NUMERIC): NUMERIC deferred end
    divided alias "/" (other: NUMERIC): NUMERIC deferred end
    …
end
```

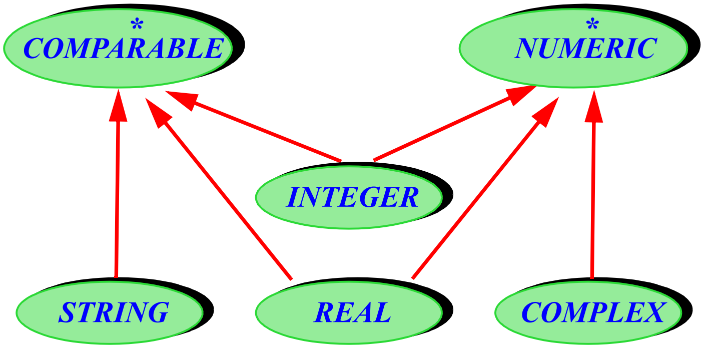
(This is only a sketch; look up the class text in EiffelStudio). Another library class, *COMPARABLE*, covers objects equipped with a total order relation:

← We studied total orders with topological sort: "Total orders", page 514.

```
deferred class COMPARABLE feature
    lesser alias "<" (other: NUMERIC): BOOLEAN deferred end
    lesser_or_equal alias "<=" (other: NUMERIC): BOOLEAN do … end
    greater alias ">" (other: NUMERIC): BOOLEAN do … end
    greater_or_equal alias ">=" (other: NUMERIC): BOOLEAN do … end
    …
end
```

Not all comparable types are numeric: strings have a total order relation — the usual lexicographic order — but no multiplication or division. Not all numeric types are comparable: there is no usable total order on complex numbers or matrices. But some types, such as *INTEGER* and *REAL*, possess both sets of properties; the corresponding classes use multiple inheritance to reflect this.

*"Usable" in the sense of retaining essential properties, such as the **field** structure of complex numbers.*



Multiple inheritance

Java and C# permit multiple inheritance not from classes but from *interfaces* as discussed earlier, similar to entirely deferred classes. The present example illustrates the difference. As highlighted above, class *COMPARABLE* needs only one deferred feature, for example *lesser* (with its alias <); all the others can be defined from it as effective features, for example

← Page 568.

```
lesser_or_equal alias "<=" (other: NUMERIC): BOOLEAN
        -- Is current object less than or equal to other?
    do
        Result := ( Current < other ) or (Current ~ other)
    ensure
        definition: Result = ((Current < other) or (Current ~ other))
    end
```

~ is object equality.

Similarly, *greater* (*other*) is defined as *other*.*lesser* (**Current**) and again *greater_or_equal* in terms of the others. These are not just initial implementations but definitive ones, as expressed by the postcondition of *lesser_or_equal*.

COMPARABLE is an example of the *Program with Holes* pattern: a scheme that leaves some operations open for descendants to implement — here we settled on *lesser*, although any of the four could play this role — and defines the others in terms of them. If you can only choose between an interface, fully deferred, and a class, fully effective, you will not be able to rely on this fundamental software development technique. More precisely, making *COMPARABLE* an interface would mean that every descendant must provide implementations of *all* its features. This may cause:

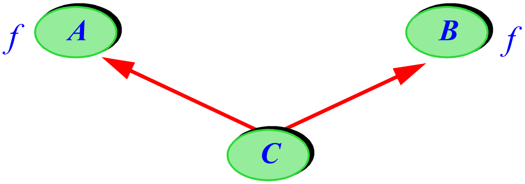
← Page 569.

- Code duplication, since all implementations of *lesser_or_equal*, for example, will be identical, using the code given above.
- The risk of errors, since — except through contracts in a language that supports them — there is no way for the author of the ancestor class to impose specific implementations on authors of descendants.

Just as the distinction between classes and interfaces is artificial, the restriction of multiple inheritance to interfaces is impractical.

Renaming features

Multiple inheritance raises the specter of overloading if a class *C* inherits from two classes with identically named features:



Name clash

This is easy to resolve if we insist on staying away from overloading: never give the same name to two different features in the same class. If you try to compile code corresponding to the above structure

```
class C inherit
    A
    B
end
```

with both *A* and *B* having unrelated features called *f*, the class *C* will not pass compilation. The solution is just as easy:

```
class C inherit
    A rename f as first_f end
    B
end
```

The **rename** clause (which can be combined with redefinition, as in **rename** *f* **as** *first_f* **redefine** *first_f* **end**) simply indicates that the feature known as *f* in *A* will be known as *first_f* in *C*. Here we could of course have renamed the feature from *B*, or both.

The renamed feature is still the same feature — "the feature formerly known as *f* " — in *C*. So the following calls are both valid with *a1*: *A* ; *c1*: *C*:

```
a1.f
c1.first_f
```

but not, of course, with *a1*.*first_f*, since *A* does not have a feature called *first_f*. The call *c1*.*f* is valid but refers to the feature from *B*; if we had also renamed that feature, then the call would be invalid. In a polymorphic situation, after we have assigned *a1* := *c1*, the two calls above would have the same effect since *a1* and *c1* denote the same object, and *f* and *first_f* denote the same feature in the corresponding classes.

The *flat view* of a class and its *contract view* both take renaming into account, as well as redefinition.

Apart from removing name clashes, renaming helps you get your feature terminology right. Sometimes when you inherit features from a parent their names are not well suited to the context of the heir; then you can just adapt them through renaming.

Compare renaming and redeclaration:

Renaming vs redeclaration

Renaming keeps the inherited feature and changes its name.

Redeclaration changes the feature (through redefinition, effecting or undefinition) and keeps its name.

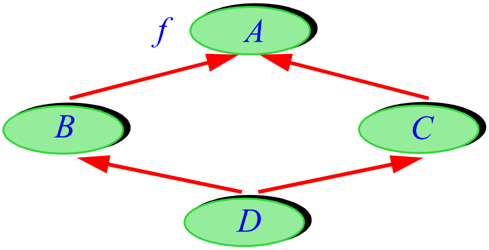
You can combine them when you want to change *both* the feature and its name.

These techniques are central to the use of inheritance as a tool for software architects. As we saw early in this book, to write a class is to build a machine; inheritance is a tool for building machines by extending and specializing existing machines. It is a construction technique more than an interface technique, as clients of a class do not need (except to use it polymorphically) to know that it has been obtained through inheritance rather than built from scratch.

← "Objects as machines", page 28.

From multiple to repeated inheritance

We have one more technical point to examine: the case of *repeated* inheritance, arising when as a result of multiple inheritance more than one path exists from a descendant to an ancestor:



Repeated inheritance

You only need to build such structures for advanced development (you can see a few examples in the EiffelBase library), but should know the rule because repeated inheritance arises in fact any time you use multiple inheritance: because of the structure discussed in the previous section, the common descendant will inherit *ANY* repeatedly.

← As illustrated in "Overall inheritance structure", page 587.

Repeated inheritance raises two questions: the fate of repeatedly inherited features; possible ambiguity through dynamic binding.

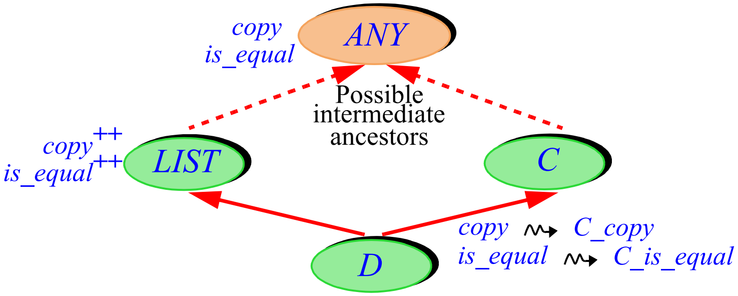
In the illustrated case, the first question arises for *f*, a feature of *A*: in *D*, does it yield one feature, or two? The answer is simple and in line with previous discussions:

- If the feature is inherited from both sides under the same name, for example if it never undergoes any renaming along the inheritance paths — it remains known as *f* throughout — it is clearly the same feature. This is a case of acceptable name clash: even though the parents have a feature with the *same name*, no problem results since they actually denote the *same feature*. It is not really a clash. This case is known as **sharing**.
- If the feature is inherited under two different names — as a result of renaming along the way — then, again to avoid any overloading, it must denote two different features. We talk of feature **replication**.

An obvious validity constraint applies in the sharing case: you need to know what is being shared. If there has been a redeclaration on either path, the version of the feature from the other side must be deferred (either originally, or through **undefine**). If you have two effective features and want to keep both, just rename one of them, getting back to the replication case.

The second question arises precisely in the replication case, when there has been a redefinition. It can occur for example with repeated inheritance from *ANY* if one of the classes along the way has introduced its own notions of copy and equality by redefining *copy* and *is_equal* from *ANY*:

The need for "select"



is_equal and *copy* should always be redefined together, since the postcondition of *copy* (*other*) states *is_equal* (*other*): any copy operation must ensure that the result is equal to the target of the copy, according to the local notion of equality.

LIST redefines *copy* and *is_equal* to ensure that they copy and compare not the list header but the actual contents of the list; most container classes of the EiffelBase library similarly define their own notion of copy and comparison. Now *D* inherits from *LIST* and also from a class *C* which has retained the default versions from *ANY*. All this works according to the previous rules; *D* must rename the features to avoid a name clash, so they are duplicated. The only problem arises under **polymorphism**: for *a* of type *ANY* and *d1* of type *D*, a call executed after a polymorphic assignment as in

```
a := d1
a.copy (…)
```

is potentially ambiguous: should it use the *LIST* version (known as *copy* in the class) or the *C* version (*C_copy*)? This situation arises whenever there is both replication and redefinition. A simple clause, **select**, is required in such cases to remove the ambiguity. Here is how you should write *D*:

```
class D inherit
    LIST [T] select copy, is_equal end
    C rename copy as C_copy, is_equal as C_is_equal end
feature
    … Rest of class text …
end
```

This means that you want to "select" the versions from *LIST* under dynamic binding for a polymorphic target with a possible ambiguity. You will need the clause in any such case; you may of course select from either branch, and you may select some features from a parent and the rest from another, although in the present case it only makes sense to select *copy* and *is_equal* consistently.

16.12 GENERICITY PLUS INHERITANCE

The introduction of inheritance enables us to revisit the other major extendibility technique for classes: genericity. You have seen inheritance and genericity shine separately; in combination they are even more potent.

← *"Static typing and genericity", 13.1, page 363.*

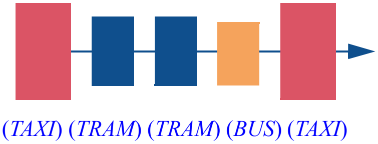
Polymorphic data structures

We have encountered a first way of making inheritance and genericity collaborate: polymorphic data structures. With a container such as

```
fleet: LIST [VEHICLE]
```

you know that the various items can be instances of any of the descendants of *VEHICLE*; this lets you play the game of dynamic binding with list elements as obtained for example through *fleet.item* during a traversal.

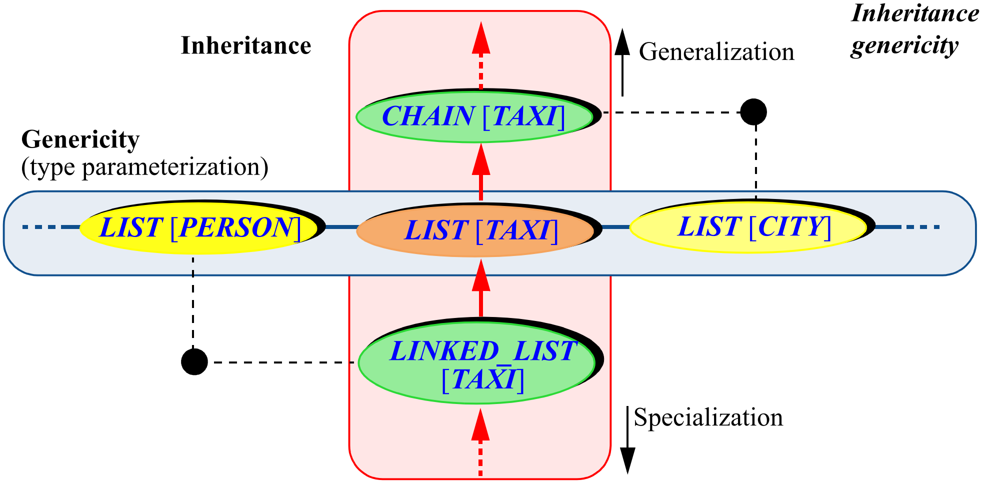
A polymorphic list



(*Figure from page 561*.)

The figure at the top of the facing page offers a graphical view of this kind of inheritance-genericity combination. It introduces no new concept, simply an informal interpretation of what you already know from the concept of

Inheritance and genericity



polymorphic data structure. We start from the basic notion of class, illustrated by a "list of taxis" abstraction in the center, with the features we expect: add an item, remove an item, move the cursor, get the item at cursor position and so on. From there we can vary the concept in two ways, represented by the two directions:

- Besides a list of taxis we may want to talk about lists of persons, cities, or objects of any other types. **Genericity** allows us to travel this horizontal dimension of the figure by providing a *type parameterization* mechanism, hereby avoiding code duplication while guaranteeing type safety as we saw in the original discussion.

- A list is a special kind of "chain" and in turn has more specialized kinds such as the "linked list", functionally a list but resulting from a specific choice of implementation. **Inheritance** allows us to travel this vertical dimension of the figure by providing a *generalization* and *specialization* mechanism, again a powerful form of reuse.

As suggested at the bottom left and top right of the figure, you can combine the two mechanisms freely by going to any point of its metaphorical plane, getting, for example, a linked list of persons or a chain of cities.

More terminology: you may encounter genericity, especially in the literature about "functional languages", under the name *parametric polymorphism*. Don't ask me why. Genericity and (especially in the Java and C# context) its variant *generics* are the more common terms.

Constrained genericity

Polymorphic data structures are not the only way to combine genericity and inheritance. The other important technique follows from exploring the question: *what can we do with an entity or expression of a formal generic type*?

In a generic class such as *LIST* [*G*] or *ARRAY* [*G*] — the text of the classes themselves, not some use such as *LIST* [*VEHICLE*] in a client — we will have declarations such as *x*: *G*; what operations are applicable to *G*?

You can guess the answer from an earlier discussion. Since *G* is just a placeholder for any type, which will be given by the actual generic parameter in a derivation (*VEHICLE* in the example), applicable operations are those that work for all classes: the features of *ANY*. So you may use *x*.*cloned*, *x*.*is_equal* (*y*) and so on, but no features beyond those of *ANY* since they might not always be available depending on the choice of generic derivation.

← *"Overall inheritance structure", 16.10, page 586.*

What if you want more operations? Sometimes this would be really handy. Consider the case of a "sorter" class providing features such as *sort* that order the elements of data structures such as arrays or lists, for example producing an ordered permutation of any list of integers. We want a mechanism that will work for many types, not just integers, so genericity seems appropriate.

Computer science uses "sorting" in the sense of ordering (producing a total order).

Sorting algorithms are a rich area of computer science; we do not study them here, although we have seen a special case, topological sort. But we do not need to study any particular sorting technique to realize that, somewhere in the algorithm, we will need — assuming we are sorting an array *a* — an instruction such as

← *Chapter 15.*

```
x := t [i] ; y := t [j]
if x < y then                                              [4]
            -- Swap the items at positions i and j in a:
     a [i] := y ; a [j] := x
end
```

where *i* and *j* are integers and *a* [*i*] and *a* [*j*] the array items at the corresponding positions. The algorithm will swap their values if it finds they are out of order. Any sorting algorithm defines a strategy for selecting successive *i* and *j* values for comparison and possible swap. That part is not our concern here; the interesting issue is how to make the highlighted comparison work. What "<" comparison operation does it use?

The right one, we want to answer: integer comparison if we are sorting integers, ranking if we are comparing tennis players, and so on. But how do we even know that there is such an operation?

We don't, and sometimes there isn't. As noted when we encountered the library class *COMPARABLE*, no useful total order exists on objects such as complex numbers or matrices.

← *Page 589.*

To provide a general-purpose sorting algorithm applicable to sorting arrays of many types we should put the above code [4] in a generic class

```
class SORTER [G … ] feature
     sort_array (a: ARRAY [G ])
               -- Reorganize elements of a according to an order relation.
          local
               x, y: G
          do
               … Code such as [4], with tests such as x < y  …
          end
     …
end
```

See below the replacement for the "…".

But where does the "<" come from? This is not as if we were using *x* and *y*, which both have type *G* (the formal generic parameter), in operations such as *x*.*cloned* or *x*.*is_equal* (*y*); these operations come from *ANY* and hence are applicable to arbitrary objects. Here we want a comparison operation, available only from specific classes such as *COMPARABLE*.

Classes such as *COMPARABLE*? Why not choose *COMPARABLE* itself; it is deferred, and its effective (directly usable) descendants will provide their versions of *lesser* **alias** "<" and other comparison operations. We may go further and expect that *any* class that represents objects with a total order relation must be a descendant of *COMPARABLE*. Then the answer to our basic genericity question is straightforward: *G*, the formal generic parameter, should no longer represent an arbitrary type but one that conforms to *COMPARABLE*. The following syntax represents this property:

```
class SORTER [G –> COMPARABLE ] feature
     … The rest as above…
```

The symbol –> (hyphen followed by angle bracket) recalls the arrows of inheritance diagrams; the type that follows, here *COMPARABLE*, is the **generic constraint**. The meaning is that a generic derivation *SORTER* [*T*] is valid only if *T* conforms to that constraint. So *SORTER* [*INTEGER*] and *SORTER* [*STRING*] are fine, as well as *SORTER* [*TENNIS_PLAYER*] if *TENNIS_PLAYER* inherits from *COMPARABLE* (effecting *lesser* to compare player ranks), but not *SORTER* [*COMPLEX*], or *SORTER* [*VEHICLE*] if we have not made vehicles comparable. Such attempts are rejected at compile time.

As you probably guessed by now, one may view the basic genericity case *LIST* [*G*], (**unconstrained** genericity) as a shorthand for *LIST* [*G* –> *ANY*]. This gives a formal basis to the above observation that operations applicable to *x* of type *G* in such a class are those of *ANY*, such as *cloned* and *is_equal*.

Constrained genericity has many applications. Some frequently occurring cases use deferred library classes similar to *COMPARABLE*:

- If you define vector or matrix classes, you will want to equip them with features covering addition and other numeric operations. For example it should be possible to compute *m1* + *m2* where *m1* and *m2* are both of type *MATRIX* [*T*]. This requires the ability to compute *t1* + *t2* where *t1* and *t2* are both of type *T*. The solution is to remember class *NUMERIC* and declare the matrix class as *MATRIX* [*G* –> *NUMERIC*]. Then you may use *MATRIX* [*INTEGER*] but not, for example, *MATRIX* [*STRING*]. An interesting twist is to make *MATRIX* itself inherit from *NUMERIC*, which makes sense since it provides all the required operations (indeed the model for *NUMERIC* is the mathematical notion of *ring*); this allows such derivations as *MATRIX* [*MATRIX* [*INTEGER*]], as well as *MATRIX* [*MATRIX* [*MATRIX* [*INTEGER*]]] and so on.

← *Page 589.*

- You may want to allow features of a class *C* [*G*] to store elements of type *G* into a hash table. This assumes that on every such element you can compute an integer *hash function*, a simple requirement but not one that all types necessarily satisfy. Those that do are descendants of the library class *HASHABLE*, and effect its deferred query *hash_code*.

← *"Hash tables", 13.9, page 411.* See *"Definition: Hash function", page 411.*

HASH_TABLE itself is declared as

```
class HASH_TABLE [ELEMENT, KEY –> HASHABLE ] feature …
```

explaining why the generic parameter to *TOPOLOGICAL_SORTER* was also constrained by *HASHABLE*: we wanted to put the elements into a hash table.

← *"Numbering the elements", page 531.*

> *HASH_TABLE* also illustrates in passing that you may have more than one generic parameter — here one constrained, the other not — and name them however you like (calling the first one *G* is a common but not obligatory convention).

A language note: it is possible to specify *multiple* generic constraints. For example you might declare a class

```
class C [G –> {COMPARABLE, NUMERIC, HASHABLE} ] feature …
```

Note the braces in case of multiple constraints.

to specify that an actual generic parameter, to be valid, must conform to *all* the types listed. *INTEGER* satisfies this. The example is extreme, but conforming to both *COMPARABLE* and *NUMERIC* is frequent.

Constrained genericity illustrates the fundamental role of types in modern programming. Other language approaches to the problem discussed here are possible, such as somehow passing a comparison routine to *SORTER*. But it is more consistent with basic object-oriented ideas to make the required functionality part of the type. This also means that we can continue relying on the compiler to perform all necessary validity checks for us, rather than running the risk of a run-time mismatch. If the compiler rejects your class because of a type inconsistency, remember that such seemingly bad news is really good news: better catch a bug before it catches you.

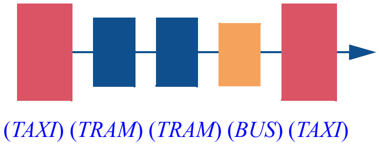
← "Touch of Methodology: Zen and the Art of Reacting to Compiler Messages", page 367.

Never underestimate the power of static typing. This is really a program verification mechanism. The sophistication of the type system that you now master — with classes, constrained and unconstrained genericity including multiple constraints, single, multiple and repeated inheritance, polymorphic variables and data structures, rules on feature calls, argument passing and assignment — defines a framework of mathematical precision, which the compiler uses to perform crucial consistency checks. The types of your program elements closely reflect the semantics of the underlying model of the world.

16.13 UNCOVERING THE ACTUAL TYPE

The discussion of dynamic binding's contribution to software architecture explains why it was not so urgent, in the presentation of polymorphism and polymorphic data structures at the beginning of this chapter, to answer the question "what if I know that the last element of the list is an instance of *TAXI*, not just *VEHICLE*, and want to apply a taxi-specific operation to it?". You are looking at an item of a list that you know as a list of vehicles:

A polymorphic list



(Figure from page 561.)

```
fleet: LIST [VEHICLE]
```

or more generally at a variable or expression of type *VEHICLE*. That is how you know it; you have renounced its specificity — its distinctive identity as a taxi, a tram or a bus — to treat it as just a vehicle. The downside is that you can no longer take advantage of its taxiness or tramness; the advantage is that you can apply any vehicle operation to it without having to know what kind of vehicle it is, *even* if the operation is different for each kind. So if you want to apply a specific operation you should, in the usual case, make it an operation applicable to all vehicles, with a specific implementation for each variant. This is the only way to avoid the Many Explicit Variants Syndrome.

← "Touch of Methodology: Fight the Many Explicit Variants syndrome", page 575.

This reasoning breaks down in two cases:

- The operation you want to apply is really specific to the chosen type, and you cannot introduce it at a higher level in the inheritance hierarchy, if only because the classes in that hierarchy are someone else's classes.
- Your program obtains objects — for example the above list — from an outside source over which it has no control; for example it retrieves them from a file, a database or a network. Then it will only know them as values of the most general kind; indeed the corresponding library features return a result of type *ANY*.

As an important example of the second case, any good O-O environment provides some kind of **serialization** mechanism to write out object structures to files, and retrieve them. The retrieval operation, such as

```
retrieved: ANY
        -- Object retrieved by the last retrieval operation
```

This is the interface you will find in Eiffel serialization library classes such as STORABLE.

can only declare the type of its result as *ANY* because the operation is general-purpose: it should work for any application domain and will return whatever object it finds, be it a taxi, a tram, a city or anything else. When you use it in a particular application and for a particular file, you expect a certain type of object, for example *TAXI*; but then you will not be able to write

```
t := my_serializer.retrieved
```

for *t* of type *TAXI*, since *ANY* does not conform to *TAXI* — it's the other way around! You need somehow to force, or *cast*, the retrieved object reference into a *TAXI* object; to throw away its nondescript identification as an *ANY*, shared with the entire object populace, and reveal its true inner nature, its taxi self. To support this identification process we need more than the inheritance, polymorphism and dynamic binding techniques seen so far.

In looking for a new mechanism, we note that the cast cannot be unconditional. When obtaining an object from a file or a network you can expect that it will be of a certain type, but you cannot be sure. You are no longer dealing with objects entirely within the control of your program, for which a declaration *t*: *TAXI* guaranteed that *t* would always be attached to a *TAXI* instance. For objects coming from the wild world out there, you no longer have such a guarantee. Even an object structure that you serialized yourself — say you saved the *fleet* structure into a file — may have been corrupted or hacked when you read it back. This imposes a rule on the relevant language mechanisms:

Touch of Methodology: Casting Principle

Any mechanism that forces a type upon an object reference without respecting the static rules of conformance must rely on a run-time test of the corresponding object, to ensure that its type conforms to the expectation.

This is a principle for language design (with direct effect on program design).

In other words such a mechanism must be conditional. If you expect a *TAXI* and uncover some unrelated object, the casting attempt should fail.

> A mechanism that does not satisfy this principle is the C language's form of casting (also present in C++, but alongside a more sophisticated construct satisfying the principle): writing (T) e, where T is a type and e a reference, will yield a reference of type T, with the same value as e, disregarding the actual type of the data at the corresponding memory location. This directly reflects machine-level operations that treat references (pointers) as just addresses without a specific type; programmers are supposed to "know what they are doing". Higher-level languages enforce more typing and more checks.

A common term for describing casting mechanisms satisfying the principle is **dynamic cast**; "conditional cast" is also appropriate. This is another area that lacks a standard terminology; other terms that you may encounter are **type narrowing** and **downcasting**, both addressing the case of casting from a general type such as *VEHICLE* to a more specific type such as *TAXI* that conforms to it. (It is more *narrowly* focused, and further *down* in the inheritance hierarchy.) This is the most common case but not the only one; we will see why the mechanism to be studied now, Object Test, can apply a dynamic cast between two arbitrary types.

Yet one more term is "typecast"; see "Derived types", page 808 in the discussion of C++.

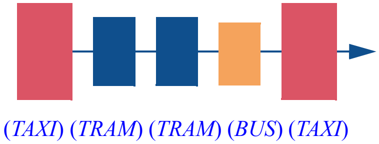
A more general term covering any kind of technique for finding out the types of objects at run time is **RTTI**, for "Run-Time Type Identification".

Sorry for this terminology chop-suey; I did not make up all these grand-sounding terms. You should know that they exist, but what really matters is to understand the underlying concepts and the general solution to be presented now. We will in fact review two dynamic cast mechanisms, one current, the other obsolete but still in use.

The object test

Assume you are convinced the last item of *fleet* is a *TAXI* object — as it is in the figure — and want to apply a taxi-specific feature such as *take* to that element. The simplistic solution will not work:

A polymorphic list



(Figure from page

```
fleet.last.take (…)
```

for reasons that should be clear but which we can analyze carefully by splitting the instruction into two:

```
-- Earlier declarations:
fleet: LIST [VEHICLE]
t: ?            -- Placeholder, should be replaced with an actual type
…
t := fleet.last                                                        [5]
t.take (…)                                                             [6]
```

WARNING: this example and the previous one are invalid.

As suggested by the "?" we cannot properly type *t* (in the sense of giving it a type — my keyboard is fine, thanks) since we are stuck:

- If we declare *t* as a *VEHICLE*, [5] is valid but not [6] since *take* is not a feature of *VEHICLE*.
- If we declare it as a *TAXI*, [6] is valid, but the assignment [5] violates the polymorphism type rule; it goes against the direction of conformance.

The **object test** construct provides the solution in such cases. An object test is a boolean expression that (as a form of RTTI) determines whether an object's type conforms to an expected type, and if so has the supplementary effect of defining a local name to represent the object for a small part of the program text, or "scope". Applying an object test to the example yields:

```
if attached {TAXI} fleet.item as t then
    t.take (…)
    … Any other TAXI operations on t, attached to a TAXI object …
else
    … Do something else (non-taxi) if necessary …
end
```

This one is valid!

(In fact, it is the solution.)

In accordance with the Casting principle the operation is conditional; it tests for the type of *fleet.last* — the actual object attached, at the time of execution, to this reference — and returns false if that type does not conform to the type listed, *TAXI*. In that case the **else** clause of the conditional, if present, will be executed. If the type does match, we have a taxi object and locally make it available through the name listed, *t*, called an **object-test local**. It is as if you had declared a local variable *t* and somehow were able to perform the assignment *t* := *fleet*.*last*, [5] above. Such an assignment is not permitted, but a by-product of the object test is to make *t* denote the original value of *fleet*.*last* throughout the **scope** of the object-test local.

If the object test serves as the condition of an **if**, as here, the scope is the **then** part, where you can use *t* as a *TAXI* variable denoting the value of *fleet*.*last*. Because it is indeed a *TAXI*, both statically as a result of the declaration of *t* and dynamically since you checked that the actual object has the expected type, you can apply *TAXI* features such as *take* without any risk.

The definition of the scope of an object test local expresses where it intuitively makes sense to include such operations safely. The following cases cover all practical needs (if you have anything more complicated just introduce a local variable):

- As just seen, if the object test appears as the condition of an **if**, the scope is the **then** part, including if you combine it with other conditions through **and then**, as in **if** **attached** {*TAXI*} *fleet*.*item* **as** *t* **and then** *v*.*is_moving* **then** …
- If you negate the object test (possibly combined with other conditions through **or else**), as in **if** **not attached** {*TAXI*} *fleet*.*item* **as** *t* **then** …, the scope is now the **else** part.
- Similarly if the object test appears negated in the exit condition — **until** clause — of a loop, again possibly combined with other booleans through **or else**, the scope is the loop body (**loop** clause).

← **and then**, **or else**: *"Semistrict boolean operators", 5.3, page 89*.

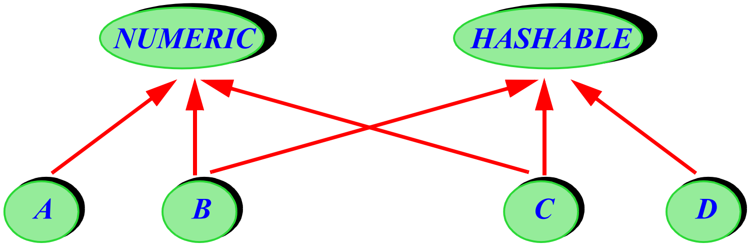
As an example of the last case, here is how to compute the number of objects preceding the first one of a specific type in a possibly polymorphic list:

```
pre_taxi_count (fleet: LIST [VEHICLE]): INTEGER
        -- Number of objects in fleet before the first TAXI.
    do
        from fleet.start until
            fleet.after or else attached {TAXI} fleet.item as t
        loop
                Result := Result +1 ; fleet.forth
        end
    ensure
        non_negative: Result >= 0
        at_most_length_of_list: Result <= fleet.count
    end
```

→ *Also do the exercise "How many taxis?", 16-E.7, page 618*.

This particular algorithm does not use the value of the object-test local *t*; as a consequence you may write the object test as just **attached** {*TAXI*} *fleet*.*item*.

There is no restriction on the types involved in an object test: in **attached** {*U*} *exp* **as** *x*, where *exp* is an expression of (static) type *T*, both *T* and *U* are arbitrary. The most common case is *downcasting* as defined earlier: *U* is a descendant of *T*; the example, with *VEHICLE* and *TAXI*, illustrates it. But this is not a requirement, and with multiple inheritance you may need an object test even for *T* and *U* not directly related by inheritance. A typical case looks like this, referring to classes mentioned in the discussion of multiple inheritance:



Indirect inheritance relationship

NUMERIC and *HASHABLE* share descendants such as *B* and *C*. If you have a list *numlist* of *NUMERIC* elements, you might want to hash any of these elements that is *HASHABLE*:

```
if attached {HASHABLE} numlist.item as h then
    your_hash_table.put (h, h.hash_code )
end
```

(with *numlist* of type *LIST* [*NUMERIC*], and *hash_code* a feature of *HASHABLE*). This is not downcasting, since *HASHABLE* does not conform to *NUMERIC* any more than the other way around, but a legitimate case all the same. It illustrates why we need a general dynamic cast mechanism, not restricted to type narrowing.

Assignment attempt

(This subsection is complementary material, eminently skippable on first reading — but do read the following one, "Using dynamic casts wisely".) Object test supersedes an older mechanism, assignment attempt. Since you may still encounter it in existing code, here is a brief overview. We can go back to the example that showed the inability of basic assignment to provide downcasting:

Warning: illustrates obsolete mechanism.

```
fleet: LIST [VEHICLE]
t: TAXI
…
-- t := fleet.last       -- For comparison only: commented out since invalid  [7]
t ?= fleet.last
t.take (…)              -- Valid but unsafe; see next
```

We declare *t* as a *TAXI*. As before, this makes [7] invalid: an assignment in the wrong conformance direction. Instead of assignment, however, we use assignment attempt, relying on the symbol **?=**, read aloud as "*may receive*", to be compared to "*receives*" for assignment **:=**. The effect is:

← "The power of assignment", page 235.

- If the run-time value of the source (the right side) is of a type conforming to the type of the target, here *TAXI*, to attach the target, here *t*, to that object, as in a regular assignment.
- Otherwise, to make *t* void.

A safe use of assignment attempt should, immediately afterwards and before applying a feature to the target, test whether it is void:

```
t ?= fleet.last
if t /= Void then
    t.take (…)          -- Valid and safe
    … Any other TAXI operations on t, attached to a TAXI object …
else
    … Do something else (non-taxi) if necessary …
end
```

Clearly, you can use assignment attempt for anything you can do with object test. The newer mechanism has the advantage of not encumbering the program with local variables such as *t*: an object-test local plays the same role but appears only at the exact place where you need it. In addition, the use of **Void** to represent failure is unsafe, since it does not force you to test for non-voidness as above: if you forget, you end up with a void call and usually a crash; this cannot happen with object test. As a consequence the language standard for Eiffel, while recognizing the assignment attempt's decades of distinguished service, retired it to introduce object test as its replacement.

Using dynamic casts wisely

Regardless of the actual dynamic cast mechanism, we should not forget the ever looming Multiple Explicit Variants Syndrome. Any kind of dynamic cast makes it possible to implement decision structures of the form "if I have a *TAXI* then do this; else if I have a *TRAM* then do that; else if I have a *BUS* …".

← "Touch of Methodology: Fight the Many Explicit Variants syndrome", page 575.

The details, by the way, have to be handled carefully, because the tests use conformance: if you are discriminating between *MOVING* objects, anything that matches *TRAM* also matches *VEHICLE*, so the order of the tests is significant.

Still, you can do it.

That is clearly not a good idea. The abuse heaped by previous sections on explicit multiple-branch choices, and the rationale for avoiding them, remain as relevant as ever. Dynamic casting is useful not as a competitor to dynamic binding, which beats it hands-down when both are available, but for specific cases of the kind cited earlier: objects coming from the outside — file, database, network — whose type the program must ascertain dynamically before it can use them; and objects accessible through a polymorphic variable or data structure, when as a programmer you know more about their types — or *think* you know more, since you must still include a run-time check — than what the declaration says.

In such cases you will usually be casting to *one* specific type, not testing against a large set of type possibilities. This is a pretty good criterion to use if you are pondering in a particular case whether to use a dynamic cast. If you are expecting a certain object type and ascertaining that the reality conforms to the expectation, you are probably OK. If you are discriminating between a whole range of types, you are almost certainly misusing the mechanism, and should instead consider dynamic binding; or, if you are not at liberty to change affected classes, learn about the Visitor pattern — our next and final topic.

16.14 REVERSING THE STRUCTURE: VISITORS AND AGENTS

The design discipline that we have discovered in this chapter, combining inheritance, deferred classes, polymorphism, redefinition and dynamic binding, all supported by contracts, yields elegant and flexible architectures. But there is still a dark side, which we cannot leave unexplored.

The dirty little secret

The dirty secret (not entirely a secret, since it was mentioned briefly at the start of the discussion) is that our recipe for keeping software architectures stable throughout system changes is biased towards one of the two principal kinds of change. With the techniques seen so far we are very good at guaranteeing smooth evolution when we know the *operations* and must deal with new *types* — but this says nothing about the reverse case!

Now I would not want you to imagine that the first fifty-six pages of this chapter were deceptive advertisement. Experience shows that the scenario we have studied in depth — extending old operations to new types — is the most frequent and delicate one in system evolution. This is where polymorphism and dynamic binding shine; object technology owns much of its success to these powerful ideas.

But the reverse scenario does occur as well, and we cannot ignore it.

Assume for example that for the benefit of some application you want to enrich Traffic objects of many different kinds with a facility for *flashing* their visual representation a few times. You may want to flash specific objects, or all objects in a list. Can you add this to the software with the same information hiding benefits we have seen — in particular, without testing individual objects for their specific types?

Let us call **target classes** the classes to which we want to add this facility (in this case, Traffic classes such as *TAXI*) and **client classes** the application classes that need to apply the new operation to target objects.

→ Target classes usually belong to the core of the application and are also called "model classes". See "The model and the view", page 675.

In favorable cases the techniques we have studied still succeed:

- If the target classes all inherit from a common ancestor, you could add the facility at that level, then redeclare it as appropriate in descendants.
- If they have no common ancestor, you could add one, say a class *FLASHABLE*, making all relevant target classes inherit (through multiple inheritance) from it.

This works, but does not scale up well if the need arises again for new operations. Besides flashing you might want to *rotate* Traffic objects (assuming this is not already possible); later on, to *raise* them above others; and so on. With multiple inheritance you can add *ROTATABLE*, *RAISABLE* and such, but this explosion of little classes does not look right.

→ "Many Little Wrappers" pattern, see page 619 in the next chapter.

As another example, important in practice, consider a development environment such as EiffelStudio or Eclipse, where the fundamental structure is the abstract syntax tree (AST), covered by target classes such as *INSTRUCTION*, *EXPRESSION*, *LOOP*. These classes possess a number of essential features, but a new client tool that comes along — a program formatter, a program analysis tool intended to find potential errors, an HTML generator ... — may need to apply a new operation to every node in an AST. If the operation is really fundamental the corresponding routine should be added to the AST classes, but this should happen only rarely. EiffelStudio addresses this issue through the Visitor techniques discussed next.

In some cases the option of modifying target classes is not available anyway, for example if they are in a library under someone else's control; even if you have access to the source code it makes no sense to modify it since the next release will invalidate your changes. Then the previous solutions do not help.

There are two possibilities in such a case:

- The Visitor pattern.
- Using the agent mechanism.

An outline of these techniques follows.

The Visitor pattern

The Visitor pattern is an architectural technique enabling you to define arbitrary facilities applicable to instances of existing classes.

→ For more about the concept of pattern see "About design patterns", page 678.

The idea is very simple: **let the operations know about the types**. If we are to apply various operations to various types, either each operation must know about every applicable type, or the other way around. With dynamic binding, each *type* knows about the applicable operations. Now we are concerned with the other case: a client class needs the ability to perform an operation on instances of many possible classes (the target classes); we may call such objects — such as taxis and trams in our staple example— *target objects*, or just *targets*.

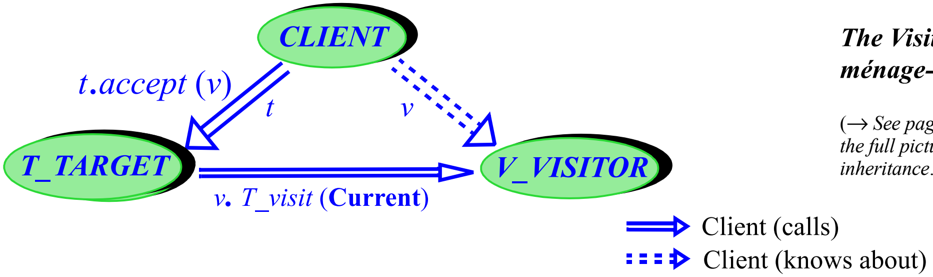
The issue is not how to define the operations. We have to assume that appropriate algorithms are available to write features such as

```
flash_taxi (t: TAXI ) do … Algorithm for flashing a taxi …end                [8]
```

and so on (*flash_tram*, *flash_bus*…). The question is about **architecture**: where do the features belong, and how can we use them in a way that preserves the extendibility of the software?

The features are not in the target classes: if they were, dynamic binding would be the solution. We are assuming this is not the case; that is why they must get their targets through arguments, such as *t*: *TAXI* above. But there is no reason to require *client* classes to implement the features either: better collect the features in a separate class which knows how to perform a specific operation, say flash, on various possible targets such as taxis and trams.

So our *pas-de-deux* between the client and the target turns into a *ménage-à-trois* between client, target and **visitor**. A visitor is an object able to apply a *single* kind of operation to *many* kinds of object; this is the reverse of classes designed for dynamic binding, although — as testimony to the power of the basic O-O ideas — visitors are still implemented as classes and the scheme still crucially relies on dynamic binding.



The Visitor ménage-à-trois

(→ See page 611 for the full picture with inheritance.)

For a target type *T*, such as *TAXI*, and an operation *V*, such as *flash*, the figure shows the interplay between:

- The target class, *T_TARGET*, representing target objects of type *T*. Class *TAXI* is a typical example.

- The visitor class *V_VISITOR*, for example *FLASH_VISITOR*, representing application of the chosen operation to objects of many different types.

- The client class, representing an application element that needs to perform the operation on target objects of various types.

Often the client class will need to perform the operation on a *set* of target objects, for example flash all Traffic objects in a list. This explains the term *visitor*: a visitor object — an instance of a class such as *FLASH_VISITOR* — enables the client to "visit" every element of a certain structure, each time performing the appropriate version of a specified operation. As you know the process of performing such visits is called "iteration" or "traversal".

← *"Definition: Iterating", page 397*; *"Traversals", page 453*.

As always in discussing software architecture for extendibility and reusability, it is important to examine who knows what, and also who does **not** need to know what, with the aim of reducing the amount of knowledge that is spread over the structure and would cause trouble when the information changes. Here:

- The target class *knows* about a specific type, such as *TAXI*, and also (since for example *TAXI* inherits from *VEHICLE* and *VEHICLE* from *MOVING*) its context in a type hierarchy. It *does not know* about new operations requested from the outside, such as flashing.

- The visitor class *knows* all about a given operation, and provides the appropriate variants for a range of relevant types, denoting the corresponding objects through arguments: this is where we will find routines such as *flash_bus*, *flash_tram*, *flash_taxi*. It *does not know* anything about clients.

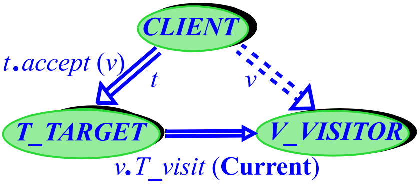
- The client class needs to apply a given operation to objects of specified types, so it must *know* these types (only their existence, *not* their other properties) and the operation (only its existence and applicability to the given types, *not* the specific algorithms in each case).

Using the Visitor pattern, the client will be able to apply the operation, for example, to all items of applicable types in a list, without knowing these types individually, as in

```
flash_all (fl: LIST [ TARGET ]  -- See below about TARGET
        -- Flash all items in fl.
    do
        from fl.start until fl.after loop
            -- "Flash fl.item"
            fl.forth
        end
    end
```

← *The line partially in red is pseudocode; see "Definition: Pseudocode", page 108. The expansion of the pseudocode comes next.*

Visitor classes



(*Figure from page 608*.)

The Visitor pattern provides an implementation of the line in pseudocode. That implementation is very simple (follow it in the figure): the basic visit operation is

```
t.accept (v)
```

for a target object *t* and a visitor object *v*, leading us to replace the pseudocode line by

```
fl.item.accept (flasher)
```

where *flasher* is the *FLASH_VISITOR* object.

All target classes must provide a feature *accept*, whose general implementation is

```
accept (v: VISITOR)
        -- Apply the relevant visit operation from v to x.
    do
        v. T_visit  (Current)
    end
```

T_visit is a visitor feature that implements the requested operation for the type *T*: for example *bus_visit*, *tram_visit*. These procedures must be provided on the visitor side:

```
bus_visit ( t: BUS  ) do flash_bus (t) end
tram_visit ( t: TRAM ) do flash_tram (t) end
taxi_visit ( t: TAXI ) do flash_taxi (t) end
… and so on …
```

It is generally possible to avoid wrapping existing routines in this way, and instead directly implement *flash_bus* under the name *bus_visit* etc.

Admire the delicately choreographed duet in which the target and the visitor engage once the client has set them in motion. The target object knows about its own type; it does not know the requested operation, but knows someone who knows: the visitor *v* passed by the client as the argument to *accept*. So it calls *T_visit* on the visitor, where *T* identifies the target type, and passes itself — **Current** — as argument. This enables the visitor to use the right operation, identified by the inclusion of *T* in the routine name, to the right object, identified by the argument to that routine.

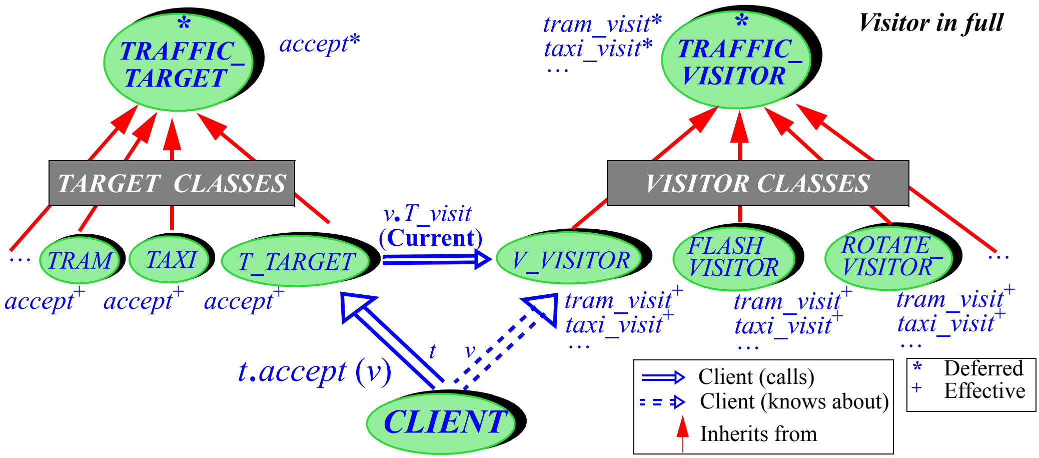
Even though the Visitor pattern is intended to remedy limitations of dynamic binding, it fundamentally relies on dynamic binding. Twice, in fact:

D1 In the client's call *t*.*accept* (*v*), to select the right target.

D2 In the target's call *v*.*T_visit* (**Current**), to select the right operation (by selecting the right visitor).

Dynamic binding is also known as dynamic *dispatch*, and more specifically as **single dispatch** since it dispatches a call to the suitable algorithm on the basis of a single criterion (the type of the call's target). The Visitor pattern is an example of **double dispatch**: selecting an action on the basis of *two* criteria, here a kind of object and a kind of operation. It illustrates a possible technique for achieving double dispatch in a framework that supports single dispatch, as most object-oriented languages do: use single dispatch *twice*. The first call, D1, taking as its *target* the target object *t*, performs a dispatch on the first criterion by applying dynamic binding to that object; it includes the other operand, the visitor *v*, as its *argument*. This allows the routine, here *accept*, to perform the second dispatch through a second call, D2, which uses this argument as its own target. The routine *T_visit* of that final call must still have access to the original target object; this is achieved by passing **Current** as the argument.

For the first case of dynamic binding, D1, to work, all target classes must have a feature *accept*, each redeclaring it in the form *v*.*T_visit* (**Current**) as above.



Visitor in full

As illustrated by the figure, *accept* will come from a common ancestor, where it is deferred. That ancestor is class *TARGET*, which can be very simple:

Appearing as TRAFFIC_TARGET; see explanation below.

```
note
    description: "Objects that can be used as targets in the Visitor pattern"
deferred class TARGET feature
    accept (v: VISITOR )
            -- Make v perform one “visit” operation on the current object
            --| Note: typical implementation is v.T_visit (Current)
            --| where T is the specific effective descendant type.
        deferred
        end
end
```

The last two lines of the header comment rely on a standard convention: starting with --| rather than just -- will cause the contract view not to display them. This is appropriate for comments that describe properties of the implementation, not relevant for clients.

← “What characterizes a metro line”, page 53.

Having to make all target classes inherit from a special *TARGET* class is disappointing, since the original idea was to reuse target classes as they are. With the technique as seen so far, if you have no say at all on target classes, you are stuck. This is the principal limitation of the Visitor pattern; to remove it, we will need to go to completely different techniques, as previewed below. In many practical cases, though, it is not as bad as it sounds: the aim is to avoid modifying target classes again and again, every time a new kind of visitation need pops up. Here you must only make sure that the target classes have **one** ancestor with one specific feature; then for the rest of their lives you can add visitors to you heart’s content.

On the visitor side too you need a common ancestor, say *VISITOR*, and necessary to make *accept* valid in *TARGET*. Here it is not a problem to require such a common ancestor, since you will need to write specific visitor classes for every case of applying the Visitor pattern.

TRAFFIC_VISITOR in the figure, see next.

The figure on the previous page extends our earlier one by including all relevant classes and inheritance relationships. The names of the deferred classes *TARGET* and *VISITOR* now start with *TRAFFIC_*, to be replaced by any name identifying the application when you use the Visitor pattern; this is because with the techniques seen so far it is really impossible to define these classes as reusable components with full generality. *VISITOR* in particular must know all the target types in the application so that it can list, even in deferred form, the relevant features, here *tram_visit*, *taxi_visit* and such.

As before, *T* and *V* stand for prototypical examples of a target type and an operation, complemented here by concrete examples such as *TRAM* and *FLASH*.

This completes the presentation of the Visitor pattern; I hope that you understand it thoroughly, not just the technique but its precise goals, scope, principles, advantages and limitations, as well as how to apply it in practice.

Improving on Visitor

The Visitor design pattern is a popular and useful technique, but suffers from the two limitations noted:

- To ensure visitability, all the target classes must descend from a common ancestor. This is not realistic if they are owned by someone else who has not prepared them accordingly.
- The pattern is not a *reusable* solution: it must be programmed anew for each use, with code that is very similar-looking in all cases; in particular, all implementations of *accept* in target classes follow the same scheme.

It is possible to improve on this basic scheme by using genericity. Beyond this, however, a full satisfactory solution relies on the mechanism studied in the next chapter: **agents**.

→ *"Generic visitor", 16-E.8, page 618*

The agent-based solution is easy to use: for each applicable target type *T*, write a *visit* routine that implements the requested operation for *T*. This does not require modifying any existing class or adding any class. Then, to apply the operation variants to many different targets, pass both the target and **agent** *visit*, an object representing the operation, to a general mechanism that applies an agent to an object without having to know anything specific about either. An exercise in the agent chapter asks you to pursue this solution further. The "pattern library" developed at ETH provides a reusable visitor solution implementing this approach.

→ *"Visiting with agents", 17-E.7, page 660.*

16.15 FURTHER READING

Bertrand Meyer: *Object-Oriented Software Construction* (Second edition, Prentice Hall, 1997).

Contains a detailed analysis of inheritance over several chapters.

Bertrand Meyer and Karine Arnout: *Componentization*: *the Visitor Example*, in *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30. Also available at se.ethz.ch/~meyer/publications/computer/visitor.pdf.

The Visitor pattern complements dynamic binding by making it easy to add an operation to a set of existing types (rather than the reverse). This article presents the pattern and proposes a reusable component that provides it, part of the ETH "pattern library". You will find many other descriptions of the Visitor pattern in the literature (including in Wikipedia); most of them derive their descriptions from E. Gamma et al., *Design Patterns*, Addison-Wesley, 1994.

16.16 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Inheritance enables a class (heir) to obtain the features and invariant clauses of another (parent); instances of the heir can be handled in the same way as instances of the parent.
- Conformance generalizes to types the descendant relation between classes.
- Together with genericity, inheritance helps define a sophisticated type system which turns the compiler into a proof tool enforcing advanced consistency properties of software systems.
- Polymorphism, applicable only to references, enables an expression declared of a certain "static" type to denote objects of various types (its "dynamic" types) at run time. The type system guarantees that the dynamic types are all descendants of the static type.
- A form of polymorphism, relying on genericity, allows the definition of container structures which will at run time be populated by objects of different types.
- Dynamic binding guarantees that in the presence of polymorphism any feature call always uses the feature version best adapted to the dynamic type of the target.
- Polymorphism and dynamic binding advance information hiding by letting clients ignore the exact type of objects they handle and the corresponding operation versions whenever that information is not necessary.
- Typing rules constrain polymorphism: the type of any object to which a variable may become attached at run time (the variable's "dynamic" type) must conform to its declared ("static") type. This ensures that in any assignment or argument passing the type of the source conforms to the type of the target.
- Deferred features have a specification, including a signature and a possible contract, but no implementation. A class is deferred if it contains at least one deferred feature, although it may have others that are effective (non-deferred). Deferred features capture high-level abstractions and are particularly useful for devising proper taxonomies. Deferred classes may not be instantiated.
- A class may redefine an inherited feature to provide a different implementation that overrides the (already effective) version defined in the parent. This combines reuse with adaptation to a new context.
- Genericity and inheritance are complementary mechanisms for type extension. Genericity provides type parameterization; inheritance provides generalization and specialization.

- "Constrained" genericity makes it possible to apply specific operations to variables of a formal generic type, by requiring that all the corresponding generic parameters conform to a given type, the generic constraint.
- Multiple inheritance lets a class benefit from the combination of several abstractions. It is a simple and effective technique; to avoid ambiguity, any conflict in feature names ("name clash") should be removed at the point of inheritance.
- Repeated inheritance arises when, as a result of multiple inheritance, a class is a descendant of another through more than one path. Repeatedly inherited features are merged if inherited under a single name, and kept separate otherwise. Any potential ambiguity under dynamic binding is resolved through a "select" specification.
- For smooth software evolution, the software's architecture should minimize the amount of *knowledge* that each part of a system possesses about the rest.
- Dynamic binding provides an excellent solution to the software evolution case of new types with their own variants of old operations.
- For the case of new operations on old types, the Visitor pattern is a widely applicable solution; it relies on "visitor" objects acting as middlemen between client classes and target classes. A more general, fully reusable solution is also possible, based on the agent mechanism.

New vocabulary

Do not be scared by the length of the following list; it is due in part to terminology variations between authors and between programming languages. The number of actual concepts is smaller; for example "dynamic dispatch" means the same as "dynamic binding". Make sure, however, that you understand all the concepts (exercise 16-E.1).

Ancestor
Assignment attempt
Cast
Client (of a visit)
Conformance
Constrained genericity
Deferred feature, class, type
Descendant
Double dispatch
Downcasting
Dynamic binding
Dynamic cast
Dynamic dispatch
Effect, effective, effecting
Flat view
Generic constraint
Immediate feature
Inheritance
Inherited feature
Interface (Java, C#)
Introduce (a feature)
"Is-a" relation
Multiple inheritance
Name clash
Object test
Object-Test local
Overriding
Parametric polymorphism
Polymorphic expression
Polymorphic data structure
Polymorphism
Precursor
Programs with Holes pattern

Proper ancestor, descendant **Redeclaration** **Redefinition**
Refinement **Repeated inheritance** **Replication (of a feature)**
Routine table **RTTI (Run-Time Type Identification)**
Single dispatch **Subclass** **Subcontracting**
Superclass **Taxonomy** **Target (of a visit)**
Type narrowing **Unconstrained genericity** **Virtual table** **Visitor**

16-E EXERCISES

16-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

16-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← Exercise "Concept map", 13-E.2, page 434.

16-E.3 Abstract syntax

← See also exercise 14-E.4, page 501.

This is a preparatory exercise for the next two (writing an interpreter and a compiler). The aim is to build programs using abstract syntax to avoid having to write a parser (a task that can be carried out simply, but using techniques that have not been covered).

We consider a programming language **L0** with the following constructs, expressed here informally with concrete syntax:

- The only data type is integer.
- A variable (that is, an integer variable) has a name, which is an arbitrary non-empty string.
- A constant is a negative, zero or positive integer.
- An expression is one of: a variable; a constant; an operator expression.
- An operator (all operators are binary) is one of: $+$, $-$, $*$.
- An operator expression is made of two expressions connected by an operator, with the expected semantics (addition, subtraction, multiplication).
- An instruction is one of: skip, read, compound (a sequence of instructions), assignment, conditional and loop).
- A skip instruction, in concrete syntax **skip**, has no effect.
- A read instruction, **read** x, names a variable x; its execution consists of reading an integer input from an interactive user and assigning it to the named variable, here x.
- An assignment is of the form $x := e$, where x is a variable and e an expression.
- An integer value can be used in a test (for conditional and loop) with the convention that 0 means false and any other value means true.

- A conditional is of the form **if** *e* **then** *i1* **else** *i2* **end** where *e* is an expression (understood as boolean as just described); *i1* and *i2* are instructions.
- A loop is of the form **from** *i1* **until** *e* **loop** *i2* **end**
- A compound is a sequence of instructions; in concrete syntax they are preceded by **do** and followed by **end**.
- A program is a compound.

1. Using this concrete syntax, write a program in **L0** to read two integers from a user and compute their greatest common divisor, apply Euclid's algorithm and not using multiplication. (You may write other **L0** programs as examples for the next questions.)

2. Design a set of classes — such as *PROGRAM*, *COMPOUND* and so on — that make it possible to represent the abstract syntax of an **L0** program. Use inheritance as appropriate. Make sure to include the appropriate creation procedures so that programs can be created (purely as object structures, with no concrete syntax).

3. Write a program that produces an abstract syntax tree representing the greatest common divisor program (and any other example you may have prepared) from question 1.

16-E.4 Unparsing

(This is a continuation of the last exercise. Its task, producing concrete from abstract syntax, is the reverse of parsing and is known as *unparsing*.)

← See also exercise 14-E.4, page 501.

Write a program that prints out a concrete representation, with the concrete syntax described in the previous exercise, of an **L0** program given as an instance of class *PROGRAM* and associated objects (instances of *COMPOUND* etc.). Every instruction should start on a single line; a compound nested in another, the branches of a conditional, the initialization and body of a loop should be indented.

Check the output of your program on the examples, such as greatest common divisor, from the previous exercises.

16-E.5 An interpreter operating on abstract syntax

(This is a continuation of the last four exercises.) Write an interpreter, that is to say, an Eiffel program that can execute any **L0** program given as an instance of class *PROGRAM* and associated objects (instances of *COMPOUND* etc.). The semantics of **L0** is that execution of an **L0** program consists of:

- Executing its associated compound.

- Printing out, each on a line, a sequence of pairs; the first element of each pair is the name of a variable used in the program, and the second element is the variable's value at the end of execution. The order of pairs is not important, but every variable used in the program must appear in exactly one pair.

Try out your interpreter by executing it on the example **L0** programs from the previous exercises and checking the results.

16-E.6 A compiler operating on abstract syntax

(This is a continuation of the previous exercises.) Write a compiler that translates any **L0** program into an Eiffel system, in the form of a root class and a set of auxiliary classes as needed.

Try out your compiler by executing it on the **L0** programs from the previous exercises, compiling the resulting Eiffel system, executing it, and checking the results. Check in particular that the results are the same as with the interpreter of the previous exercise.

16-E.7 How many taxis?

For this exercise you can use as inspiration the function *pre_taxi_count*. Consider *fleet*: *LIST* [*VEHICLE*].

← *Page 603.*

1. Write a function that computes the number of vehicles in the list that are instances of *TAXI*.

2. Write a function that computes the number of vehicles in the list that are *direct* instances of *TAXI*.

16-E.8 Generic visitor

Show how to improve the Visitor pattern by representing the target class as a generic parameter of the *VISITOR* class. Make sure to spell out the solution at the same level of detail as the discussion in this chapter. Do you still need a multitude of *VISITOR* variants? Is the solution fully reusable? Discuss its advantages and limitations as compared to the basic Visitor pattern.

17

Operations as objects: agents and lambda calculus

The object-oriented framework has already given us a set of powerful mechanisms to write our programs. In this chapter we again extend our powers of expression, adding mechanisms that let us abstract operations and pass them around for later operations.

17.1 BEYOND THE DUALITY

The extension will require treating *operations* as if they were *objects*, which appears at first to contradict the basic duality, so far taken for granted, between these two notions:

- Programs manipulate objects.
- They do so by applying operations to these objects.

The textual structure of our O-O programs also relies on this distinction: we divide programs into classes, each based on a type of objects, and each operation is attached, in the form of a routine, to one of these classes.

The two notions seem clearly distinct: what the program can do (operations); what it can do it to (objects).

And yet it is sometimes interesting to treat an operation as an object or, more precisely, to define objects whose sole role is to describe an operation. We call such objects *agents*. This chapter studies them in detail, but it is not hard to get the basic idea. You can obtain a simple agent through the notation

```
agent r
```

This is an expression; its value is an agent representing the routine *r*. Because it is an expression, you can assign it to a variable, as in

```
a := agent r
```

with *a* of the appropriate type.

What can you do with an agent? Well, it is associated with a routine or other feature, so one of its uses is to call that feature. With *a* denoting an agent after the above assignment, the call

```
a.call ([x, y])                                                    [1]
```

will have the same effect as if you directly called

```
r (x, y)                                                           [2]
```

for any applicable *x* and *y*. The feature *call* is applicable to all agents; it takes a single *tuple*, here [*x*, *y*], as argument. A tuple is simply an object representing a sequence of values (as I am sure you remember, but if not you should refresh your memory before proceeding with this chapter).

← *"Tuples", 13.5, page 389.*

Why use *call* on an agent, as in [1], when you could just call the routine directly as in [2]? Indeed if you know which routine *r* you want to call there is no point in going through an agent. But now assume you got *a* from another program element, for example as an argument to the current routine. Then all you know is that *a* denotes a routine (and, as we will see, what kind of arguments that routine takes); but you do not know the routine itself — the original *r*.

This is indeed what agents give us: the ability to build and dispatch objects representing operations ready to be executed, with a complete separation between:

- Agent *definition*: the place in the software that defines an agent around a routine *r*, through **agent** *r*, and of course must know about *r*.
- Agent *call*: any place in the software that receives an agent *a* and can apply features such as *call* to it, without knowing what routine it carries.

This mechanism has many different applications, of which we will now explore some of the most important:

- *Iteration*: providing a general mechanism that applies an arbitrary operation to every item of a data structure.
- *Numerical programming*, as when computing the integral of a function over an interval; we may represent the function as an agent.
- Equipping an interactive application with an *undo-redo* mechanism.

Another area where agents play an important role is *event-driven design*, also known as Publish-Subscribe, particularly useful for graphical user interfaces; it is the subject of the next chapter.

We will compare agents with other techniques, based on previously studied mechanisms such as dynamic binding, which would also be available to address some of these applications; we will take a peek at the mathematical basis, the fascinating theory of *lambda calculus*; and we will examine some techniques available in languages other than Eiffel.

17.2 WHY OBJECTIFY OPERATIONS?

It is good first to understand why we need some kind of mechanism to treat operations as objects, and what we would do if we did not have it. Here are four examples: iteration, integration, observation, undoing.

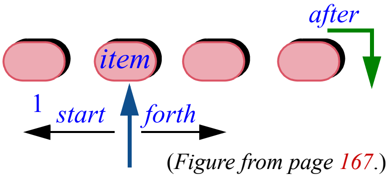
Four applications of agents

First, **iteration**. We have become used, in our loops, to schemes that apply a certain *operation* to every element of a sequential structure such as a list. They look like this:

```
from start until after loop
    "Apply action to item"
    forth
end                                                    [3]
```

← *The second line is pseudocode (see page 108.)*

(where *start* brings the cursor to the first element, *forth* moves it one position forward, *after* says whether it is past all items, and *item* yields the item at cursor position).



(*Figure from page 167.*)

In Traffic, we can apply this scheme to an instance of *ROUTE*, denoting an itinerary with a number of stops. We might want to print the names of all stops in order; to compute the total travel time (by adding the times from each stop to the next one); to produce a list of restaurants along the route (from a list of nearby restaurants, available for each stop); and so on. In each case the solution will look like [3]. We have a name for such schemes: iteration, already encountered in the discussion of data structures. You can use such an iteration scheme, for any given *action*, to produce a routine that applies *action* to every stop along a route.

Now assume that you do *not* want to write a new routine each time you need this scheme. Can we go up one notch on the abstraction scale and simply write something like [3] in a routine where *action* is not hardwired any more, but just an argument? Then we could use that routine with different actions, and let it take care of the looping.

Iteration mechanisms will indeed enable us to provide routines such as *do_all* which you can call with an agent argument representing the action:

```
your_route.do_all (agent action)
```

The second example is from numerical mathematics: **integration**. Given a function *f* (*x*: *REAL*): *REAL* defined over an interval [*a*, *b*], algorithms exist (we will see the basic one below) to compute a good approximation of the integral of *f* over that interval:

$$\int_a^b f\,(x)\,dx$$

The problem here is: is it possible to define a general integration mechanism — say a feature *integral* from a reusable class *INTEGRATOR* — that we can apply to any existing routine *f* representing the mathematical function? Agents will allow us to provide such a mechanism, and call it as

```
your_integrator.integral (agent f , a, b)
```

with *your_integrator* of type *INTEGRATOR*.

The third example is a preview of the next chapter: event-driven design. Assume some part of the system can trigger certain events and other parts need to execute some operations whenever such an event occurs. An example event is a clock tick, happening whenever a set time has elapsed; then a clock display module must update an image, another module needs to update the total time count, and so on. Each such "subscriber" module needs to register a certain action to be executed whenever such an event occurs. We will devise an architecture enabling subscribers to achieve this simply through

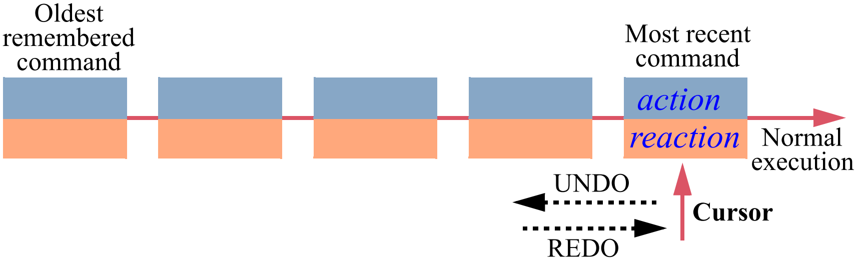
```
clock_tick.subscribe (agent some_routine)
```

where *clock_tick* represents the event type and *subscribe* is a general-purpose library feature. Such subscribers are said to "observe" the event type.

The last example corresponds to a functionality widely needed in interactive systems: **undoing** an operation. Although not widely acknowledged — I have never seen a statue of its inventor — the humble CTRL-Z or equivalent is one of the milestones in the history of humankind, accounting for our need to be saved from our own messing up. Even when we do not actually mess up, we like to try out ideas, see what happens, and backpedal if the result is not to our liking.

The only really good undo-redo mechanism is one that lets you undo and redo not just the last operation but many. You probably do not need much convincing as a user of existing software, but now think of how you would *write* a program with built-in undo-redo to any level.

The most radical technique involves representing all undoable-redoable actions as objects which you can put into a data structure, say *history*, which can be implemented as a list of agent pairs:



A history list

Each action comes from a routine. In this solution the system never executes such a routine, say *r*, directly; instead, it calls

```
execute (agent r , agent r_inverse)
```

where *execute* performs *call* (the mechanism for calling the routine associated with an agent, as previewed above) on its first argument, but also appends the object pair of its two arguments into the history list. Each pair in the history list contains two agents, one representing an action and the other — appearing as *reaction* in the figure — representing the reverse action; this assumes that for every routine *r* implementing a user command you also provide a routine *r_inverse* that cancels the action (otherwise you could not offer an undo-redo mechanism). Then if the user requests one or more "undo", you perform

```
history.item.reaction.call ([])
history.back
```

Assuming no arguments, hence the empty argument tuple [].

as many times as needed, but of course not going further back than the first item. For a "redo" request after one or more "undo", perform

```
history.forth
history.item.action.call ([])
```

not going further than the last item.

A world without agents

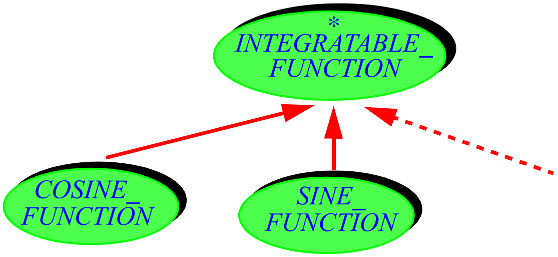
We cannot really understand agents in depth unless we ask ourselves how we would address the above problems if we did not have a special mechanism.

Can we find a solution at all? Of course we can. If what you need is an object wrapper around an action, it suffices to create that object yourself, devising the appropriate class. That will be the hurdle: defining new classes. Let us see the idea at work in the previous examples.

Integration is typical. To define the integration function *integral*, give it an argument of type *INTEGRATABLE_FUNCTION*. That would be a deferred class, looking like this:

```
note
    description: "Functions that can be integrated over finite intervals"
deferred class INTEGRATABLE_FUNCTION feature
    item (x: REAL): REAL
            -- Function’s value for x.
        deferred
        end
end
```

You may devise a more sophisticated form of this class — for example by adding a query *defined* (*x*: *REAL*) and using it in a precondition for *item* — but this simple version suffices to understand the architectural issue.



Classes for mathematical functions

For each particular function to be integrated you must write, as the figure suggests, a little effective class such as *COSINE_FUNCTION* which provides the desired implementation of *item*:

```
        -- In class COSINE_FUNCTION
item (x: REAL): REAL
        -- Function’s value for x.
    do
        Result := cosine (x)
    end
```

← cosine and any other math routine must be obtained, e.g. through inheritance, from some math library class.

Then to obtain the integral of the cosine function over [*a*, *b*], you declare a variable *f*: *INTEGRATABLE_FUNCTION*, make sure it is dynamically attached to an object of type *COSINE_FUNCTION*, and call

```
your_integrator.integral (f, a, b)                                    [4]
```

The function *integral* (*f*: *INTEGRATABLE_FUNCTION*; *x*, *y*: *REAL*) is easy to write, using any algorithm for approximating integrals; whenever it needs to evaluate the value of the function at a certain point *x*, it uses *f*.*item* (*x*). Note the role of dynamic binding: the run-time type of *f*, such as *COSINE_FUNCTION*, determines which *item* feature this call will use. To make sure that you understand this scheme and review your understanding of fundamental O-O techniques it is a good idea to try to write *integral* yourself:

→ *There is one in "Agents for numerical programming", 17.4, page 634.*

Programming time!

An integration library without agents

Write a class *INTEGRATION* with a feature *integral* that computes the integral, over a finite interval, of a function passed as an argument of type *INTEGRATABLE_FUNCTION*. Devising appropriate descendants of *INTEGRATABLE_FUNCTION*, apply your work to the computation of integrals of various sample functions.

For a simple integration algorithm, you may use the model of the agent-based version given below.

→ *"Agents for numerical programming", 17.4, page 634.*

This example is typical of how to take advantage of standard O-O techniques that you can use if you do not have agents. The ideas are easily transposed to all our other examples:

- For iteration, the deferred class will be *ITERATABLE_ACTION*; effective descendants provide specific versions of a procedure *call* describing one execution of the iterated operation.
- For observation (event-driven programming), the deferred class is *OBSERVER*; specific observer classes inherit from it and provide their own versions of the *update* procedure which publishers will call when triggering an event. What this describes is the exact principle of a well-known *design pattern*, Observer, discussed in the next chapter.
- For Undo-Redo, every command of the interactive system must be implemented by a command class providing two procedures: *execute* to perform the command, and *cancel* to undo the effect of the last *execute*. An instance of this class describes information resulting from one execution of the command, and necessary to undo it later should this be requested; for example, in a text editor, an instance of *LINE_DELETION* has two fields, the content of the line being deleted and the position of that line in the text, so that the *cancel* procedure can re-insert a line deleted by *execute*. All such command classes inherit from a deferred class *COMMAND* where *execute* and *cancel* are deferred. The history list can then be implemented as, for example, a *LINKED_LIST* [*COMMAND*]. This is the scheme for another classic design pattern, "Command".

→ *"The observer pattern", 18.4, page 678.*

We may call this technique the "**Many Little Wrappers**" design pattern because it uses dynamic binding based on writing a class, typically small, to wrap each variant of an operation.

The pattern works, but it has the obvious disadvantage suggested by its name: bloating the software with numerous small classes. There is nothing wrong in principle with small classes, but a class should embody a significant abstraction, and having just one significant feature (such as *item* in the integration case) makes it suspicious. This suspicion is reinforced by the observation that in two of the examples (integration and iteration) the classes have only one significant feature (*item*, *call*). In particular, they have no attributes, and hence each needs only one instance (such as the instance of *COSINE_FUNCTION* attached to *f* in [4]). A class with just one instance is known as a **singleton**, but here the objects are not only single, they also have no fields — strange objects indeed. Each of the classes is essentially there to encapsulate a single routine. We may call them **One-Song-Artist** classes.

Having to write many such wrappers complicates the software, in particular its inheritance structure, as we will see for the Observer pattern in the next chapter. In the end, it is frustrating that we cannot directly use the *cosine* function for integration, or a routine *print_stop_name* for route traversal. Why do we need a class wrapper? In an extreme case, the same mathematical operation could conceivably be amenable to integration *and* iteration *and* observation; we would wrap it in three different ways!

Among our examples, the one case where wrapping does not come out as too artificial or bothersome is undo-redo, because the *COMMAND* abstraction seems warranted: it has two equally important routines, *execute* and *cancel*, so it is at least a two-song artist; and descendant classes describe meaningful objects, with many different instances (for example every execution of a *LINE_DELETION* yields a new instance) characterized by meaningful fields (such as the specific content and position of the deleted line).

In the other cases, it seems hard to justify the Many Little Wrappers technique. We do need to wrap individual routines into objects, but we would prefer not to do the wrapping ourselves. Better rely on a language mechanism.

Agents are that mechanism. If *f* is a routine, you can get it gift-wrapped for free by just writing **agent** *f*; this gives you an object that has everything about *f*, including the ability to call *f* (through *call*), for any applicable arguments, whenever you need to. In all the cases cited and many others, agents and related techniques are superior to the Many Little Wrappers pattern. So I hope you did not mind this little digression — about what to do *without* agents — since it gives us a better appreciation for the benefits of a simple, built-in mechanism to deal with actions through objects.

→ On related techniques see "Other language constructs", 17.8, page 654.

17.3 AGENTS FOR ITERATION

Now that we see where agents fit and why we need them, we can go beyond the earlier overview and see the full details. The present section completes the iterator example; the following one deals with integration. The next chapter has a detailed agent-based solution to the problem of event observation.

Basic iterating schemes

A simple example from Traffic illustrates the use and definition of iterators through agents. Consider the notion of *ROUTE*. We can add to *ROUTE* a routine *do_at_every_stop* that takes an action as argument and applies it to every stop. This will make it possible to use

```
your_route.do_at_every_stop (agent print_stop_name)                [5]
your_route.do_at_every_stop (agent append_restaurants)
…
your_route.do_at_every_stop (agent other_operation)
```

This simply assumes that *print_stop_name*, *append_restaurants* and *other_operation* are routines — specifically, procedures — all taking a *STOP* as argument.

How will *do_at_every_stop* make all these uses possible? It abstracts the standard iteration scheme cited earlier in this chapter [3]:

← *Page 621.*

```
do_at_every_stop ( action :…)
        -- Apply action to every stop in this route.
    do
        from start until after loop
            action.call ([item])                                   [6]
            forth
        end
end
```

→ *The type for action appears below (see page 629).*

To trigger the associated routine, this uses *call*, a procedure available on all agents, whose effect is to call the agent's routine, with the arguments given; more precisely, given as a single *tuple*, in this case [*item*]. As you know, a sequence of values in square brackets is a manifest tuple; since *action* is intended to represent routines such as *print_stop_name* that take one argument, the tuple used here, [*item*], has just one element.

← *"Tuples", 13.5, page 389.*

The effect of the highlighted call *action.call* ([*item*]) [6] is exactly the same as that of a direct call to the corresponding routine, such as

```
print_stop_name (item)                                             [7]
```

if the argument passed to *do_at_every_stop* was **agent** *print_stop_name*, or

```
append_restaurants (item)                                              [8]
```

if the argument was **agent** *append_restaurants*, and so on. The difference with [6] is that within *do_at_every_stop* we do not know what actual routine *action* represents, so we cannot use a direct call such as [7] or [8]; we need an agent, represented here by *action*.

This technique is the basic mechanism for providing iterators in the EiffelBase library; we will study the actual library implementation below.

→ *"Writing an iterator", page 631.*

Iterating for predicate calculus

An interesting application of iteration is to give us a direct implementation of the predicate calculus mechanisms: for all (∀), exists (∃). Assume that you want to state that all items of an array of integers *a*, of bounds *a*.*lower* and *a*.*upper*, are positive. In predicate calculus we learned to express this as

← *"Predicate calculus", 5.4, page 94.*

$\forall\ s: a.lower\ ..\ a.upper \mid\ a\ [i] > 0$ **[9]**

where *i*..*j* denotes the interval containing all values between *i* and *j* inclusive. Without agents you can use *all_positive* (*a*) if you write a function *all_positive* (*ia*: *ARRAY* [*INTEGER*]) which determines the result through a loop. But thanks to agents you do not need to write such a routine; just use

```
(a.lower |..| a.upper).for_all (agent is_positive)                     [10]
```

which is very close to [9]. |..| is the operator alias of a function *interval* from class *INTEGER*, which yields a result of type *INTEGER_INTERVAL*, a library class, providing functions *for_all* and *exists* which take as argument an agent representing the test being "for-alled" or "existed" across the interval.

These particular *for_all* and *exists* are for arrays, but we will soon see similar functions applicable to lists and other sequential structures.

> [10] still requires you to write a small function to test whether an integer is positive: *is_positive* (*n*: *INTEGER*): *BOOLEAN*. This is more reasonable than requiring something like *all_positive* for every such case. At the end of this chapter we will learn how to get rid of even *is_positive* by writing the needed agent *inline*, without having to introduce an explicit routine.

→ *"Inline agents", 17.7, page 652.*

Exploration time!

You may wish to take a quick look now at the functions *for_all* and *exists* in *INTEGER_INTERVAL*. Do not get stuck with the type declarations (they are explained next), but make sure you understand the algorithms. Also see *exists1*.

Agent types

In the declaration of *do_at_every_stop* we need to fill in the type of *action*, representing an agent. The actual declaration will be:

← *The type was left unfilled, as "…", in [6].*

```
do_at_every_stop (action: PROCEDURE [ANY, TUPLE [G]]
    … The rest as before [6] …
```

PROCEDURE is a generic library class describing command (procedure) agents. It takes two generic parameters, representing type properties of the procedure *p* associated with the agent:

- The first denotes the class from which *p* comes, or an ancestor of that class. Since *ANY* is ancestor to all classes you can usually use *ANY*, as we do here with *do_at_every_stop*, since in an actual argument **agent** *p* corresponding to *action* we do not care what class *p* comes from.
- The second parameter is always a tuple type. The tuple component types correspond to the types of the arguments to *p*.

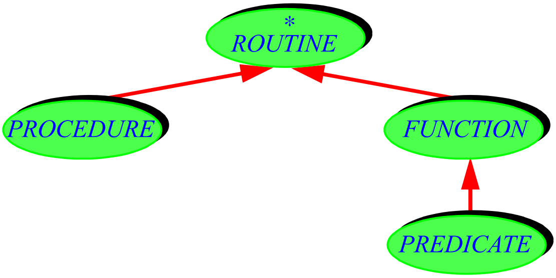
Here *p* will be a procedure, such as *print_stop_name* and *append_restaurants*, taking one argument of type *G* (the generic parameter of *LINEAR* and descendants, also serving as the type for *item* and representing the type of the items in the data structure). As a consequence, the second generic parameter of *PROCEDURE* must be *TUPLE* [*G*].

It is this choice of *TUPLE* [*G*] as the second parameter type that enables the body of *do_at_every_stop* to call the associated routine *p*, whatever it is, with valid arguments, through the following line from [6]:

```
action.call ([item])                                    [11]
```

Indeed, procedure *call* is declared in *PROCEDURE* (check the class text in the library!) as taking an argument of type *OPEN*, the second generic parameter.

PROCEDURE describes agents associated with commands. It is part of a hierarchy of four classes in the Kernel Library:



Agent classes

FUNCTION is for agents denoting queries, except that queries returning a result of type *BOOLEAN* are covered by *PREDICATE*. *FUNCTION* has a third generic parameter, representing the type of the function's result. Class *ROUTINE*, deferred class, covers all agent variants.

Here are the headers of the classes involved:

```
deferred class ROUTINE [BASE, OPEN –> TUPLE]

class PROCEDURE [BASE, OPEN –> TUPLE] inherit
    ROUTINE [BASE, OPEN]

class FUNCTION [BASE, OPEN –> TUPLE, RES] inherit
    ROUTINE [BASE, OPEN]

class PREDICATE [BASE, OPEN –> TUPLE] inherit
    FUNCTION [BASE, OPEN, BOOLEAN]
```

In the actual class texts, the formal generic matters have longer names BASE_TYPE, OPEN_ARGS and RESULT_TYPE to avoid conflicts with programmer-chosen class names.

The second parameter, *OPEN*, is constrained by *TUPLE*, so you can only use a tuple type — *TUPLE* [*G*] in the above example — as actual parameter. Given a feature *f*, the expression **agent** *f* is of a type derived from one of the above, depending on the nature of *f*: procedure, boolean query, other query.

← *"Constrained genericity", page 596.*

For an agent representing procedures with two arguments of types *T* and *U*, use

```
PROCEDURE [C, TUPLE [T, U]]  -- C is often just ANY
```

and similarly for the other cases. For a routine with no arguments, the second actual parameter will be just *TUPLE*, as in *PROCEDURE* [*ANY*, *TUPLE*].

Class *ROUTINE* declares:

```
call (v: OPEN)
    -- Call feature with all its operands, using v for the open operands.
```

allowing you to call the agent by passing an appropriate tuple as illustrated above [11]. If there are no arguments — the actual parameter for *OPEN* was just *TUPLE* — you will pass to *call* an empty tuple [].

In addition, *FUNCTION* and *PREDICATE* include the feature

```
last_result: RES
    -- Function result returned by last call to call, if any
```

and, for convenience, the function *item* combining *call* and *last_result*:

```
item (v: like open_operands): RES
        -- Result of calling feature with all operands, using v for open operands.
        -- (Will call call.)
    ensure
        set_by_call: Result = last_result
```

→ *In cases so far the "open operands" are the arguments. For more, see "Open operands", 17.5, page 636.*

so that *f.item* ([*x*]), for a function agent *f*, yields the result of calling the associated function on the argument *x*.

A home for fundamental iterators

Class *LINEAR* in EiffelBase, ancestor to all the list classes such as *LIST*, *LINKED_LIST* and others, describes any structure that can be traversed linearly. As such, it is the natural home for a set of iterator features:

- *do_all* applies a certain action in turn to all item of the structure, like our *do_at_every_stop* example.
- *do_if* applies it to all elements that satisfy a certain condition; there are also *do_while* and *do_until*.
- *for_all* tests whether a certain property (again represented by an agent) holds of all elements of a structure, and *exists* tests whether it holds of at least one.

The arguments are:

- In the first two categories, *action* representing the action to be applied, of type *PROCEDURE* [*ANY*, *TUPLE* [*G*]].
- In the last two categories, *test* representing a boolean-valued query, of type *PREDICATE* [*ANY*, *TUPLE* [*G*]].

(*do_if*, *do_while* and *do_until* have both arguments.) As an example of use, assume a certain class has an integer attribute *sum* and the procedure

```
increase_sum (n: INTEGER)
        -- Add n to sum.
    do sum := sum + n ensure added: sum = old sum + n end
```

Then given a list *il*: *LIST* [*INTEGER*] the call

```
il.do_all (agent increase_sum)
```

will (after *sum* := 0) assign to *sum* the total of the values of *il*'s items.

Writing an iterator

Of course I do not expect you to be content with the knowledge of how to *use* iterators such as *do_all*; we must learn how to *write* them. So we move to the internal picture.

Anatomy Lesson

As part of our regular series of examining *real* code in depth, we now take a look at the text of *do_all* in class *LINEAR* [*G*] from EiffelBase. You are encouraged afterwards to explore its companions such as *do_if* and *for_all*.

Here is the routine's text, copy-pasted from the library:

Version 6.3.

Procedure do_all in class LINEAR

```
do_all (action: PROCEDURE [ANY, TUPLE [G]])
            -- Apply action to every item.
            -- Semantics not guaranteed if action changes the structure;
            -- In such a case, apply iterator to clone of structure instead.
        local
            c: CURSOR
        do
            c := cursor
            from                        -- Core loop
                start
            until
                after
            loop
                action.call ([item])
                forth
            end
            go_to (c)
end
```

We can first ignore the cursor-related instructions and focus on the signature, comment and the highlighted part marked "Core loop".

Procedure *do_all* takes a single argument, *action*, representing the agent to be iterated. Its type *PROCEDURE* [*ANY*, *TUPLE* [*G*]] indicates that the agent's associated procedure can come from an arbitrary class (as expressed by *ANY*) and (as expressed by *TUPLE* [*G*]) should take one argument of type *G*, the formal generic parameter of the enclosing class.

→ Exercise "An iterator that shoots itself in the foot", 17-E.5, page 660.

The header comment tells us "Semantics not guaranteed if *action* changes the structure". The warning is important: havoc could result if *action* changes the data structure itself, for example by deleting an element. (An exercise asks you to try this for yourself if you have the nerve.) It is OK for *action* to change the *contents* of objects in the structure; for example you can safely use *do_all* to add one to every element of a list of integers, through

```
do_all (agent increment)
```

In this example both increment and the call to do_all are assumed to be in a descendant of LIST [INTEGER].

with

```
increment do active.put (x + 1) end
```

since this does not modify the structure. If you *do* want to modify it, the last line of *do_all's* header comment indicates that it is safe to iterate on a clone (duplicate) of the original structure; then *action* can modify the original without affecting the clone. For a clone of a structure *s*, simply use *s.cloned*.

> The requirement stated by this header comment is legitimate: the notion of iterating on a data structure stops making sense is the structure itself changes as you are iterating on it. Still, as you may have reflected, it is regrettable that to enforce it we have to resort to exhortation through a header comment, rather than expressing it in the contract for the routine, in the form of a precondition on *action*. Contracts are currently not expressive enough to state such properties.

The "Core loop", the heart of the algorithm, is the same scheme we already saw in a special case [6]:

```
from start until after loop
    action.call ([item])                                    [12]
    forth
end
```

With this, you understand the essential properties of *do_all* and similar operators.

Since we are studying software exactly as it is in the library, it is useful to go a bit further than usual into the implementation details, with a comment on performance and a clarification of what the algorithm does with the cursor.

First, although you might not immediately guess it, a major compiler optimization is essential to the success of the above scheme, specifically the instruction marked [12]. A manifest tuple expression such as [*item*] represents a tuple object. Once it has been passed to *call*, the object is no longer needed; but a naïve implementation would create a new object every time. This is bad for performance; not so much a space issue, as the garbage collector will eventually reclaim the unneeded objects, but a time penalty, as object creation is expensive. To avoid this penalty we should — like a regular coffee drinker who, instead of using a paper cup each time, decides to get a good, sturdy cup once and for all — create a single tuple object and reuse it throughout.

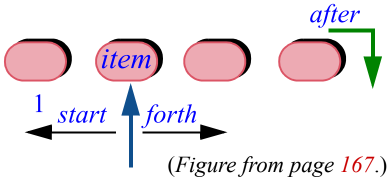
You can program this optimization explicitly: declare a tuple variable *t*, create the tuple at the beginning, then on each iteration fill *t* with *item* and pass it — rather than the manifest tuple [*item*] — as argument to *call*.

→ Do this as an exercise: "Manual optimization", 17-E.6, page 660.

This is not something you need to worry about, however, as the compiler does the optimization for you:

Touch of Optimization:
Reusing a tuple

In schemes such as [12] for *do_all*, requiring many tuples each used a single time, the EiffelStudio compiler generates code that creates a single tuple and refills it repeatedly.



(*Figure from page 167.*)

The last detail is the variable *c*. Its purpose is to ensure that the iterator leaves the structure in the state where it found it.

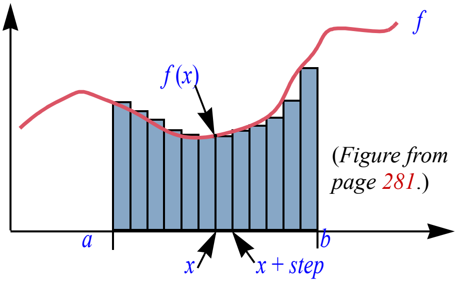
A *LINEAR* structure has a cursor; the iteration in *do_all* moves it using *start* and *forth*. But other parts of the software may also work on the list and, having moved the cursor to some position, may expect to find it later in that same position. Iterators such as *do_all* must be good citizens: they can move the cursor while they operate, but they must restore it when they are done

It suffices to use a variable *c* of type *CURSOR*; an instance of *CURSOR* is an abstract description of a cursor position, independent of the representation. The query *cursor* returns such an object, denoting the current position. At the beginning, we record it into *c*; at the end, the command *go_to* (*c*) moves the cursor back to the position represented by *c*.

An instance of *CURSOR* is an "**external cursor**", representing a position in a traversable structure but, unlike the "internal cursor", kept as a separate object rather than stored in the structure itself. Here we use an external cursor to remember the initial position of the internal cursor and restore it later.

← *Internal and external cursors were briefly discussed in the introductory part of "Lists", 13.6, page 391.*

17.4 AGENTS FOR NUMERICAL PROGRAMMING



(*Figure from page 281.*)

As iterators illustrate, agents allow us to describe operations that manipulate other operations. Such needs frequently arise in *numerical* programming; integration is a typical example.

The standard numerical technique to integrate a real function *f* over a finite interval *a*..*b* is — as previewed in an earlier chapter — to approximate the exact integral $\int_a^b f(x)\,dx$ by the sum of the areas of many small rectangles. If all these rectangles have width *step*, the one starting at abscissa *x* has area *step*∗ *f* (*x*). An approximation of the integral over the interval is the sum $\sum_{i=0}^{n-1} f(a + i \times step) \times step$ of these areas for all *x* such that $a \le x < b$; the number *n* of rectangles is approximately $(b - a) / step$.

What agents give us here is the possibility of writing an integration function *integral* not just for a specific function *f* — say the cosine function — but for any applicable function.

Here is an implementation of *integral*:

```
integral (f: FUNCTION [ANY, TUPLE [REAL], REAL] ; a, b: REAL): REAL
        -- Approximation of integral of f over interval a..b.
    local
        x: REAL ; i: INTEGER
    do
        from x := a until x >= b loop
            Result := Result + f.item ([x])
            i := i + 1; x := a + i * step
        end
    end
```

The declaration of *f* indicates that it must be a function that takes an argument of type *REAL* and yields a result of type *REAL*. To evaluate *f* at point *x* we just use *f.item* ([*x*]); we have seen that *item*, for a function agent, calls the function through *call* and returns the result of the call. This feature expects a tuple as argument, hence the square brackets around *x*.

> On the numerical side, note the computation of *x* from scratch on every iteration of the loop, rather than by successive additions which would cause the error to accumulate.

← *"Computing with numbers", page 279.*

Function *integral* most likely belongs in a class *INTEGRATOR* describing objects in charge of performing integration operations on mathematical functions. If you adopt such a design, and *your_integrator* is of the corresponding type, then you will obtain the integral of a function *f* over an interval *a..b* as the value of

```
your_integrator.integral (agent f, a, b)
```

Class *INTEGRATOR* is also where *step* should be declared as a *REAL* attribute, with an associated setter procedure so that clients can control the precision of the integration process. The class is more than a mere "wrapper" for *integral*; it describes a meaningful abstraction, "integration control".

17.5 OPEN OPERANDS

Sometimes you will need a bit more flexibility in using agents, to set some arguments once and for all in the agent's definition and leave others to be filled separately by each call. We talk of "*closed*" and "*open*" arguments.

Open arguments

As a simple example consider a variant of the last scheme where you still want to compute a function's integral, but the function has extra arguments:

$$\int_a^b g\ (u, x, v)\ dx$$

Variables u and v retain constant values during the integration — only the x axis is involved, as before — but they are still needed to evaluate g for every value of x. You could rely on the previous solution by obtaining the integral as

```
your_integrator.integral (agent g_extended , x, y)                    [13]
```

after defining an auxiliary function

```
g_extended (x: REAL): REAL
        -- Same as g but with first and second arguments set to u and v.
    do
        Result := g (u, x, v)
    end
```

assuming that u and v are attributes. This works; but it is tedious to write such auxiliary functions, especially if the pattern recurs. It will be even more unpleasant if u and v are local variables or formal arguments.

A function such as *g_extended* is just a wrapper, whose only purpose is to freeze some of the arguments of a function, turning it into a function of the remaining arguments only. This is needed so often that a special notation is appropriate. You can obtain the same effect as [13], without writing a wrapper function, through the expression

```
your_integrator.integral (agent g (u, ? , v), x, y)
```

The agent expression **agent** *g* (*u*, **?**, *v*) denotes:

> The one-argument function obtained from the three-argument function g by freezing its first and third arguments, to the values u and v respectively, and retaining only as a true argument the one at the second position, marked "**?**".

More generally, in an agent expression you may use an argument list, corresponding to the signature of the underlying function, but in this list you may replace any of the arguments by a question mark **?**. These are known as **open** arguments to the agent, and the others — the ones given by normal values, like *u* and *v* above — as **closed**. The agent represents a function of the open arguments only.

This means in particular that our first agent notation, **agent** *f*, is just an abbreviation for

```
agent f (?, ?, …)
```

with *all* arguments open. Of course in this case it is just as simple to use the shorter notation **agent** *f*.

The notion of open argument increases the versatility of agents, saving the need for many auxiliary routines such as *g_extended*. As another example, let us vary a bit the earlier iteration scheme involving a list *il* of integers, assuming a routine

← Page 631.

```
increase_sum_by_power (m, n: INTEGER)
          -- Add to sum the value of m to the power n.
     do sum := sum + m ^ n ensure added: sum = old sum + m ^ n end
```

with *sum* now a *REAL* since that is the type the power operator ^ returns. Then, after executing *sum* := 0.0, we can assign to *sum* the sum of the squares of all the elements of *il* through

```
il.do_all (agent increase_sum_by_power ( ? , 2)
```

To summarize:

Definition: Open and closed operand

An operand of an agent is **closed** if it is specified in the agent's *definition*.

An operand is **open** if it will be provided only in *calls* to the agent. In the agent definition, it is marked with a question mark "**?**".

Terminology reminder: the "definition" of an agent is the expression that specifies it, such as **agent** *f* (*a*, **?**); a "call" to an agent is an instruction (involving a call to *call*) or expression (involving a call to *item* for a *FUNCTION* agent) that calls its associated routine at run time.

← From page 620.

← Page 631.

Open targets

The last box sneakily introduced a new term, *operand*. So far we had been looking at open *arguments*. Why another notion? The reason is that sometimes what you need to keep open includes not just arguments but the **target** of a call.

Consider again the earlier example involving routes and their stops:

```
your_route.do_all (agent print_stop_name)                                [14]
```

This is identical to [5] except that we no longer need the ad hoc procedure *do_at_every_stop*; we directly use *do_all*, since Traffic's *ROUTE* class is actually a descendant of *LINEAR* [*STOP*]. The example assumes a procedure *print_stop_name* of signature

```
print_stop_name (s: STOP)
```

so that **agent** *print_stop_name*, appearing in a class *C*, has the type *PROCEDURE* [*C*, *TUPLE* [*STOP*]], matching the type for the formal argument of *do_all*.

The procedure *print_stop_name* looks at instances of *STOP* from the outside: it does not belong to this class but takes an argument of type *STOP*. This argument is the one that remains open since [14] is really an abbreviation for

```
your_route.do_all (agent print_stop_name (?))                            [15]
```

Whenever *do_all* issues a *call* to the agent — we saw where this occurs: *action*.*call* ([*item*]) for every *item* representing a *STOP* in the route — has the same effect as a direct call to the associated routine, of the form

← *"*Core loop*" of do_all, page 632.*

```
print_stop_name (your_route.item)                                        [16]
```

where the highlighting emphasizes that the iterated action, corresponding to the **?** in the agent, is passed as argument. The agent-based iteration scheme [14] is equivalent to a loop that would explicitly iterate through *your_route*, initializing the iteration through *your_route*.*start*, advancing it through *your_route*.*forth* and executing the call [16] at every step.

Like any call in object-oriented programming, this call has a target, but here the target is implicit: the current object. (We can always make it explicit by writing the call as **Current**.*print_stop_name* (*your_route*.*item*).)

← *"Definitions: Qualified and unqualified call", page 134.*

Assume now, however, that the operation we want to iterate is no longer such an outside action but one given by a feature of class *STOP* itself. For example class *STOP* may have a feature *close* to mark the current stop inoperative. If we want to close an entire line, we should iterate *close* over all its stops; but *close* takes no argument, since it is called just on its target, as in the typical call

```
some_stop.close
```

so that the action to be iterated, replacing [16], is

```
your_route.item.close
```

In this case it is the routine's target, not one of its arguments, that we want to keep open in the argument to *do_all*, replacing **agent** *print_stop_name* (**?**) in [15] (or the short form [14]).

At first we might think of writing that argument as something like **?**.*close*. But this would not work since it misses the type of the target; many classes may have a feature called *close*. We must specify the target type; the valid form for the example, illustrating the notation for open targets, is

```
your_route.do_all (agent {STOP}.close)                                  [17]
```

Now you see the need for the term *operand*: it covers all the values needed to execute a call — target and arguments.

For open arguments a plain "**?**" will generally do, since the types follow from the routine's signature — for example in [14] and [15] we know that *print_stop_name* takes one argument of type *STOP* —, but the form {*TYPE*}, listing an explicit type, is also permitted. (It can be useful if you want to specify a *TYPE* other than the argument's declared type, although it must conform to it.)

All combinations of open and closed operands are valid (assuming *f* with arguments and *g* without arguments in a class *C*):

- Everything closed: **agent** *f* (*x*, *y*, *z*), **agent** *g*.
- Target closed, all arguments (if any) open, as in **agent** *g* (no arguments) and **agent** *f* (**?**, **?**, **?**). As we saw, the latter can also be written just **agent** *f*.

agent *g fits both of the first two cases.*

- Target closed, some arguments open, some closed: **agent** *f* (**?**, *y*, **?**).
- Target open, some or all arguments closed: **agent** {*C*}.*f* (**?**, *y*, **?**), **agent** {*C*}.*f* (*x*, *y*, *z*).
- Everything open: **agent** {*C*}.*f* (abbreviation for **agent** {*C*}.*f* (**?**, **?**, **?**)), **agent** {*C*}.*g*.

These mechanisms are what allows us to use a **single set of iterators** — *do_all*, *do_if*, *for_all* and others — in *LINEAR* and all its descendants. Without them, we would need *two sets* of iterators: one to iterate operations that work on their target, and the other to iterate operations that work on their argument.

17.6 LAMBDA CALCULUS

We have now seen the basics of agents (and quite a few details as well); I hope you appreciate the power of expression of this mechanism and are already thinking of all kinds of wondrous applications.

There is actually one more level of flexibility, but before we get there allow me to take you through a tour of the underlying mathematical ideas. To understand what agents really are about — in particular, to get a deeper understanding of the concepts of "open" and "closed" operands as just studied — you should know the basics of *lambda calculus*.

It is a beautiful theory, developed in the 1930s before there were any computers; the discovery thirty years later that it provides a clear basis for many of the concepts of programming languages led to a revival of interest, and lambda calculus remains a fertile area of research.

Lambda calculus gives us a theory of the notion of *function*, reduced to its essence: not any particular kind of function, such as the functions of trigonometry or real analysis with their specific properties, but the very idea of a function as a mechanism that takes arguments and yields a result. This is the *mathematical* notion of function, but since it underlies the concept of routine in computer science the theory will give us new insights directly relevant to programmers: what is the scope of a variable, what is the role of an argument, and how can we treat a routine as if it were an object — the very goal that this chapter pursues by wrapping routines into agents.

Operations on functions

The basic idea is simple: a notation and transformation rules allowing us to play with functions as we play with other mathematical objects.

Given two numbers a and b, you can write combinations like $a + b$ or $sin\ (a) + cos\ (b)$; these use functions with well-defined signatures, for example

```
sin: REAL → REAL          -- Meaning: For any argument of type REAL,
                          -- sin yields a result of type real
"+": [REAL × REAL] → REAL
                          -- × is cartesian product; brackets are for grouping
```

Can we play similar games with functions? Even in elementary mathematics we do find operators on functions: if f and g are functions with appropriate signatures, their composition, written $g \circ f$ or sometimes $f\ ;\ g$ (the notation we will use, because it retains the order of application), is the function h such that $h\ (x) = g\ (f\ (x))$ for any applicable argument x. This makes ";" an operation on functions, the way "+" is an operation on real numbers.

Lambda calculus will enable us to define many operators such as ";", whose arguments are functions.

We can continue up the ladder of abstraction. Composition, ";", can itself be viewed as a function: if f and g have — for some sets X, Y, Z — the signatures

$$f : X \rightarrow Y$$
$$g: Y \rightarrow Z$$

their composition $f ; g$, called h above, has signature $X \rightarrow Z$. Now ";", as defined above, can itself be defined as a function that given any two arguments such as f and g yields a result such as h. That function has signature

$$";": [[X \rightarrow Y] \times [Y \rightarrow Z]] \rightarrow [X \rightarrow Z] \qquad \textbf{[18]}$$

We can go on defining functions that operate on functions that themselves operate on functions and so on. Lambda calculus gives us a vocabulary and rules — a theory — for dealing with such functions at an arbitrary level.

Lambda expressions

First we need a simple notation for defining functions. We will assume that we have at our disposal basic operations such as "+" on integers and reals; this is only for the sake of examples, since lambda calculus can be defined without reference to such existing mathematical theories. The symbol $\triangleq$ will, as usual, mean "is defined as". To define a function "square" of signature

$$square: REAL \rightarrow REAL$$

yielding the square of a number, we write a **lambda expression** as follows:

$$square \triangleq \lambda\, x : REAL \mid x * x \qquad \textbf{[19]}$$

The right-hand side (after the $\triangleq$) is the lambda expression; it denotes the function that, for any x of type $REAL$, yields $x * x$.

The symbol λ, "lambda", is just a matter of convention but gives the whole approach its name. To introduce the value, the lambda calculus literature generally uses a dot **.**, as in $\lambda\, x$: $REAL$ **.** $x * x$; t this does not work in an object-oriented context where "**.**" has another role, so we use a vertical bar | instead.

This is reminiscent of how we define a routine in programming:

```
square (x: REAL): REAL                    [20]
        -- Square of x.
    do
        Result := x * x
    end
```

with a more compact form in line with mathematical practice. The variable following the λ, here x, is known as a **bound variable** of the lambda expression and is similar to the formal arguments of a routine.

The choice of bound variable does not affect the informal meaning of the lambda expression. Just as we may choose the name y for the argument to the routine [20], without affecting the routine's meaning as long as we use y instead of x throughout its text, so does [19] denotes the same function as

$$\lambda\ y : REAL \mid y * y$$

This observation will be formalized below through the notion of *alpha-conversion*.

A lambda expression may have more than one bound variable, as long as all these variables have different names:

$$\begin{array}{l} \lambda\ x, y\colon INTEGER \mid x + y \quad \text{-- The addition function} \\ \lambda\ x\colon NATURAL,\ z\colon REAL \mid z^x \text{ -- Notation when the types are different} \end{array}$$

What do lambda expressions buy us? At first sight, [19] states the same property as if we had just said that *square* is the function such that

$$\forall\ x\colon REAL \mid square\ (x) = x * x \qquad \textbf{[21]}$$

but the difference is that [21] only **talks about** the function *square*, giving properties of its values for possible arguments, whereas [19] **defines** *square* as a mathematical object in its own right, in the same way that we can define the number π by giving its value.

Either an approximate value, of the exact value as a sequence limit or other math formula.

One of the immediate benefits is to allow clear definitions of higher-order functions such as composition (signature given by [18]):

$$\text{";"} \triangleq \lambda f\colon X \rightarrow Y, g\colon Y \rightarrow Z \mid g\ (f\ (x))$$

X, Y, Z are assumed to be known sets. Since they are arbitrary, we could introduce a genericity mechanism for lambda expressions, as for classes, turning X, Y and Z here into formal generic parameters. We do not need such a notation for this short overview of lambda calculus.

In this example the source set in the signature, $[X \rightarrow Y] \times [Y \rightarrow Z]$, is a cartesian product; correspondingly, the lambda expression has two bound variables f and g.

You will have noted that every definition of a function by a lambda expression so far has been preceded by a specification of the signature of the function; in addition, every bound variable is declared with its type (as in $f: X \to Y$), like a formal argument in a routine. It is also possible to omit the types entirely, with lambda expressions such as $\lambda f, g \mid g(f(x))$, yielding the variant of the approach known as **untyped lambda calculus**. Here we will stick to typed lambda calculus, for the same reasons we use typing in programming with languages such as Eiffel: readability, and avoiding errors.

***Touch of Methodology*:**
Declaring the signature

Whenever you define a function by a lambda expression, precede the definition by a declaration of the function's signature.

If the signature appears just before, we may omit the declarations of the bound variables, as in

$$\text{";"}: [[X \to Y] \times [Y \to Z]] \to [X \to Z]$$
$$\text{";"} \triangleq \lambda\ f, g \ \mid g(f(x)) \quad \text{-- No need to declare } f \text{ and } g.$$

Currying

As an example of higher-order function that can be described through a lambda expression, consider *currying*.

Currying — so named in honor of the American mathematician Haskell Curry, one of the founders of the theory known as *combinatory logic* of which lambda calculus is a part — allows us, without loss of generality, to work only with functions of just one argument. First, a notational convention:

***Touch of Notation*:**
Brackets and parentheses

In ordinary mathematical notation, parentheses serve both for grouping and for function application, as in $f\ (a\ *\ (b\ +\ c))$ (innermost for grouping, outermost for application). This would be very confusing in a discussion of operators on functions.

In the present discussion of lambda calculus, parentheses are **only for function application**; grouping uses square brackets. So

$$[f\,;g]\ (a * [b + c])$$

is the application of the function $f\ ;\ g$ (the composition of f and g) to an argument that is the product of a and $b + c$.

When we are given a function, it often has two arguments — like “**;**” as just seen (on functions) and “+” (on reals) — or more. Consider such a function:

$$f: [X \times Y] \rightarrow Z$$

for given X, Y, Z. From f we can define a function f' with signature

$$f': X \rightarrow [Y \rightarrow Z]$$

as

$$f' \triangleq \lambda\, x: X \mid [\lambda\, y: Y \mid f(x, y)] \qquad \mathbf{[22]}$$

What does this mean? Unlike f, function f' takes only one argument, of type X; also unlike f it does not directly yield a result of type Z. Instead, for any argument x it yields a function, highlighted above. Let us call that function g. It *itself* takes an argument y of type Y, then yields a result of type Z. This result $g\ (y)$ is $f\,(x, y)$: the same as if we had directly applied f to *two* arguments.

We say that f' is the **curried** version of f. Currying a two-argument function means turning it into a one-argument function, related to the original by [22]. Another way of expressing this is to say that to curry the function is to *specialize* it on its first argument. This leaves a function of one argument.

If add is the addition operation on integers (which we may write in lambda notation as $add \triangleq \lambda\, x, y\!: INTEGER \mid x + y$), then $curry\ (add)$ is the function

$$add' \triangleq \lambda\, x: INTEGER \mid [\lambda\, y: INTEGER \mid x + y]$$

so that $add'\ (1)$, for example, is $\lambda\ y: INTEGER \mid 1 + x$: the “increment” function, adding one to any given integer.

The correspondence between a two-argument function f and its curried version (called f' above) is one-to-one: informally, we do not lose any information by specializing f on its first argument, since the effect of the second argument remains embodied in the argument of the resulting function f'.

It is interesting — and an example of the expressive power of lambda notation — to state this correspondence between f and f' explicitly, by introducing currying itself as a function, say $curry$, defined by a lambda expression. For given X, Y, Z its signature is

$$curry: [[X \times Y] \rightarrow Z] \quad \rightarrow \quad [X \rightarrow [Y \rightarrow Z]]$$

and its value:

$$curry \triangleq \lambda f: [X \times Y] \rightarrow Z \mid [\lambda\, x: X \mid [\lambda\, y: Y \mid f(x, y)]]$$

You should similarly define the inverse function, yielding f from f'.

→ *Exercise*: *“Uncurrying”, 17-E.9, page 661*.

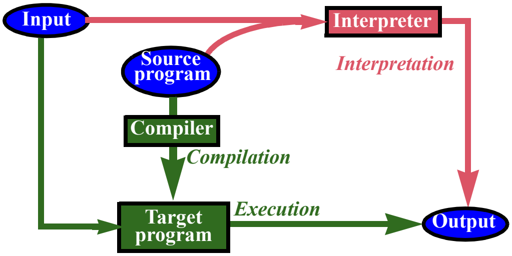
Generalized currying

Although our basic examples all curry a two-argument function on its first argument, it is easy to generalize the concept: you can curry any function of n arguments ($n \geq 1$) on any choice of m arguments ($1 \leq m \leq n$) simply by setting values for these arguments. This yields a function of the remaining $n - m$ arguments, representing a specialized version of the original function, also known as a **partial evaluation**. If $m = n$, you get a constant function.

Currying in practice

As an example of what currying represents in practice, consider the difference between *compilation* and *interpretation*. If we have an interpreter for a programming language, we may view it abstractly as a function of signature

$$\boxed{interpreter: Program \times Input \rightarrow Output}$$



← *Figure from the discussion of compilation, page 330.*

where *Program* is the set of all correct programs in the language, *Input* the set of possible inputs and *Output* the set of possible outputs. (This is a simplified but not incorrect view of what programs are about.) Now a *compiler* produces, from the source program, a machine code program; because we have a mechanism — the hardware — to execute such programs without further effort on our part, we may consider them to be members of the set

$$\boxed{Machine\_program \triangleq Input \rightarrow Output}$$

A compiler generates such a machine code program from a source program, so it is abstractly a function of signature

$$\boxed{compiler: Program \rightarrow [Input \rightarrow Output]}$$

When we have two possible execution mechanisms for the same programming language it is important that they implement identical semantics.

> This is essential for example in EiffelStudio, where you typically go back and forth between the fully compiled, fully optimized *finalized* form of compilation and the fast incremental recompilation or *Melting Ice*, which is mostly interpreted; certainly you want your finalized code as delivered to your customer to produce — just faster — *exactly* the same result as the Melting Ice version.

← *"The melting ice technology", page 357.*

Stating that

```
compiler = curry (interpreter)
```

captures this consistency requirement concisely and elegantly.

The notion of currying is particularly relevant for an object-oriented programmer. At the center of O-O style of programming appears a distinctive style that makes every operation relative to an object, as in the standard call

```
x.f (args)
```

(but also in an "unqualified" call *f* (*args*), which is the same operation applied to the current object, and which you may indeed also write **Current**.*f* (*args*)).

← *"Definitions: Qualified and unqualified call", page 134.*

Object-oriented programmers never really use the idiom "Apply this operation to *those* objects out there", which dominates non-O-O forms of programming and would lead to calls of the form *f* (*x*, *arg1*, *arg2*, …) with all operands treated on an equal footing. Instead it is always "Apply this operation to *this* object *x* over here — oh, and by the way, you might need a couple arguments, here's *args* for you".

In the end you can express with one style what you would with the other, but the consequences of the object-oriented style on the structure of programs are profound: two notions that it makes possible are the class, as a form of both type and module, and inheritance.

All this boils down to the concepts of this section. Object-oriented programming is curried programming.

The calculus

What we have seen so far of lambda calculus is lambda expressions: a notation, providing useful insights, but not a calculus. Relying on that notation, the calculus provides a fascinating theory of functions and the operations on them; it is beyond the scope of this book but you should be familiar with its basic ideas.

Lambda calculus can model the general notion of *computation* through two basic operations on lambda expressions: alpha-conversion and beta-reduction (also written α- and β-).

To define these notions, we need to distinguish between two kinds of occurrence of a variable in a lambda expression: bound and free.

As you remember, we say that $x, y, \ldots$ are the *bound variables* of a lambda expression $\lambda\, x: X, y: Y, \ldots \mid e$. Then an **occurrence** of a variable a in such an expression is **bound** if either:

- *a* is one of the bound variables (*x*, *y*, …).
- The occurrence is (recursively) a bound occurrence of *a* in *e*.

This is an example of a **recursive definition** as discussed in an earlier chapter.

← *Chapter 14.*

The notion immediately generalizes to a non-lambda expression *exp*: an occurrence is bound in *exp* if it is bound in one of its lambda subexpressions. For example in

$$[f\,;\,g]\ (\lambda\ a : \mathit{INTEGER} \mid a + f\,(a, b))$$

the occurrences of *a* are bound, but not those of *f*, *g* and *b*. An occurrence that is not bound, such as those of *f*, *g* and *b* in this example, is **free**. In

$$\lambda\ x : \mathit{INTEGER}\ \mid\ [\lambda\ y : \mathit{INTEGER}\ \mid\ x + y + z]$$

the occurrences of *x* and *y* are bound, but the occurrence of *z* is free. Informally, this means that *x* and *y* are names local to the expression, but *z* must be defined outside of it. This is exactly what we get in programming: in

```
f (x, y: INTEGER): INTEGER do Result := x + y + z end
```

x and *y* are formal arguments, meaning that they are just conventional names used to define the function, and any other names would work if they do not conflict with each other or with names from the enclosing class; but *z* must come from the context. In practice it should be a feature (specifically a query: attribute or function) of the class.

We say that *x* "occurs bound" in an expression *e* if it has at least one bound occurrence in *e*, and that it "occurs free" if it has at least one free occurrence; in the second case *x* is a "**free variable**" of *e*.

We already know what it means for x to be a "bound variable" of e.

The other basic notion is **substitution**:

Definition: variable substitution

Let *exp* be an expression, *x* a variable and *e* another expression. Then

$$exp\ [x := e]$$

denotes the expression obtained from *exp* by replacing (substituting) every *free* occurrence of *x* by *e*.

The := symbol is the same as for assignment in programming, but of course this is a mathematical notation.

For example, if *exp* is

$$\lambda\ \ z : \mathit{INTEGER}\ \mid x + y + z * x$$

and *e* is *sin* (*x*), then *exp* [*x* := *e*] is $\lambda\ z : \mathit{INTEGER}\ \mid\ sin\ (x) + y + z * sin\ (x)$. As this example indicates, *e* may contain several occurrences of *x*.

We only substitute free occurrences: if *exp* is

$$\lambda\, x, z : INTEGER \mid x + y + z * x$$

(the same as before except that x is now bound), then $exp\ [x := e]$ is identical to *exp* since we do not substitute the bound occurrences of x. If *exp* is

$$\lambda\, y : INTEGER \mid f\,(x, [\lambda\, x : INTEGER \mid x + y])$$

we will substitute the first (highlighted) occurrence of x, which is free, but not the bound variable x in the innermost lambda expression, where this would make no sense because in that subexpression x is just an arbitrary name; $\lambda\, z : INTEGER \mid z + y$, where x does not appear, denotes (informally) exactly the same function. Alpha-conversion will make this clear.

We start with the other operation, **beta-reduction**, the central rule that captures the essence of lambda notation. Beta-reduction enables us to get rid of a bound variable (and hence, if it there is no other, of a λ) by transforming

$$[\lambda\, x : X \mid exp]\ (e)$$

into

$$exp\ [x := e]$$

if **no free variable of** e **occurs bound in** *exp*. This neatly expresses the notion of applying a function to actual arguments: since $\lambda\, x : X \mid exp$ intuitively stands for the function that yields *exp* as a function of x, applying it to e should stand for *exp* with every free occurrence of x replaced by e. For example, writing $e \xrightarrow[\beta]{} f$ for "beta-reduction transforms e into f":

→ A slightly less restrictive condition will do; see the exercise "Beta-reduction condition", 17-E.10, page 661.

$$\begin{array}{l}
[\lambda\, x : X \mid x + y]\ (z) \xrightarrow[\beta]{} z + y \\
[\lambda\, x : X \mid x + y]\ (y) \xrightarrow[\beta]{} y + y \\
[\lambda\, x : X \mid x + y]\ (x) \xrightarrow[\beta]{} x + y \\
[\lambda\, x : X \mid z + y]\ (e) \xrightarrow[\beta]{} z + y
\end{array}$$

In the last example, the bound variable x is not actually used in *exp*; we may view the lambda expression as representing a constant function of x. So no substitution occurs when we apply the expression to an arbitrary argument e; the lambda abstraction just disappears.

As the second and third examples show, beta-reduction is possible even if e uses variables that occur in exp, provided these occurrences are not bound. This restriction does not rule out the third example, because exp is $x + y$, where x does not occur bound: it only occurs bound in the full enclosing lambda expression $\lambda\ x : X \mid x + y$. The restriction would only prevent beta-reduction of an expression such as

$$[\lambda\, x : X \mid [\lambda\, y : Y \mid x + y]]\ (y) \qquad \textbf{[23]}$$

where the reduction would yield $\lambda\ y : Y \mid y + y$, which incorrectly confuses y with the bound variable — "incorrectly" according to the informal intent of the lambda expressions involved.

Does this mean we can never — through beta-reduction — simplify an expression $[\lambda\, x : X \mid exp]\ (e)$ if we are unfortunate enough that one of the free variables of e has been chosen as bound variable for some subexpression of exp? This would of course be regrettable since bound variables are just arbitrary names. If we replace [23] by

$$[\lambda\, x : X \mid [\lambda\ z : Y \mid x + z\ ]]\ (y)$$

beta-reduction becomes possible, yielding $\lambda\ z : Y \mid y + z$; but that was just a change of bound variable, not affecting the informal understanding of the underlying function. You do the same, in programming, when you choose a new name for a variable or a formal argument, for example to remove a conflict with the name of an attribute of the enclosing class.

> Some programming languages allow such conflicts, with the convention that the most local name takes precedence. Eiffel prohibits them to avoid the risk of confusion, since finding a new name is easy and makes the program clearer.

To legitimize such harmless changes of bound variable, we need the second rule: **alpha-conversion**. Given a variable y, alpha-conversion transforms a lambda expression

$$\lambda\, x : X, \dots \mid exp$$

in which y has **neither free nor bound occurrences**, into

$$\lambda\, y : X, \dots \mid exp\ [x := y]$$

→ Again the condition is, for simplicity, stronger than needed; see the exercise "Alpha-conversion condition", 17-E.11, page 661.

The condition on y prohibits us from replacing x by y in either of

$$\lambda\, x : X \mid x + y \qquad \textbf{[24]}$$
$$\lambda\, y : X \mid x + y \qquad \textbf{[25]}$$

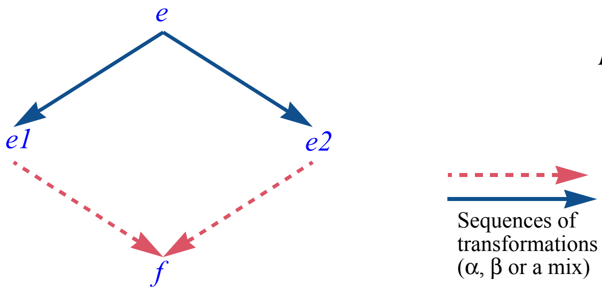
since the resulting expression $\lambda\, y : X \mid y + y$ would, in both cases, lose the intended semantics of the original expression:

- [24] represents a function of one argument, which returns y added to whatever value is passed to the function as argument. For this function, y is a free variable: a value contributed by the context of the expression (for example an enclosing expression). If the substitution were permitted, its result would be $\lambda\, y : X \mid y + y$. This denotes a function of one argument, locally usurping the name y to denote that argument; whatever value you pass to that function, it returns the value added to itself. The two functions are completely different!
- In [25] y is bound, but then alpha-conversion would merge it with the free variable x.

The last observation indicates that the requirement on y as stated above is in fact stronger than it needs to be: we do not need to ban alpha-conversion if y if it has any bound occurrences in *exp*, only if any of these occurrences appears in a context where x is also bound.

→ *Exercise 17-E.11.*

Alpha-conversion and beta-reduction provide the basis for a full-fledged **theory of computation**, which describes any computation as a sequence of such transformations of (possibly complex) lambda expressions. A fundamental consistency property of that theory is the *Church-Rosser Theorem*, stating that if from a given lambda expression e two separate sequences of transformations yield different expressions $e1$ and $e2$, then there exist two other sequences that transform $e1$ and $e2$, respectively into some common expression f (see the illustration below). This means that if several transformations are possible on any particular expression, it does not matter which one you choose to apply first, as you will eventually get to the same "canonical" result f.



Church-Rosser property

Lambda calculus and agents

I hope that as you were reading about lambda calculus you started to make the connection with the concepts of the preceding sections.

The routine as we knew it until the present chapter was a program structuring construct, but could not join the games of program execution, along with references, basic objects (integers etc.) and more complex objects. Like mathematical functions in the absence of a framework such as lambda calculus, routines remained — according to a metaphor widely used in the programming language literature — "second-class citizens".

Like lambda calculus turns functions into first-class citizens of the mathematical world, so do agents set routines loose, wrapped in objects, as fully privileged values for program execution.

Even in the absence of an agent mechanism, routines are a form of lambda expression, and routine call is a form of beta-reduction. But the reduction must be planned statically, through calls such as $f(x, y)$ with f specified explicitly. Object-oriented programming introduces a first element of dynamism thanks to dynamic binding, allowing f to have several variants and the selection between them to take place anew for each call $x.f(x, y)$ on the basis of the type of the object attached to x. This dynamic element is what enables us to address some of the examples of this chapter through the Many Little Wrappers pattern (with the limitations cited); but the choice it offers is restricted to a set of variants specified in advance. With agents, beta-reduction becomes a completely dynamic operation, $a.call([x, y])$, not requiring us to know anything — save for the signature — about the routine that the agent a represents.

The concepts of lambda calculus help us understand the nature of "open" and "closed" operands. They correspond in agents to bound and free variables in lambda expressions.

- In $e \triangleq \lambda\, x : INTEGER \mid x + y$, the bound variable x represents an argument to be provided at the time of beta-reduction; the free variable y comes from the environment, typically an enclosing expression (it will remain unevaluated in a beta-reduction of e).
- In **agent** $f(?, y)$, the open argument will provided at the time of a call; the closed argument y is provided at the time of definition.

You may also have noted that having closed arguments in an agent is essentially to *curry* the routine on these arguments (taking currying in its general form applied to any m of n arguments). Consider the various dynamic forms that we can produce from a routine:

← *"Generalized currying", page 645.*

- **agent** {*C*}.*f* is a dynamic version of the full *f*, faithful to the signature of the original.

- At the other extreme, **agent** *f* (*x*, *y*) and **agent** *t*.*f* (*x*, *y*), with all operands closed, are completely curried versions; you call them (if the value of *a* is such an agent) as *a*.*call* ([]) with no arguments. This, by the way, reminds us of one of the differences between mathematics and programming: we noted earlier that a math function curried on all its arguments is a constant function, but successive calls to *a*.*call* ([]) need not produce the same effect, because even if *a* has not changed the surrounding objects may have.

← *“Math is static, software is dynamic”, page 227.*

- In-between, an agent with some operands closed and some open, such as **agent** *a*.*f* (**?**, *x*, *y*), is similar to a function curried on the closed operands.

In one respect, agents as seen so far are less general than lambda expressions. To use **agent** *a*.*f* (**?**, *x*, *y*), or any of the other variants, we must assume there is a function *f* to build on. It is as if in lambda expressions $\lambda\ x, \ldots \mid exp$ we restricted *exp* to be of the form *f* (*args*) where *f* is a function. We are now going to see how to remove that restriction — and put the final touch on the agent mechanism — by allowing for agents the equivalent of an arbitrary *exp*.

17.7 INLINE AGENTS

The study of lambda calculus suggests a generalization of the basic agent mechanism, giving us flexibility beyond what we have already gained through the introduction of open and closed arguments.

The agents that we have used so far each proceed from an existing routine. But sometimes you want an agent and no routine. You just need to pass along some computation or property as an agent, and it is cumbersome to extend the enclosing class with a routine just for that purpose. The routine may not be an interesting feature for the class, and will just make it seem more complex. **Inline** agents let you define an agent without burdening any class.

The need often arises in writing contracts — of all kinds: preconditions, postconditions, class invariants. For example the invariant of a class could specify that all the elements of a certain array of integers *a* are positive. We already know how to state this thanks to the class *INTEGER_INTERVAL* and the operator function |..| which, for any two integers *a* and *b*, enables us to express the interval *a* |..| *b*. We saw how to state the requisite condition, equivalent to $\forall\ s: a.lower\ ..\ a.upper \mid\ a\,[i] > 0$ in predicate calculus:

← *“Iterating for predicate calculus”, page 628.*

```
(a.lower |..| a.upper).for_all (agent is_positive)                    [26]
```

←*Same as* [10], *page* 628.

To make such a condition meaningful, you must have written a little function for the occasion:

```
is_positive (n: INTEGER): BOOLEAN                                    [27]
        -- Is n greater than zero?
    do
        Result := (n > 0)
    ensure
        definition: Result = (n > 0)
    end
```

This is a bit of a nuisance. Not so much the writing of the code; you should never worry about a few extra keystrokes if the result is relevant. But assume you only need *is_positive* for expressing the above property [26], for example as a clause in a class invariant. You are then encumbering the class with a feature that is not an essential feature of the corresponding data abstraction. This is particularly unpleasant if you have many such properties, as will be the case if you try to write precise and extensive contracts. True, you need not export these features, but they become part of the class anyway. It would be better to express the relevant properties or computations just at the place where you need them, with no visibility beyond that context.

Inline agents fit right here. An inline agent, as the name suggests, is a routine-like declaration yielding an agent — nothing else, no routine of the class — and declared where the agent is needed. The syntax is straightforward as illustrated by the rewriting of the last example; we merge [27] into [26], yielding

```
(a.lower |..| a.upper).for_all
        (agent (n: INTEGER): BOOLEAN                                 [28]
                -- Is n greater than zero?
            do
                Result := (n > 0)
            ensure
                definition: Result = (n > 0)
            end)
```

From the second line on, the text is the same as in [27]; the only difference is that there is no longer a routine name (such as *is_positive*).

This property characterizes an inline agent: it is an **anonymous routine**.

The syntax of an inline agent is indeed, as this example shows, that of a routine declaration, with the routine name replaced by the keyword **agent**. You may include all the components applicable to a routine, such as pre- and postconditions, or a **local** clause to give the agent its own local variables. Their names must be different from those of features of the class and local variables of the enclosing routines; this is different from the convention for lambda expressions (where inner bindings simply take precedence over outer ones), but avoids any confusion. Names are not a scarce resource — or, put differently, you should take care of your own alpha-conversions.

Even though there can be no name conflicts with local variables of the enclosing routines, you may *not* use them directly in the agent. In the rare case you need them, you will have to pass them as arguments to the agent.

Illustrated above for predicates, the inline agent mechanism is just as useful for procedures and functions of any signature.

This mechanism completes our panoply of agent mechanisms, providing a major boost to the expressiveness of our object-oriented programs. The next chapter will take us through a major application of this mechanism, addressing in an elegant way the "observation" problem sketched earlier.

← *"Four applications of agents", page 621.*

17.8 OTHER LANGUAGE CONSTRUCTS

At the beginning of this chapter we saw that a number of situations call for the possibility of passing around data — objects, in an O-O framework — that wrap computations. The agent mechanism addresses that need effectively.

← *"Why objectify operations?", 17.2, page 621.*

Not all programming languages, however, have such a construct. In fact, of languages commonly used in industry, only Eiffel, Smalltalk and C# have something like it (with significant differences in the details). So it is interesting to review briefly what solutions are available depending on the kind of languages you may have to use.

There are basically four approaches:

- A mechanism supporting lambda expressions, such as agents.
- Routines as arguments to other routines.
- Function pointers.
- In object-oriented programming, the Many Little Wrappers pattern.

Agent-like mechanisms

Agents as we have studied them in this chapter is a form of the first approach. **C#** offers *delegates*, which pursue the same aim. Other than (fairly important) differences of spirit and notation, the main difference between C# delegates and Eiffel agents is that the target of a delegate cannot be open; the expression **agent** *{STOP}* **.***close* has no direct equivalent in C#. The C# appendix gives details of the delegate mechanism.

From [17], part of "Open targets", page 638. On C# delegates, see "Delegates and events", page 791.

Smalltalk has a notion of *block*, a segment of code that can be passed around as an object. Note that Smalltalk is an untyped language, meaning that there is no way to check at compile time that blocks will be used with the proper arguments; a mismatch will result in a run-time error.

Functional languages typically support the ability to treat functions (their routines) as data. This was already the case with the original Lisp, where an expression of the form

← "Functional programming and functional languages", page 324.

```
(defun f (x y) ("expression involving x and y"))
```

Lisp syntax.

defines f as a function of two arguments. Then you can use f as argument to another function, for example in

```
(curry f)
```

Lisp syntax.

where curry itself can be defined in Lisp. The language was indeed defined explicitly on the basis of (untyped) lambda calculus, so it is not surprising that much of what we have seen in this chapter can be done fairly naturally. This also applies to more recent functional languages, such as Haskell and ML. Two points are worth noting:

- Functional languages were not initially object-oriented. Some of them have added O-O constructs such as class and inheritance, but not all the concepts that we have taken for granted are applicable in a functional environment.
- While some functional languages are statically typed, others are not; whether you get the benefits of static type checking depends on which variant you use.

The term **closure** is often used about functional languages to denote expressions representing routines that can be passed around as data even though they may need to access global variables.

Routines as arguments

A number of programming languages allow you to pass a routine as argument to another routine, with a syntax such as

```
integral (f: function (x: REAL): REAL ; x, y: REAL): REAL
```

Not the exact syntax of a specific language.

You can then pass to integral, as actual argument, a routine with a matching signature, as in integral (cosine, 0, 1). The language must provide an appropriate notation to call the corresponding routine, here through f, from within the code of a routine such as integral.

Compared to agents or closures, this solution has limitations:

- “Routine” is a special argument type which does not generally fit well in the type system of the language.
- Typically, information about a routine is not a well-defined value, as it is in the case of an agent or a closure, and hence cannot be assigned to a variable (for which, because of the previous point, it would be hard to declare a type); it can only be used as argument to a routine.
- Because there is no proper typing for routine arguments, it is generally not possible or at least not simple to move up in abstraction and define functions such as composition or currying.
- All you can do on a routine argument is to call it. In contrast, agents are full-fledged objects whose features provide information on the associated routine.
- Some issues arise when routines access global variables; they affect the compiler writer but also, to some extent, the programmer.
- The approach does not fit well with an object-oriented scheme, since it uses data other than objects.

The approach, however, fills many of the basic needs and has been used successfully in non-O-O languages, going as far back as Fortran and continuing with Pascal and several of its successors.

Function pointers

Computers, as you know, use memory to store not only objects but programs. At run time, a particular routine resides at a particular address, and it is possible to transfer execution to the code at that address. If there is a way for the program to denote that address, and a mechanism to say “execute routine at address *a*, then return and continue”, you can treat routine addresses as data through which to call the corresponding routines.

← “The stored-program computer”, page 10.

At the machine level this technique is what makes all the others possible:

- When you use a routine as argument to another routine, what the compiler will actually pass is the routine's address.
- An agent object will internally contain — although not in a field that your program can directly access — the address of the associated routine.
- Dynamic binding, necessary for the Many Little Wrappers pattern, assumes the run-time ability to call a routine through its address, stored in some data structure representing properties of a type. The routine table, studied as part of the implementation of inheritance, is an example of such a structure.

← *"A peek at the implementation", 16.8, page 575.*

All these techniques, however, are for the compiler to use when generating code, not for the application programmer when writing programs; they hide the physical routine address under one or more layers of abstraction, enabling programmers to think in high-level terms: routines (or groups of redeclared routines known through a common signature and contract), agents, objects.

C and C++ let you pass the name of a function (the only kind of routine, procedures being treated as functions with a "void" result type) as actual argument, or assign it to a variable. Then if *f* is the corresponding formal argument or variable, you can call the original function through

→ *Appendices C and D.*

```
(*f) (args)
```

When declaring a formal argument representing a function you can specify the full signature, known as a *prototype*, so that an actual argument that does not match the signature will be rejected at compile time. This technique then becomes the same as the previous one ("routines as arguments"). Providing the signature is, however, not compulsory; you can get away without it at the price of a possible compile-time "warning" — a message that signals a possible problem but does not prevent compilation. With this option, which assimilates the function name to the corresponding machine-level address, you gain the same flexibility as if you were programming in assembly language but lose the benefits of type checking.

Many Little Wrappers and nested classes

If a programming language does not support any of the preceding techniques but is object-oriented — with classes, inheritance, polymorphism and dynamic binding — you can use the Many Little Wrappers pattern studied at the beginning of this chapter.

The main disadvantage is the need to write many little classes, often with just one routine.

Java, which has no agent-like mechanism and no way to pass routines as arguments, mitigates this problem by allowing the programmer to declare a class as local to another class; this is known as a **nested class**. You can then use that class, as if it were a feature of the enclosing class, to describe objects that will only need to be created by features of the latter. This technique avoids polluting the global name space of the program (that is to say, the set of class names directly available to other software components); but the basic problems remain the same.

→ On Java's relation to agents, see "Agents", page 766 and "Nested and anonymous classes", page 767.

17.9 FURTHER READING

J. Roger Hindley and Jonathan P. Seldin: *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts, Cambridge University Press, 1986,

> Classic reference on lambda calculus and the companion theory of combinators (which directly serves as the basis for some functional programming languages). A mathematical text, not written specifically for computer scientists; remarkably clear, defines all needed concepts.

Chris Hankin: *An Introduction to Lambda Calculi for Computer Scientists*, King's College Publications, London, 2004.

> This one is specifically intended for computer scientists.

17.10 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Many program schemes benefit from a mechanism for packaging a routine into an object and storing it away for later call. The corresponding language construct may be called "agent"; other common names include "delegate" (in the C# language) and "closure".
- An agent wrapping a routine can be treated as any other object, for example assigned to variables and passed around the program structure through feature calls. It can be called at any time through a feature applicable to all agents; this triggers a call of the associated routine, but the context of the agent's call need not know, and usually does not know, what that routine is.
- Agents can have any number of "open operands", corresponding to the bound variables of a lambda expression. Open operands may include some or all of the arguments, as well as the target. Closed arguments (the non-open ones) are specified in the agent's definition; open arguments must be provided, in the form of a tuple, for each call to the agent.

- Agents can be defined on the basis of an existing routine; it suffices to specify the values of closed operands if any. To avoid defining a new routine when none is available, it is also possible to declare an agent "inline" by writing the instructions directly in the agent's definition.

- In a programming language not supporting agents or a similar mechanism, passing functions around as data requires the use of many wrapper classes, or routines as arguments, or routine addresses. These solutions are less convenient and, in the last case, less type-safe.

- The theory of lambda calculus provides a mathematical framework for understanding agent.

- A lambda expression includes *bound variables* and a defining expression (itself possibly a lambda expression), which may involve the bound variables, as well as other variables said to occur *free*. It represents a function; applying the function to arguments yields the defining expression after substitution of each argument for the corresponding bound variable. This process is known as *beta-reduction*.

- The bound variables of a lambda expression are arbitrary names. They can be changed throughout the expression (including in its defining expression) as long as this does not create any conflicts, in particular with free variables. This process is known as *alpha-conversion*.

- To *curry* a function of n arguments is to specialize it on m of its arguments ($1 \leq m < n$), leaving a function of $n - m$ arguments.

New vocabulary

Agent
Alpha-conversion
Beta-reduction
Church-Rosser property
Closed operand
Closure
First-class citizen
Inline agent
Lambda calculus
Lambda expression
Many Little Wrappers pattern
Nested class
One-Song-Artist class
Open operand
Operand
Partial evaluation
Prototype (C, C++)
Substitution (of a variable in an expression)

17-E EXERCISES

17-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

17-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise "Concept map", 16-E.2, page 616.*

17-E.3 An integration class without agents

See the corresponding "Programming Time!".

← *Page 625.*

17-E.4 Iterator objects

Devise an iterating mechanism that does not use agents but relies on a *LINEAR_ITERATOR* class describing objects able to iterate a specific operation on a linear structure such as a list.

17-E.5 An iterator that shoots itself in the foot

(This is a programming exercise of the masochistic kind, asking you to violate a methodology prescription just to contemplate the resulting mess.) Working with a descendant of *LINEAR* such as *LINKED_LIST*, use the procedure *do_all* with an agent argument representing a routine that — disregarding the explicit prescription in *do_all*'s header comment — changes the structure, in such a way that *do_all* crashes execution or produces an otherwise inconsistent result. With the help of the debugger if needed, analyze the exact circumstances leading to this failure.

17-E.6 Manual optimization

Rewrite the *do_all* iterator of *LINEAR* so that it does not use a manifest tuple as argument to *call*, but instead a tuple variable *t* that is refilled with a new value before each call. *Hint 1*: to create the tuple object initially, just assign it a value. *Hint 2*: Re-read about the properties of tuples, especially tags.

← *See [12], page 633, and the discussion that follows. On tuples: 13.5, page 389.*

17-E.7 Visiting with agents

Consider an existing set of classes, for example a subset of the Traffic classes. Assume that programmers can write *visit* operations that may have a variant for each of these classes; they take the target object as argument. The aim of the exercise is to define a feature *apply* that applies the appropriate *visit* operation to any such object, passed as argument, without knowing the specific type. (The feature *apply* can declare that argument as being of type *ANY*, or use *C* for some other class *C* known to be a common ancestor of all applicable classes.)

You are not permitted to modify any of the target classes, or their ancestors. The Visitor pattern is not applicable since you may not assume that the classes are descendants of a *VISITOR* class.

← *"Reversing the structure: visitors and agents", page 606.*

Show that it is possible to use agents to achieve the desired goal. *Hint*: follow the model of iterator classes as defined in this chapter.

You may find a solution — not a hint, the full design and implementation — in the Visitor componentization article cited in an earlier discussion.

← *"Further reading", 16.15, page 613.*

17-E.8 The Halting Problem with agents

Devise a more concise proof of undecidability of the Halting Problem, not using any files, directories or string representation of program texts, but instead working on program elements passed as agents.

← *"An application: proving the undecidability of the halting problem", 8.9, page 223.* See also *"From loops to recursion", 14.6, page 471.*

17-E.9 Uncurrying

It was noted that currying is a one-to-one function. Write the signature and definition of the function *uncurry* that, given a one-argument function f' whose result is a one-argument function, yields the associated two-argument function f such that $f' = curry\ (f)$.

17-E.10 Beta-reduction condition

Show that the condition for beta-reduction of $[\lambda\ x : X \mid exp]\ (e)$, "no free variable of e occurs bound in exp", is stronger than actually needed for the reduction to preserve the informal semantics of function application, and devise a less restrictive but still correct condition.

17-E.11 Alpha-conversion condition

Show that the condition for alpha-conversion of $e \triangleq \lambda\ x : X \mid exp$ into $\lambda\ y : X \mid exp\ [x := y]$, "$y$ occurs neither free nor bound in e", is stronger than actually needed for the reduction to preserve the informal semantics of change of variable, and devise a less restrictive but still correct condition.

18

Event-driven design

Who's in charge?

In the style of programming that we have used so far, the program defines the order of operations. It follows its own scenario, defined by control structures: sequence, conditional, loop. The external world has its say — through user interaction, database access and other input, affecting the conditions that control loops, conditionals and dynamic binding; but it is the program that decides when to evaluate these conditions.

In this chapter we explore another scheme, where the program no longer specifies the sequencing of operations directly but is organized instead as a set of *services* ready to be triggered in response to *events*, such as might result from a user clicking a button, a sensor detecting a temperature change, a message arriving on a communication port. At any time, the next event determines which service gets solicited. Once that service has carried out its function, the program gets back to waiting for events.

Such an **event-driven** scheme requires proper initialization: before the real action begins, there must be a setup phase to register services with event types.

This architectural style — in the end another control structure, to be added to our previous catalog — is also known as **publish-subscribe**, a metaphor emphasizing a possible division of roles between software elements:

← *Chapter 7, Control structures.*

- Some elements, the *publishers*, may trigger events during execution.
- Some elements, the *subscribers*, express their interest in certain types of events, indicating what services they want provided in response.

These roles are not exclusive, as some subscribers may trigger events of their own. Note that "event" is a *software* concept: even when events originate outside of the software — mouse click, sensor measurement, message arrival — they must be translated into software events for processing; and the software may trigger its own events, unrelated to any external impulse.

Event-driven programming is applicable to many different areas of programming. It has been particularly successful for Graphical User Interfaces (GUI), which will be our primary example.

18.1 EVENT-DRIVEN GUI PROGRAMMING

Good old input

Before we had GUIs, programs would take their input from some sequential medium. For example a program would read a sequence of lines, processing each of them along the way:

```
from
    read_line
    count := 0
until
    exhausted
loop
    count := count + 1
        -- Store last_line at position count in Result:
    Result [count] := last_line
    read_line
end
```

GENERAL DESCRIPTION 2801 · ISSUE 3
TELETYPE
MODEL 28 PAGE PRINTER
A new concept in Data Communications
TELETYPE model 28
THE LEADERSHIP LINE

By permission of Alcatel-Lucent USA, credits p. 847.

where *read_line* attempts to read the next line of input, leaving it in *last_line*, and *exhausted* does not refer to the mood of the programmer but is set to true by *read_line* if there are no more lines to be consumed.

With such a scheme **the program is in control**: it decides when it needs some input. The rest of the world — here a file, or a user typing in lines at a terminal — has to provide that input.

Modern interfaces

Welcome to the modern world. If you write a program with a GUI, you let users choose, at each step, what they want to do, out of many possibilities — including some unrelated to your program, since a user may go to another window, for example to answer an email.

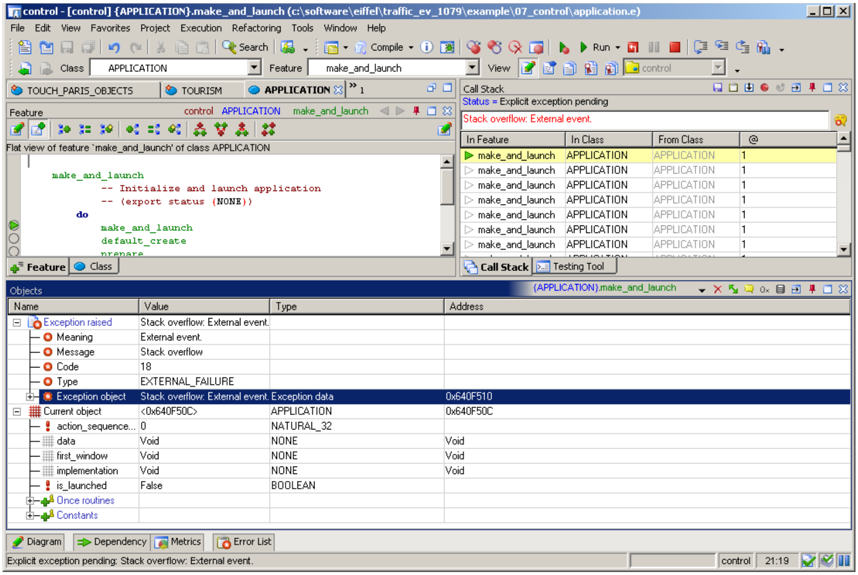
Consider the screen on the facing page. (It illustrates a stack overflow from infinite recursion, triggered by running a program in EiffelStudio. The precise example is irrelevant — any modern GUI program would do.) The user interface includes "*controls*": text fields, buttons, menus, grids and others.

← Stack overflow was mentioned in "Vicious circle?", page 473.

We expect that the user will perform some input action, and we want to process it appropriately in our program. The action might be to type characters into the text field at the top left, to click a button, or to select a menu.

A "control" is a GUI element, such as a window or button. This is Windows terminology; in the Unix world the term is "widgets".

But which of these will happen first? Indeed, will any happen at all?



control - [control] {APPLICATION}.make_and_launch (c:\software\eiffel\traffic_ev_1079\example\07_control\application.e)

Flat view of feature `make_and_launch' of class APPLICATION

```
    make_and_launch
            -- Initialize and launch application
            -- (export status {NONE})
        do
            make_and_launch
            default_create
            prepare
```

Call Stack

Status = Explicit exception pending

Stack overflow: External event.

In Feature	In Class	From Class	@
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1
make_and_launch	APPLICATION	APPLICATION	1

Name	Value	Type	Address
Exception raised	Stack overflow: External event.		
Meaning	External event.		
Message	Stack overflow		
Code	18		
Type	EXTERNAL_FAILURE		
Exception object	Stack overflow: External event.	Exception data	0x640F510
Current object	<0x640F50C>	APPLICATION	0x640F50C
action_sequence...	0	NATURAL_32	
data	Void	NONE	Void
first_window	Void	NONE	Void
implementation	Void	NONE	Void
is_launched	False	BOOLEAN	
Once routines			
Constants			

Explicit exception pending: Stack overflow: External event.

A program GUI

We do not know.

Of course we could use a big **if** … **then** … **elseif** … **end**, or a multi-branch listing all possibilities:

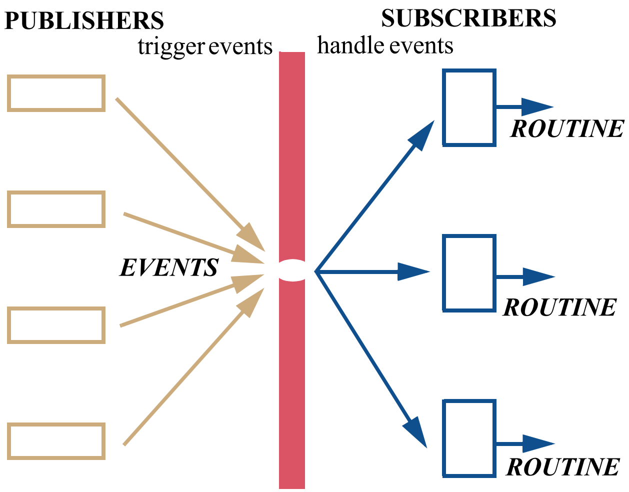
```
inspect
    user_action
when "Clicked the Stop button" then
    "Terminate execution"
when "Entered text in the Class Name field" then
    "Update the top-left subwindow to show the corresponding class"
when … Many other branches …
end
```

but this suffers from all the problems we have seen with multiple-choice algorithm structures (explaining why dynamic binding is such an improvement): it is big and complex, and highly sensitive to any change in the setup. We want a simpler and more stable architecture, which will not require updating each time there is a new control.

← *"Beware of choices bearing many cases", page 574.*

Event-driven (publish-subscribe) design addresses such a situation through a completely different scheme.

We may picture this scheme as one of those nuclear physics experiments (see the figure on the next page) that hurl various particles at some innocent screen pierced with a little hole, to find out what might show up on the other side.



Triggering and handling events

The publish-subscribe style is useful in many different application areas; GUI programming is just an example. You can find many others, in fields such as:

- Communication and networking, where a node on a network can broadcast messages that any other node may pick up.
- Process control. This term covers software systems associated with industrial processes, for example in factories. Such a system might have sensors monitoring temperature, pressure, humidity; any new recording, or just those exceeding specified thresholds, may trigger an event which some elements of the software are prepared to handle.

18.2 TERMINOLOGY

In describing event-driven programming it is important to define the concepts carefully, distinguishing in particular — as in other areas of programming — between types and instances.

Events, publishers and subscribers

Definitions: Event

An **event** is a run-time operation, executed by a software element to make some information (including the information that it occurred) available for potential use by certain software elements not specified by the operation.

This definition highlights the distinctive properties of events:

- An event releases some **information**. A mouse click should indicate the cursor position; a temperature change, the old and new temperatures.
- Part of the information, always included, is that the event **occurred**: on 5 August 1492, Christopher Columbus set sail; five minutes ago (this is less widely known) I clicked the left button of my mouse. Usually there is more: when and where did Columbus sail? What were the cursor coordinates? But in some cases all that matters is that the event occurred, as with a *timeout* event indicating that a previously set deadline has passed.
- **"Certain"** software elements can use this information. This is sufficiently vague to permit various setups: allowing *any* module of the system to find out about events, or identifying *specific* modules as the only eligible ones.
- In all cases, however, what characterizes event-driven design is that **the event itself does not name the recipients**. Otherwise an event would just be like a routine call, such as $x.f(a, b, c)$, which satisfies all the other properties of the definition: it is an operation that makes information (the arguments a, b, c) available to a software element (the feature f). But when you call a routine you explicitly say whom you are calling. An event is different: it just sends the information out there, for consumption by any software element that has the ability to process it.

  Remember that for our purposes an event is a **software** operation; phenomena triggered outside of the software may be called *external events*. An example such as "mouse click event" does not denote the user's click action, an external event; it is the result of a GUI library detecting the external event and turning it into a software event, which other parts of the software can process. In addition to such cases, a system may also have its own software-only events.

Some associated terminology, most of it already encountered informally:

Definitions: Trigger, publish, publisher, subscriber

To **trigger** (or **publish**) an event is to execute it. A software element that may trigger events is a **publisher**. A software element that may use the event's information is a **subscriber**.

Remember that an event is defined as an operation to be executed.

The same software element may act as both a publisher and a subscriber; in particular it is a common scheme for a subscriber to react to an event by triggering another event.

In the literature you will encounter competitors to the above terms: subscribers are also called **observers**, hence the "Observer pattern" studied next; they are said to **observe** the publishers but also, without fear of mixing sensory metaphors, to **listen** to them, gaining one more name: **listener**. Publishers, the targets of all this visual or auditory attention, are entitled to their own synonyms: **subject** and — in "observer" terminology — **observed**.

→ "Listener" is used in Java programming. See "Nested and anonymous classes", page 767.

Arguments and event types

We need a name for the information that comes — according to the definition — with any event:

> **Definitions: Argument**
>
> The information associated with an event (other than the information that the event occurred) constitutes the event's **arguments**.

The term "argument" highlights the similarity with routines. Pushing this similarity further, we will assume that the arguments are grouped in an ordered list, like the arguments in a call *x*.*f* (*a*, *b*, *c*). As with routines, the list can be empty; this would be the case in the timeout example.

How do subscribers find out that an event occurred? One model is *polling*: checking repeatedly (as when you subscribe to a newspaper and go see whether the day's edition has been delivered to your mailbox). Another is **notification**: the triggering of an event causes all potential recipients to be notified.

Models for distributing information over the Internet are classified into "pull" (waiting for users to access information) and "push" (sending it to them). The distinction between polling and notification is similar.

The notification model is more flexible and we will assume it from now on. It can only work if subscribers express their interest in advance, just as you subscribe to a newspaper to receive it every day. But to *what* can you subscribe? It cannot be to an *event*: the event is an operation occurring once: before it is triggered the event does not exist, and afterwards it is too late to subscribe to it! This would be like subscribing to today's newspaper after you have spotted the headline on your neighbor's copy, or retroactively buying shares of a company after the announcement of its latest dividend.

What subscribers need is an **event type**, describing possible events that share general characteristics. For example all left-button mouse clicks are of the same event type, but of a different type from key-press events. This notion of event type plays a central role in event-driven design and will be the central abstraction in our search for a good O-O architecture.

All events of a type have the same *argument type list*. For example, the argument list for any left mouse click event includes the mouse coordinates, two integers. Here too we may borrow a concept from routines, the **signature**, or list of argument types — a procedure *print* (*v*: *VALUE*; *f*: *FORMAT*) has signature [*VALUE*, *FORMAT*], a list of types — and extend it to event types:

← "Anatomy of a routine declaration", page 215. For a function, the signature also includes the result's type.

> **Definitions: Event type, signature**
>
> Any event belongs to an **event type**.
> All events of a given event type have the same **signature**.

For example:

- The signature for "temperature change" may be [*REAL*, *REAL*] to represent old and new temperatures.

- A "left click" event type may have signature [*INTEGER*, *INTEGER*].

  It is also possible to have a single "mouse click" event with a third signature component indicating which button was clicked. This is the case in the EiffelVision library, which also adds arguments such as pressure applied, useful (especially in game applications) for joysticks and exotic pointing devices.

- Although we might define an event type for each key on the keyboard, it is more attractive to use a single "key press" event type of signature [*CHARACTER*], where the argument is the key code.

- For an event type such as "timeout" describing events without arguments, the signature is empty, just as with an argument-less routine.

Whenever a publisher triggers an event, it must provide a value for every argument (if any): mouse coordinates, key code, temperatures. This is once again as with routines, where every call must provide actual arguments.

> The term "event *type*" may suggest another analogy, where event types correspond to the **types** of O-O programming (classes, possibly with generic parameters), and events to their *instances* (objects). But comparing event types to **routines** is more appropriate; then an *event* of a given type corresponds to one specific *call* to a routine.

In this analysis, then, an event is not an object — and **an event type is not a class**. Instead the *general notion* of event type is a class, called *EVENT_TYPE* below; and *one particular event type*, for example "left-button mouse click" (the *idea* of left clicks, not that one time last Monday when I distractedly clicked OK in response to "Delete all?"), is an object. As always when you are hesitating about introducing a class, the criterion is "is this a meaningful data abstraction, with a set of well-understood operations applicable to all instances?". Here:

- If we decided to build a class to represent a particular event type, its instances would be events of that type; but they have no useful features. More precisely, an event has its own data, the arguments, but we only need queries to access these arguments; there are no commands.

- In contrast, if we treat an event type as an object, there are a number of clearly useful commands and queries: trigger a particular event of this type now, with given arguments; subscribe a given subscriber to this event type; unsubscribe a subscriber; list the subscribers; find out how many events of this type have been triggered so far; and so on. This is the kind of rich feature set that characterizes a legitimate class.

← *Class EVENT_TYPE in the final design*: *"Using agents: the event library", 18.5, page 686*.

Not treating each event as an object is also good for performance, since it is common for execution to trigger many events; every tiny move of the cursor is an event, so we should avoid creating all the corresponding objects — even though this does not get us out of the woods since the *arguments* of each event must still be recorded, each represented by a tuple. A good GUI library will remove the performance overhead by recognizing a sequence of contiguous moves in close succession and allocating just one tuple instead of dozens or hundreds.

It is useful to have terms for subscribers' actions with event types and events:

Definitions: Subscribe, register, handle, catch

A software element may become a subscriber to a certain event type by **subscribing** (or **registering**) to it. By doing so it asks to be notified of future events of that event type, so that it can obtain the associated arguments and execute specified actions in response.

When a subscriber gets notified of an event to whose type it has subscribed, it **handles** (or **catches**) the event by executing the registered action.

Although registration (and deregistration) may occur at any time, it is common to have an initialization phase that puts subscriptions in place, followed by the main execution step where publishers trigger events which subscribers handle.

Registering, for a subscriber, means specifying a certain action for execution in response to any event of the specified type. There must be a way for the action to obtain the values of the event's arguments. The obvious way to achieve such registration is to specify a **routine**, whose signature matches the event type's signature. Then an event of the given type will cause a call to the routine, with the event's arguments serving as actual arguments to the call.

We now have the full picture of how an event-driven design works:

The event-driven scheme

E1 Some elements, *publishers*, make known to the rest of the system what *event types* they may trigger.

E2 Some elements, *subscribers*, are interested in *handling* events of certain event types. They *register* the corresponding actions.

E3 At any time, a publisher can *trigger* an event. This will cause execution of actions registered by subscribers for the event's type. These actions can use the event's arguments.

In the GUI example:

E1 A publisher is some element of the software that tracks input devices and triggers events under specified circumstances, for example mouse click or key press. You usually do not need to write such software; rather, you rely on a **GUI library** — EiffelVision for Eiffel, Swing for Java, Windows Forms for .NET… — that takes care of triggering the right events.

E2 A subscriber is any element that needs to handle such GUI events; it registers the routines it wants to execute in response to these events. For example you may register, for the mouse click event type on a button that says “OK” in a file-saving dialog, a routine that saves the file.

E3 If, during execution, a user clicks the OK button, this will cause execution of the routine — or routines — registered for the event type.

An important property of this scheme, illustrated by the separation between the two sides of our earlier figure, is that subscribers and publishers do not need to know about each other. More precisely, the definition of “event” *requires* that publishers do not know the subscribers; the other way around it is more a matter of methodology, and we will see how various architectural solutions fare against this criterion.

← *“Triggering and handling events”, page 666.*

Keeping the distinction clear

You might think the distinction between events and event types obvious, but in fact — this is a warning, to help you understand the literature if you start using various event-driven programming mechanisms — many descriptions confuse the two; this can make simple things sound tricky.

The following excerpt comes from the introductory presentation of event handling in the online documentation of .NET, a Microsoft framework whose concepts are reflected in the C# and Visual Basic .NET languages:

From msdn2.microsoft.com/en-us/library/awbftdfh(VS.80).aspx as of April 2009. Numbers, italics and colors added.

> ### Events Overview
>
> *Events* have the following properties:
>
> 1 The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
>
> 2 An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
>
> 3 Events that have no subscribers are never called.
>
> 4 Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
>
> 5 When an event has multiple subscribers, the *event* handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].
>
> 6 Events can be used to synchronize threads.
>
> 7 In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

I have highlighted in green those occurrences of "event" where I think the authors really mean event, and in yellow those for which they mean event type (a term that does occur in the .NET documentation, but rarely). Where the word is in *italics*, it covers both. This is all my interpretation, but I think that you will agree. In particular:

- It is not possible (points 1, 5) to subscribe to an event; as we have seen, the event does not exist until it has been raised, and when it has been raised that is too late. (Nice idea, though: wouldn't you like to subscribe retroactively to the event "IBM's shares rise by at least 5%"?) A subscriber subscribes to an event *type* — to declare that it wishes to be notified of any *event* of that type raised during execution.
- Point 7 talks about properties of *classes* describing event *types*, as indeed in .NET every event type must be declared as a class. Such a class must inherit from the "delegate" class EventHandler (.NET delegate classes provide an agent-like mechanism) and use another class EventArgs describing the notion of event arguments.
- Point 3 sounds mysterious until you realize that it means: "If an *event type* has no subscriber, triggering an *event* of that type has no effect." All it describes is an internal optimization: by detecting that an event type has no subscriber, the event mechanism can remove the overhead of raising the corresponding events, which in .NET implies creating an object for each. (The mystery is compounded by the use of "call" for what the rest of the documentation refers to as "raising" an event.)

The possibility of confusion is particularly vivid in two places:

- "*A subscriber can handle multiple events from multiple publishers*" (point 2). This comment might seem to suggest some sophisticated concurrent computation scheme, where a subscriber catches events from various places at once, but in reality it is just a mundane observation: a given subscriber may register for several event types, and several publishers may trigger events of a given type.
- Point 5 states that when "*an event*" has multiple subscribers, each will handle it synchronously (meaning right away, blocking further processing) when "*an event*" is raised. Read literally, this would suggest that two "events" are involved! That is not the idea: the sentence is simply trying to say that when multiple subscribers have registered for a certain event *type*, they handle the corresponding *events* synchronously. It uses a single word, in the same breath, with two different meanings.

So when you read about event-driven schemes remember to ask yourself whether people are talking about events or event types — and (since this is the time for one of our periodic exhortations) please make sure that your own technical documentation defines and uses precise terminology.

Contexts

A subscriber that registers says: “for events of *this type*, execute *that action*”. In practice it may be useful, especially for GUI applications, to provide one more piece of information: “for events of this type occurring *in that context*, execute that action”. For example:

- “If the user clicks the left button *on the OK button*, save the file”.
- “If the mouse enters *this window*, change the border color to red ”.
- “If this sensor reports a temperature *above* 25^o *C*, ring the alarm”.

In the first case the “context” is an icon and the event type is “mouse click”; in the second, they are a window and “mouse enter”; in the third, a temperature sensor and a measurement report.

For GUI programming, a context is usually just a user interface element. As the last example indicates, the notion is more general; a context can be any boolean-valued condition. This covers the GUI example as a special case, taking as boolean condition a property such as “the cursor is on this button” or “the cursor has entered that window”. Here is a general definition:

Definition: Context

In event-driven design, a **context** is a boolean expression specified by a subscriber at *registration* time, but evaluated at *triggering* time, such that the registered action will only be executed if the evaluation yields **True**.

← *“Writing an iterator”, page 631.*

We had a taste of the notion of context — in a programming style that was not event-driven — when we encountered iterators such as *do_if*, which performs an action on all the items of a structure that satisfy a certain condition: this is similar to how a context enables a subscriber to state that it is interested in events of a certain type but only if a certain condition holds at triggering time.

We could do without the notion of context by including the associated condition in the registered action itself, which we could write, for example

```
if “The cursor is on the Exit icon” then
    “Normal code for the action”
end
```

but it is more convenient to distinguish the condition by specifying it, along with the event type and the action, at the time of registration.

18.3 PUBLISH-SUBSCRIBE REQUIREMENTS

With the concepts in place, we will now look for a general solution to the problem of devising an event-driven architecture. We start with the constraints that any good solution must satisfy.

Publishers and subscribers

In devising a software architecture supporting the publish-subscribe paradigm, we should consider the following requirements.

- *Publishers must not need to know who the subscribers are*: they trigger events, but, per the basic definition of events, do not know who may process them. This is typically the case if the publisher is a GUI library: the routines of the library know how to detect a user event such as a click, but should not have to know about any particular application that reacts to these events, or how it reacts. To an application, a button click may signal a request to start the compilation, run the payroll, shut down the factory or launch the rocket. To the GUI library, a click is just a click.

- *Any event triggered by one publisher may be consumed by several subscribers*. A temperature change in a factory control system may have to be reflected in many different places that "observe" the event type, for example an alphanumeric temperature display, a graphical display, a database that records all value changes, or a security system that triggers certain actions if the value is beyond preset bounds.

- *The subscribers should not need to know about the publishers*. This is a more advanced requirement, but often desirable too: subscribers know about event types to which they subscribe, but do not have to know where these event types come from. Remember that one of the aims of event-driven design is to provide a flexible architecture where we can plug in various publishers and various subscribers, possibly written by different people at different times.

- *You may wish to let subscribers register and deregister while the application is running*. The usual scheme is that registration occurs during initialization, to set things up before "real" execution starts; but this is not an obligation, and the extra flexibility may be useful.

- *It should be possible to make events dependent or not on a context*. We have seen the usefulness of binding events to contexts, but the solution should also provide the ability — without having to define an artificial context — just to subscribe to an event regardless of where it happens.

- *It should be possible to connect publishers and subscribers with minimal work*. The actions to be subscribed often come from an existing application, to which you want to add an event-driven scheme. To connect the two sides you will have to add some program text, often called "**glue code**"; the less of it the better.

The last requirement is critical to the quality of a system's architecture, especially when the goal is to build user interfaces: you should not have to design the core of an application differently because of a particular interface. This observation directly leads to our next notions, model and view.

The model and the view

For user interface design we need not only to separate subscribers from publishers but also to distinguish two complementary aspects of an application:

Definitions: model, view of a software system

The **model** (also called *business model*) is the part of a software system that handles data representing information from the application domain.

A **view** is a presentation of part of that information, in the system's interaction with the outside: human users, material devices, other software.

← *"Definitions: Data, information", page 8.*

"*Application domain*" as used in this definition is also a common phrase, denoting the technical area in which or for which the software operates. For a payroll processing program, the application domain is human resources of companies; for a text preparation program it is text processing; for flight control software, the application domain is air traffic control.

Although the application domain need not have anything to do with software, the "*model*" is a part of the software: the part that deals with that application domain. For payroll processing it is the part of the software that processes information on employees and hours worked, computes salaries, updates the database. For the flight system it is the part that determines airplane itineraries, takeoff times, authorizations and so on. One could say that the model is the part of the software that does the "real job" at hand, independently of interaction with users of the software and the rest of the world.

← *See also the notion of target class in "Reversing the structure: visitors and agents", 16.14, page 606.*

> "Business model" is more precise but we usually just say "model" because the word "business" may be misinterpreted as restricting us to business-oriented application domains (company management, finance etc.) at the expense of engineering domains such as text processing and flight control.

A "*view*" is a presentation of the information, typically for input or output. A GUI is a view: for example a flight system has a user interface allowing controllers to follow plane trajectories and enter commands.

Usually a program covers just one — possibly broad — application domain, but it may have more than one view, hence "*the* model" and "*a* view" in the above definition. It is then good practice to assign the two aspects to two different parts of a system's architecture. In a naïve design for a small program you might not pay much attention to this issue. But in a significant system you should, if only because you may need to plan for *several views*, such as:

- A GUI view (occasionally, several).
- A Web view ("WUI"), allowing use of the system through a Web browser.
- A purely textual (non-graphical) interface, for situations in which graphics support is not available.
- A "*batch*" interface where the system takes its input from a prepared scenario and produces its output in one chunk. This is particularly useful for *testing* interactive systems. Interactive testing is hard, as it requires people spending long sessions with the system to try many different combinations; you may instead prepare a collection of scenarios (typically recorded from sessions with human users) and run them without further interaction.
- Views provided by other programs, running locally and accessing the functionality through an API.
- *Web service* views provided by programs running on other computers and accessing the functionality through a Web-directed API. (Web services require specific techniques, such as the SOAP protocol.)

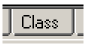
Often one view is enough at the beginning; that is why it is a common design mistake to build a system with the model and the view intricately connected. Then when you need to introduce other views you may be forced to perform extensive redesign. To avoid this you should practice model-view separation as a general principle, right from the start of a design:

Touch of Methodology: Model-View Separation Principle

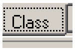
In designing the architecture of a software system, keep the coupling between model elements and view elements to a minimum.

If we use an event-driven model this rule goes well with a clear separation of publishers and subscribers. Both the subscribers and the publishers will interact with the view, but in a decoupled way:

- *Publishers* trigger events which may immediately update the view, typically in minor ways; for example the cursor may change shape when it enters a certain window, and a button usually changes its aspect when it has been pressed like the Class button on the right (if you look carefully).



Not pressed

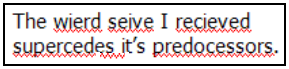


Pressed

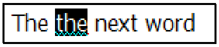
- *Subscribers* catch events (of event types to which they are subscribed), and process them. The processing may update the view.

Note that the publisher-subscriber and model-view divisions are orthogonal: both publishers and subscribers may need to interact with the model as well as with the views, as we can see in the example of a text processing system:

- The need for a publisher to trigger an event may be due to something that happens in a view — a user moves the mouse or clicks a button — or in the model, as when the spell checker detects a misspelled word and a view highlights it.
- The processing of an event by a subscriber will often cause modifications both to the model and to the view. For example if the user has selected a certain text and then presses the Delete key, the effect must be both to remove the selected part from the representation of the text kept internally by the system (model) and to update the display so that it no longer shows that part (view).


The wierd seive I recieved supercedes it's predocessors.

Flagging spelling errors

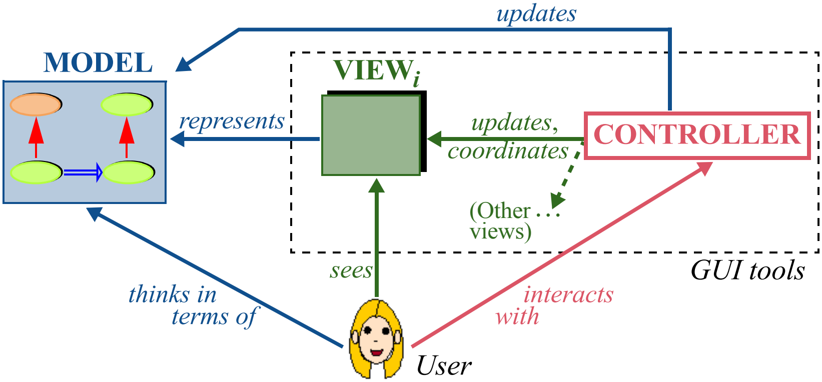

The the next word

Selecting text for deletion

Model-View-Controller

A particularly interesting scheme for GUI design is "Model-View-Controller" or MVC. The role of the third element, Controller, is to direct the execution of an interactive session; this may include creating and coordinating views.

Each of the three parts communicates with the other two:



MVC structure

The presence of a controller provides further separation between the model and the views. (Remember that there may be more than one view, hence "VIEW$_i$" in the figure.) The controller handles user actions, which may lead to updates of the view, the model, or both.

As before, a view provides a visual representation of the model or part of it.

The system designer may assume that *users understand the model*: using a text processing system, I should know about fonts, sections and paragraphs; playing a video game, I should have a feel for rockets and spaceships. A good system enables its users to *think* in terms of the model: even though what I see on the screen is no more than a few pixels making up some circular shape, I think of it as a flying vessel. The controller enables me to act on these views, for example by rolling my mouse wheel to make the vessel fly faster; it will then update both the model, by calling features of the corresponding objects to change their attributes (speed, position), and the view, by reflecting the effect of these changes in the visual representation.

The MVC paradigm has had a considerable influence on the spread of graphical interactive applications over the past decades. We will see at the end of this chapter that by taking the notion of event-driven design to its full consequences we can get the benefits of MVC but with a simpler architecture, bypassing some of the relations that populate the figure on the previous page.

That figure provides an opportunity for a side comment serving as general advice. Too often in presentations of software concepts you will find impressive diagrams with boxes connected by arrows but little specification of what they mean (their *semantics*). The last figure uses labels such as "*represents*" and "*updates*" to make the semantics clear. (Unlabeled arrows reflect standard conventions for client and inheritance links.) A picture is *not* worth any number of words if it is just splashes of color. Do not succumb to the lure of senseless graphics; assign precise semantics to each symbol you use, and state it explicitly.

18.4 THE OBSERVER PATTERN

Before we review what will be the definitive scheme for event-driven design (at least for the kind of examples discussed in this chapter), let us explore a well-known *design pattern*, "Observer", which also addresses the problem.

About design patterns

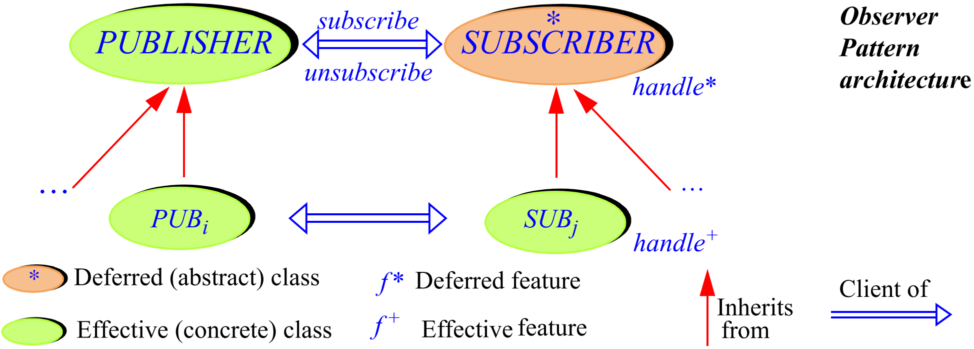
A design pattern is a standardized architecture addressing a certain class of problems. Such an architecture is defined by typical classes that must be part of the solution, their role, their relations — who inherits from whom, who is a client of whom — and instructions for customizing them as the problem varies. Design patterns emerged in the mid-nineties as a way to record and catalog design solutions, also known as "*best practices*", that good programmers had devised over the years, often reinventing them independently.

A couple dozen of these patterns, Observer among them, are widely documented and taught; hundreds more have been described or proposed.

Observer basics

As a general solution for event-driven design, Observer is actually not very good; we will analyze its limitations. But you should know about it anyway for several reasons: it is a kind of classic; it elegantly takes advantage of O-O techniques such as polymorphism and dynamic binding; it may be the best you can do in a language that does not support such notions as agents, genericity and tuples; and it provides a good basis for moving on to the more reasonable solution studied next.

The following figure illustrates a typical Observer architecture. *PUBLISHER* and *SUBSCRIBER* are two general-purpose classes, not specifically dependent on your application; PUB_i and SUB_j stand for typical publisher and subscriber classes in your application.



Observer Pattern architecture

Although both *PUBLISHER* and *SUBSCRIBER* are intended to serve as ancestors to classes doing the actual job of publishing and handling events, only *SUBSCRIBER* need be deferred; its deferred procedure *handle* will define, as effected in each concrete subscriber class SUB_j, how subscribers handle events. *PUBLISHER* needs no such deferred feature.

We could still make PUBLISHER deferred to prohibit direct instantiation. For a refresher on these concepts see "Deferred classes and features", 16.5, page 565.

As noted, we may say that the subscribers "observe" the publishers, standing on alert for any messages from them (hence the name of the pattern), and that the publishers are the "subjects" of this observation. You will similarly encounter, in the pattern literature, other names for the key features: "attach" for *subscribe*, "detach" for *unsubscribe*, "notify" for *publish*, "update" for *handle*.

The publisher side

Class *PUBLISHER* describes the properties of a typical publisher in charge of an event type — meaning it can trigger events of that type, through the procedure *publish*. The main data structure is a list of *subscribers* to the event type.

```
note
    what: ["Objects that can publish events, all of the same type,
            monitored by subscribers"]
class
    PUBLISHER
feature {SUBSCRIBER} -- Status report
    subscribed (s: SUBSCRIBER): BOOLEAN
            -- Is s subscribed to this publisher?
        do
            Result := subscribers.has (s)
        ensure
            present: has (s)
        end
feature {SUBSCRIBER} -- Element change
    subscribe (s: SUBSCRIBER)
            -- Make s a subscriber of this publisher.
        do
            subscribers.extend (s)
        ensure
            present: subscribed (s)
        end
    unsubscribe (s: SUBSCRIBER)
            -- Set s to be no longer a subscriber of this publisher.
        do
            subscribers.remove_all_occurrences (s)
        ensure
            absent: not subscribed (s)
        end
    publish (args: LIST [ANY])            -- Argument Scheme 1
            -- Publish event to subscribers.
        do
            … See below …
        end
feature {NONE} -- Implementation
    subscribers: LINKED_LIST [SUBSCRIBER]
        -- Subscribers subscribed to this publisher's event.
end
```

See next about publish, the type of its argument, and its "Argument Scheme".

Procedure *publish* will notify all subscribers that an event (of the event type for which the publisher is responsible) has occurred. We will find it easier to write it after devising the class representing a typical subscriber.

The implementation allows calling *subscribe* twice for the same subscriber; then (see *publish* below) the subscriber will execute the subscribed action twice for each event — most likely not the desired effect. To avoid this we could wrap the body of *subscribe* in **if not** *subscribed* (*s*) **then** … **end**, but then a linked list is no longer efficient since *has* requires a traversal. While not critical to the present discussion, this matter must be addressed for any actual use of the pattern; it is the subject of an exercise at the end of this chapter.

→ *"Efficient Observer", 18-E.3, page 697.*

Apart from *subscribers*, meant for internal purposes only and hence secret (exported to *NONE*), the features are relevant to subscriber objects but not to any others; they are hence exported to *SUBSCRIBER*. (As you remember, this means they are also exported to the descendants of this class, which need the ability to *subscribe* and *unsubscribe*.) As a general rule, it is a good idea to export features selectively when they are only intended for specific classes and their descendants. Better err on the side of restrictiveness to avoid mistakes caused by classes calling features that are none of their business; it is easy to ease the restrictions later if you find that new classes need the features.

← *"Overall inheritance structure", 16.10, page 586.*

The subscriber side

```
note
    what: "Objects that can register to handle events of a given type"
deferred class
    SUBSCRIBER
feature -- Element change
    subscribe (p: PUBLISHER)
            -- Subscribe to p.
        do
            p.subscribe (Current)
        ensure
            present: p.subscribed (Current)
        end
    unsubscribe (p: PUBLISHER)
            -- Ensure that this subscriber is not subscribed to p.
        do
            p.unsubscribe (Current)
        ensure
            absent: not p.subscribed (Current)
        end
feature {NONE} -- Basic operations
    handle (args: LIST [ANY])                 -- Argument Scheme 1
            -- React to publication of one event of subscribed type
        deferred
        end
end
```

See below about the "Argument Scheme" and the type of args.

This class is deferred: any application class can, if its instances may need to act as subscribers, inherit from *SUBSCRIBER*. We will call such descendants “subscriber classes” and their instances “subscribers”.

To subscribe to an event type, through the corresponding publisher *p*, a subscriber executes *subscribe* (*p*). Note how this procedure (and, similarly, *unsubscribe*) uses the corresponding feature from *PUBLISHER* to subscribe the current object. That was one of the reasons for exporting the *PUBLISHER* features selectively: it would be useless for a subscriber class or one of its clients to use *subscribe* from *PUBLISHER* directly, since subscribing only makes sense if you provide the corresponding *handle* mechanism; the feature of general interest is the one from *SUBSCRIBER*. (This also justifies using the same names for the features in the two classes, which keeps the terminology simple and causes no confusion since only the *SUBSCRIBER* features are widely exported.)

> Procedure *unsubscribe* removes an observer from the attention of the corresponding publisher. To avoid *memory leaks*, do not forget to call it when a subscriber no longer needs its subscription. This recommendation also applies to other architectural techniques and is further discussed below.

→ *“Subscriber discipline”, 18.6, page 690.*

Each subscriber class will provide its own version of *handle*, describing how it handles an event. The basic idea of *handle* is simple: just call the desired operation, passing it the event arguments if any.

There is, however, an unpleasant part: ensuring that the operation is getting arguments under the right types. The reason is that we tried to make *PUBLISHER* and *SUBSCRIBER* general, and so had to declare for *args*, representing the event arguments in both *publish* and *handle*, a completely general type: *LIST* [*ANY*]. But then *handle* has to force (“*cast*”) the right type and number of arguments.

← *Casting was discussed in “Uncovering the actual type”, 16.13, page 599.*

Assume for example that the event type declares two arguments of respective types *T* and *U*; we want to process each event by calling a routine *op* (*x*: *T*; *y*: *U*). We have to write *handle* as follows:

```
handle (args: LIST [ANY])                      -- Argument Scheme 1
          -- React to publication of event by performing op on the arguments.
     do
          if args.count >= 2 and then
               (attached {T} args.item (1) as x) and
                (attached {U} args.item (2) as y)
          then
               op (x, y)
          else
               -- Do nothing, or report error
          end
     end
```

The Object Tests make sure that the first and second elements of the *args* list are of the expected types, and bind them to *x* and *y* within the **then** clause.

← *“Uncovering the actual type”, 16.13, page 599.*

The only way to avoid this awkward run-time testing of argument types would be to specialize *PUBLISHER* and *SUBSCRIBER* by declaring the exact arguments to *publish* and *subscribe*, for example

```
publish (x: T; y: U)                    -- Argument Scheme 2
```

and similarly for *handle* in *SUBSCRIBER*. This loses the generality of the scheme since you cannot use the same *PUBLISHER* and *SUBSCRIBER* classes for event types of different signatures. Although it is partly a matter of taste, I would actually recommend this “Argument Scheme 2” if you need to use the Observer pattern, because it will detect type errors — a publisher passing the wrong types of arguments to an event — at compile time, where they belong.

With *handle* as written on the previous page, you will only find such errors at run time, through the tests on the size and element types of *args*; that is too late to do anything serious about the issue, as reflected by the rather lame “Do nothing, or report error” above: doing nothing means ignoring an event (is that what we want, even if the event is somehow deficient since it does not provide the right arguments?); and if we report an error, report it to whom? The message should be for the developers — us! — but will be displayed to the end users.

It was noted in the discussion of object test that this mechanism should generally be reserved for objects coming from the outside, rather than those under the program’s direct control, for which the designer is in charge of guaranteeing the right types statically. Here the publishing and handling of arguments belong to the same program; using object test just does not sound right.

It is actually possible to obtain a type-safe solution by making classes *PUBLISHER* and *SUBSCRIBER* generic; the generic parameter is a tuple type representing the signature of the event type (that is to say, the sequence of its argument types). That solution will appear in the final publish-subscribe architecture below (“Event Library”). We will not develop it further for the Observer pattern because it relies on mechanisms — tuple types, constrained genericity — that are not all available in other languages: if you are programming in Eiffel, which has them, you should use that final architecture (relying on agents), which is better than an Observer pattern anyway and is available through a prebuilt library. It is a good exercise, however, to see how to improve Observer through these ideas; try it now on the basis of the hints just given, or wait until you have seen the solution below.

→ *“Type-safe Observer”, 18-E.4, page 698.*

Publishing an event

The only missing part of the Observer pattern's implementation is the body of the *publish* procedure in *PUBLISHER*, although I hope you have already composed it in your mind. This is where the pattern gets really elegant:

```
publish (args: … Argument Scheme 1 or 2, see above discussion …)
        -- Publish event to subscribers.
    do
        -- Ask every subscriber in turn to handle the message:
        from subscribers.start until subscribers.after loop
            subscribers.item.handle (args)
            subscribers.forth
        end
    end
```

← *To be inserted in class PUBLISHER, page 680.*

With "Argument Scheme 1", *args* is of type *LIST* [*ANY*]; with "Argument Scheme 2", the declaration will specify the exact expected types.

← *Page 683.*

The highlighted instruction takes advantage of polymorphism and dynamic binding: *subscribers* is a polymorphic container; each item in the list may be of a different *SUBSCRIBER* type, characterized by a specific version of *handle*; dynamic binding ensures that the right version is called in each case. A festival of the best practices in object-oriented architecture!

← *"Polymorphic data structures", page 560.*

Assessing the Observer pattern

The Observer pattern is widely known and used; it is an interesting application of object-oriented techniques. As a general solution to the publish-subscribe problem it suffers from a number of limitations:

- The argument business, as discussed, is unpleasant, causing a dilemma between two equally unattractive schemes: awkward, type-unsafe run-time testing of arguments, and specific, quasi-identical *PUBLISHER* and *SUBSCRIBER* classes for every event type signature.
- Subscribers directly subscribe to publishers. This causes undesirable coupling between the two sides: subscribers should not have to know which part of an application or library triggers certain events. What we miss here is an intermediary — a kind of broker — between the two sides. The more fundamental reason is that the design misses a key abstraction: the notion of event type, which it merges with the notion of publisher.
- With a single general-purpose *PUBLISHER* class, a subscriber may register with only one publisher; with that publisher, it can register only one action, as represented by *handle*; as a consequence it may subscribe to only one type of event. This is severely restrictive. An application component should

be able to register various operations with various publishers. It is possible to address this problem by adding to *publish* and *handle* an argument representing the publisher, so that subscribers can discriminate between publishers; this solution is detrimental to modular design since the handling procedures will now need to know about all events of interest. Another technique is to have several independent publisher classes, one for each type of event; this resolves the issue but sacrifices reusability.

- Because publisher and subscriber classes must inherit from *PUBLISHER* and *SUBSCRIBER*, it is not easy to connect an *existing* model to a new view without adding significant glue code. In particular, you cannot directly reuse an existing procedure from the model (*op* in our example) as the action to be registered by a subscriber: you have to fill in the implementation of *handle* so that it calls that procedure, with the arguments passed by the publisher.

← op was used on page 682.

- The previous problem gets worse in languages without multiple inheritance. *PUBLISHER* and *SUBSCRIBER*, intended to be inherited by publisher and subscriber classes, both need effective features: respectively *publish*, with its fundamental algorithm, and *subscribe*. In languages that do not support multiple inheritance from classes with effective features, this prevents publisher and subscriber classes from having other parents as may be required by their role in the model. The only solution is to write special publishers and suppliers — more glue code — and make them clients of the corresponding model classes.
- Note finally that the classes given above already correct some problems that arise with standard implementations of the Observer pattern in the literature. For example the usual presentation binds a subscriber to a publisher at *creation* time, using the publisher as an argument to the observer's creation procedure. Instead, the above implementation provides a *subscribe* procedure in *SUBSCRIBER*, to bind the observer to a specific publisher when desired; so at least you can later unsubscribe, and re-subscribe to a different publisher.

All these problems have not prevented designers from using Observer successfully for many years, but they have two serious consequences. First, the resulting solutions lack flexibility; they may cause unnecessary work, for example writing of glue code, and unnecessary coupling between elements of the software, which is always bad for the long-term evolution of the system. Second, they are not *reusable*: each programmer must rebuild the pattern for every system that needs it, adapting it to the system's particular needs.

The preceding assessment of "Observer" is an example of how one may analyze a proposed *software architecture*. You may get some inspiration from it when you are presented with possible design alternatives. The criteria are always the same: reliability (decreasing the likelihood of bugs), reusability (minimizing the amount of work to integrate the solution into a new program), extendibility (minimizing adaptation effort when the problem varies), and simplicity.

→ See "Touch of Methodology: Assessing software architectures", page 695.

18.5 USING AGENTS: THE EVENT LIBRARY

We are now going to see how, by giving the notion of event type its full role, we can obtain a solution that removes all these limitations. It is not only more flexible than what we have seen so far, and fully reusable (through a library class which you can use on the sole basis of its API); it is also much simpler. The key boost comes from the agent and tuple mechanisms.

Basic API

We focus on the essential data abstraction resulting from the discussion at the beginning of this chapter: event type. We will no longer have *PUBLISHER* or *SUBSCRIBER* classes, but just one class — yes, a single class solves the entire problem — called *EVENT_TYPE*.

Fundamentally, two features characterize an event type:

- Subscribing: a subscriber object can register its interest in the event type by subscribing a specified action, to be represented by an agent.
- Publishing: triggering an event.

We benefit from language mechanisms to take care of the most delicate problems identified in the last section:

- Each event type has its own signature. We can define the signature as a tuple type, and use it as the generic parameter to *EVENT_TYPE*.
- Each subscription should subscribe a specific action. We simply pass this action as an agent. This allows us in particular to reuse an existing feature from the business model.

These observations are enough to give the interface of the class:

```
note
    what: "Event types, allowing publishing and subscribing"
class EVENT_TYPE [ARGUMENTS -> TUPLE] feature
    publish (args: ARGUMENTS)
            -- Trigger an event of this type.

    subscribe (action: PROCEDURE [ANY, ARGUMENTS])
            -- Register action to be executed for events of this type.

    unsubscribe (action: PROCEDURE [ANY, ARGUMENTS])
            -- De-register action for events of this type.
end
```

Class interface only. The implementations of publish and subscribe appear below.

If you are an application developer who needs to integrate an event-driven scheme in a system, the above interface — for the class as available in the Event

Library — is all you need to know. Of course we will explore the implementation too, as I am sure you will want to see it. (It will actually be more fun if you try to devise it yourself first.) But for the moment let us look at how a typical client programmer, knowing only the above, will set up an event-driven architecture.

Using event types

The first step is to define an event type. This simply means providing an instance of the above library class, with the appropriate actual generic parameters. For example, you can define

```
left_click: EVENT_TYPE [ TUPLE [x: INTEGER; y: INTEGER] ]          [1]
        -- Event type representing left-button click events
    once
        create Result
    end
```

The function *left_click* returns an object representing the desired event type.

Remember, we do not need an object per event; that would be a waste of space. We only need an object per event *type*, such as left-click. Because this object must be available to several parts of the software — publishers and subscribers — the execution needs just one instance; this is an opportunity to use a "once routine" (one of the very few Eiffel mechanism that we had not seen yet, and the last one for this book). As the name suggests, a once routine (marked **once** instead of **do** or **deferred**) has its body executed at most once, on its first call if any; subsequent calls will not execute any code and, in the case of a function as here, they will return the value computed by the first. One of the advantages is that you do not need to worry about when to create the object; whichever part of the execution first uses *left_click* will (unknowingly) do it.

We will see shortly where the event type declaration [1] should appear; until then, let us assume that subscriber and publisher classes both have access to it.

To trigger an event, a publisher — for example a GUI library element that detects a mouse click — simply calls *publish* on this event type object, with the appropriate argument tuple; in our example:

```
left_click.publish ([your_x, your_y])
```

On the subscriber side things are equally simple; to subscribe an action represented by a procedure *p* (*x*, *y*: *INTEGER*), it suffices to use

```
left_click.subscribe (agent p )          [2]
```

This scheme has considerable flexibility, achieved in part through the answer to the pending question of where to declare the event type:

- If you want to have a single event type published to all potential subscribers, just make it available to both publisher and subscriber classes by putting its declaration [1] in a "facilities" class to which they all have access, for example by inheriting from it.
- Note, however, that the event type is just an ordinary object, and the corresponding features such as *left_click* just ordinary features that may belong to any class. So the publisher classes — for example classes representing graphical widgets, such as *BUTTON*, in a library such as EiffelVision — may declare *left_click* as one of their features. Then the scheme for a typical subscription call becomes

```
your_button. left_click.subscribe (agent p)                [3]
```

This allows a subscriber to monitor — "observe" or "listen to", in Observer pattern terminology — mouse events from one particular button of the GUI. Such a scheme implements the notion of **context** introduced earlier; here the context is the button.

←*"Contexts", page 673.*

Whenever the context is relevant — meaning whenever subscribers do not just subscribe to an event type as in [2], but to events occurring in a context, as in [3] — the proper architectural decision is to declare the relevant event types in the corresponding context classes. The declaration of *left_click* [1] becomes part of a class *BUTTON*. It remains a **once** function, since the event type is common to all buttons of that kind; the event type object will be created on the first *subscribe* or *publish* call (whichever comes first). If left-click is relevant for several kinds of widget — buttons, windows, menu entries … — then each of the corresponding classes will have an attribute such as *left_click*, of the same type. The **once** mechanism ensures, as desired, that there is one event type object — more precisely: at most one — for each of these widget types.

This is the technique used by EiffelVision.

So we get the appropriate flexibility, and can tick off the last remaining item ("*It should be possible to make events dependent or not on a context*") on our list of requirements for a publish-subscribe architecture:

←*"Publishers and subscribers", page 674.*

- For events that are relevant independently of any context information, declare the event type in a generally accessible class.)
- If a context is needed, declare the event type as a feature of a class representing the applicable contexts; it will be accessible at run time as a property of a specific context object.

In the first case, the event type will have at most one instance, shared by all subscribers. In the second case, there will be at most one instance for each context type for which the event type is relevant.

Event type implementation

Now for the internal picture: we still have to see the implementation of *EVENT_TYPE*. It is similar to the above implementation of a *PUBLISHER*. A secret feature *subscribers* keeps the list of subscribers. Its signature is now

```
subscribers: LINKED_LIST [PROCEDURE [ANY, ARGUMENTS]]
```

(where, as before, *LINKED_LIST* is a naïve structure but sufficient for this discussion; for a better one, look up the actual class text of *EVENT_TYPE* in the Event Library, or do the exercise). The items we store in the list are no longer "subscribers", a notion that the architecture does not need any more, just agents. The type of every such agent, *PROCEDURE* [*ANY*, *ARGUMENTS*], indicates that the agent represents a procedure with an argument of the tuple type *ARGUMENTS*, as defined for the class. This considerably improves the type safety of the solution over what we saw previously: mismatches will be caught at compile time as bad arguments to *subscribe*.

→ *"Efficient Observer", 18-E.3, page 697.*

For *subscribe* it suffices (in the "naïve" implementation) to perform

```
subscribe (action: PROCEDURE [ANY, ARGUMENTS])
        -- Register action to be executed for events of this type.
    do
        subscribers.extend (action)
    ensure
        present: subscribers.has (action)
    end
```

The use of *ARGUMENTS* as the second generic parameter for the *PROCEDURE* type of *action* ensures compile-time rejection of procedures that do not take arguments of a matching type.

To publish an event, we traverse the list and call the corresponding agents. This is in fact the same code as in class *PUBLISHER* for the Observer pattern, although *args* is now of a more appropriate type, *ARGUMENTS*:

← *Page 684.*

```
publish (args: ARGUMENTS)
        -- Publish event to subscribers.
    do
        -- Trigger an event of this type.
        from subscribers.start until subscribers.after loop
            subscribers.item.call (args)
            subscribers.forth
        end
    end
```

Any argument to the agent feature *call* must be a tuple; this is indeed the case since *ARGUMENTS* is constrained to be a tuple type.

The solution just described is at the heart of the "Event Library", and also of the EiffelVision GUI library; it is widely used for graphical applications, some small and some complex, including the EiffelStudio environment itself.

The class includes a few more details, which it is a good idea to peruse:

Program Reading Time!
Event types

Read the *EVENT_TYPE* library class and make sure you understand all of it.

18.6 SUBSCRIBER DISCIPLINE

If you apply any of the techniques of this chapter, from the crude Observer pattern to the agent-based mechanism, you should be aware of a performance issue which can lead to potentially disastrous "memory leaks", but is easy to avoid if you pay attention to the subscribers' behavior:

***Touch of Methodology*:**
Do not forget to unsubscribe

If you know that after a certain stage of system execution a certain subscriber will no longer need to be notified of events of a certain event type, do not forget to include the appropriate call to *unsubscribe*.

Why this rule? The problem is memory usage. It is clear from the implementation of *subscribe* — both the version from *PUBLISHER* in the Observer pattern and the version from *EVENT_TYPE* in the Event Library approach — that registering a subscriber causes the publisher to record, in its list *subscribers*, a reference to the subscriber object. In a GUI application the publisher belongs to a view, and the subscriber to the model. So a view object retains a reference to a model object, which itself may directly and indirectly refer to many other model objects (say planes, flights, schedules and so on in our flight control example).

← *Page 680.*

← *Page 686.*

But then it becomes impossible — unless the view objects themselves go away — to get rid of any such model object even if the computation does not need it any more. In a modern environment with garbage collection, the GC will never be able to reclaim the objects as long as others refer to them. If memory reclamation is manual (as in C and C++ environments), the situation is even worse. In either case we have a source of "memory leak": as execution fails to return unneeded space, memory occupation continues to grow.

Hence the above rule: subscribing is great, but once you no longer need a service do not forget — as with free magazines and catalogs, if you do not want to see your mailbox inexorably fill up — to unsubscribe.

Methodological rules are never as effective as tools and architectures that guarantee the desired goal. In this case, however, there is no obvious way to enforce unsubscription, other than through this methodological advice.

When you subscribe an agent and want to be able to unsubscribe later, you should use a variable representing the agent:

```
handler := agent p                                                    [4]
left_click.subscribe (handler)
…                 -- Then, when the time comes to unsubscribe:
left_click.unsubscribe (handler)
```

Subscribing through a variable, rather than using the agent directly as in the earlier form *left_click*.*subscribe* (**agent** *p*), ensures that the unsubscription applies to the same agent object (unlike *left_click*.*unsubscribe* (**agent** *p*) which would apply to a new object). If you have subscribed a given *handler* more than once to a given event type, *unsubscribe* (internally using *remove_all*) removes all such subscriptions.

18.7 SOFTWARE ARCHITECTURE LESSONS

The designs reviewed in this chapter prompt some general observations about software architecture.

Choosing the right abstractions

The most important issue in software design, at least with an object-oriented approach, is to identify the right classes — data abstractions. (The second most important issue is to identify the relations between these classes.)

In the Observer pattern, the key abstractions are "Publisher" and "Subscriber". Both are useful concepts, but they turn out to yield an imperfect architecture; the basic reason is that these are not good enough abstractions for the publish-subscribe paradigm. At first sight they would appear to be appropriate, if only because they faithfully reflect the two words in the general name for the approach. What characterizes a good data abstraction, however, is not an attractive name but a set of consistent features. The only significant feature of a publisher is that it publishes events from a given event type, and the only significant feature of a subscriber is that it can subscribe to events from a given event type. That is a bit light.

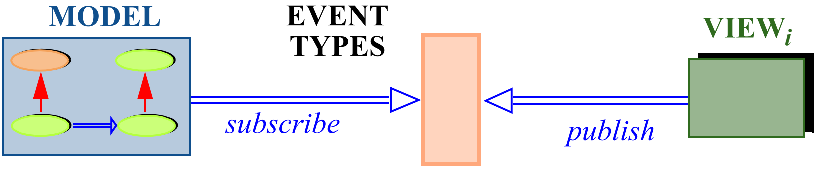
The more significant concept, not recognized by the Observer design, is the notion of event type. This is a clearly recognizable data abstraction with several characteristic features: commands to publish and subscribe events, and the notion of argument (which could be given more weight through a setter command and a query). So it meets the criteria.

By treating *EVENT_TYPE* as the key abstraction, yielding the basic class of the final design, we avoid forcing publisher and subscriber classes to inherit from specific parents. A publisher is simply a software element that uses *publish* for some event type, and a subscriber an element that uses *subscribe*.

MVC revisited

One of the consequences of the last design is to simplify the overall architecture suggested by the Model-View-Controller paradigm. The Controller part is "glue code" and we should keep it to the strict minimum.

EVENT_TYPE provides the heart of the controller architecture. In a simple scheme it can actually be sufficient, if we let elements of the model subscribe directly to events:



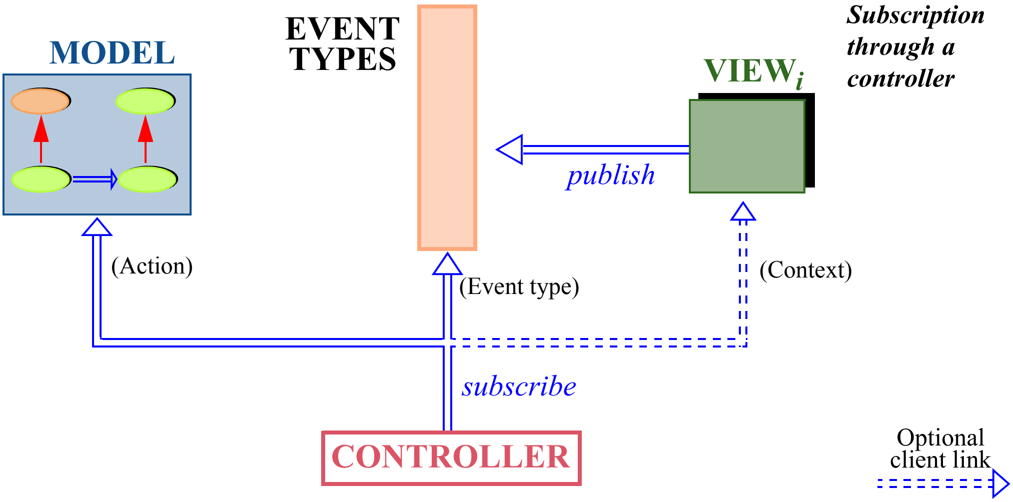
Direct subscription

(The double arrows represent, as usual, the client relation, used here to implement the more abstract relations of the general MVC picture.) In this scheme there is no explicit controller component.

While the model does not directly know about the view (if it does not use contexts), it does connect to specific event types. This setup has both limitations and advantages:

- On the negative side, it can make it harder to change views: while we are not limited to a single view, any new view should trigger the same events. This assumes that the various views are not entirely dissimilar; for example one could be a GUI and another a Web interface.
- It has, on the other hand, the merit of great simplicity. Model elements can directly specify which actions to execute for specific events coming from the interface. There is essentially no glue code.

This scheme is good for relatively simple programs where the interface, or at least the interface style, is known and stable. For more sophisticated cases, we may reintroduce an explicit controller, taking the task of event type subscription away from the model:



Subscription through a controller

The controller is now a separate part of the software. Its job is to subscribe elements of the model to event types; it will have connections both to:

- The model since the arguments to *subscribe* are actions to be subscribed, and these must be agents built from the mechanisms of the model.
- The view, if contexts are used. The figure shows this as an optional client link.

← *As per style [3], page 688.*

This solution achieves complete uncoupling between model and view; in a typical application the controller will still be a small component, achieving the goal of using as little glue code as possible.

The model as publisher

In the GUI schemes seen so far all events come from the GUI, normally through the mechanisms of a library, and are processed by the model. In other words the views are publishers, and model elements are subscribers.

It is possible to extend the scheme to let the model publish too. For example if an element of the GUI such as pie chart reflects a set of values which the model may change, the corresponding model elements may publish change events. Views become subscribers to such events.

This possibility is easy to add to the second scheme, subscription through a controller. The controller will now act as a fully bidirectional broker, receiving events from the views for processing by the model and the other way around.

This solution, which adds complexity to the controller, is only useful in the case of multiple views.

Invest then enjoy

Common to the two architectures we have seen, Observer and Event Library, is the need to subscribe to event types prior to processing them.

It is possible for subscribers to subscribe and unsubscribe at any time; in fact, with the Event Library solution, the program can create new event types at any stage of the computation. While this flexibility can be useful, the more typical usage scenario clearly divides execution into two steps:

- During initialization, subscribers register their actions, typically coming from the model.
- Then starts execution proper. At this stage the control structure becomes event-driven: execution proceeds as publishers trigger events, which (possibly depending on the contexts) cause execution of the subscribed model actions.

(So from the order of events it really is the "Subscribe-Publish" paradigm.) Think of the life story of a successful investor: set up everything, then sit back and prosper from the proceeds.

You may remember a variant of the same general approach, the "compilation" strategy that worked so well for topological sort: first translate the data into an appropriate form, then exploit it.

← *"Interpretation vs compilation", page 542.*

Assessing software architectures

The key to the quality of a software system is in its architecture, which covers such aspects as:

- The choice of classes, based on appropriate data abstractions.
- Deciding which classes will be related, with the overall goal of minimizing the number of such links (to preserve the ability to modify and reuse various parts of the software independently).
- For each such link, deciding between client and inheritance.
- Attaching features to the appropriate classes.
- Equipping classes and features with the proper contracts.
- For these contracts, deciding between a "demanding" style (strong preconditions, making the client responsible for providing appropriate values), a "tolerant" style (the reverse), or an intermediate solution.
- Removing unneeded elements.

- Avoiding code duplication and removing it if already present; techniques involve inheritance (to make two or more classes inherit from an ancestor that captures their commonality) as well as genericity, tuples and agents.
- Taking advantage of known design patterns.
- Devising good APIs: simple, easy to learn and remember, equipped with the proper contracts.
- Ensuring consistency: throughout the system, similar goals should be achieved through similar means. This governs all the aspects listed so far; for example, if you use inheritance for a certain class relationship, you should not use the client relation elsewhere if the conditions are the same. Consistency is also particularly important for an API, to ensure that once programmers have learned to use a certain group of classes they can expect to find similar conventions in others.

Such tasks can be carried out to improve existing designs, an activity known as *refactoring*. It is indeed a good idea to look at existing software critically, but prevention beats cure. The best time to do design is the first.

Whether as initial design or as refactoring, work on software architecture is challenging and rewarding; the discussion in this chapter — and a few others in this book, such as the development of topological sort — give an idea of what it involves. The criteria for success are always the same:

***Touch of Methodology*:**
Assessing software architectures

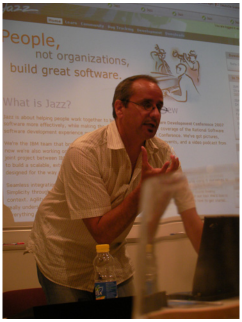
When examining possible design solutions for a given problem, discuss alternatives critically. The key criteria are: reliability, extendibility, reusability and simplicity.

18.8 FURTHER READING

Reenskaug

Trygve Reenskaug, MVC papers at heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

> Trygve Reenskaug, a Norwegian computer scientist, introduced the Model-View-Controller pattern while at Xerox PARC (the famed Palo Alto Research Center) in 1979. The page listed contains a collection of his papers on the topic. I find his original 1979 MVC memo (barely more than a page) still one of the best presentations of MVC.



Gamma (2007)

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns*, Addison-Wesley, 1994.

The classic text on design patterns. Contains the standard description of Observer, along with many others, all in C++.

Bertrand Meyer: *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*, in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271. Available online at se.ethz.ch/~meyer/publications/lncs/events.pdf.

A significant part of the present chapter's material derives from this article, which analyzes the publish-subscribe pattern in depth, discussing three solutions: Observer pattern, .NET delegate mechanism, and the event library as presented above.

18.9 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Event-driven design, also called "publish-subscribe", leads to systems whose execution is driven by responses to events rather than by traditional control structures. The events are triggered by the software, often in reaction to external events. GUI programming is one of the important areas of application.
- The key abstraction in event-driven design is the notion of event type.
- *Publishers* are software elements that may trigger events of a certain event type. *Subscribers* are elements that request to be notified of events of a certain type by *registering* actions to be executed in response.
- In a system with one or more interfaces or "views", an important design guideline is to keep the views separate from the core or the application, known as the "model".
- The Model-View-Controller architecture interposes a "controller" between the model and the view to handle interactions with users.
- The Observer pattern addresses event-driven design by providing high-level classes *PUBLISHER* and *SUBSCRIBER*, from which publisher and subscriber classes must respectively inherit. Every subscriber class provides an *update* procedure to describe the action to be executed in response to specified events. Internally, each publisher object keeps a list of its subscribers. To trigger an event, it calls *update* on its subscribers; thanks to dynamic binding, each such call executes the desired version.
- Agents, constrained genericity and tuples allow a general solution to event-driven design through a single reusable class based on the problem's central abstraction: *EVENT_TYPE*.

- Software architecture is the key to software quality. Devising effective architectures and improving existing ones (refactoring) should be a constant effort, focused on simplicity and striving at reliability, extendibility and reusability.

New vocabulary

Application domain	**Argument (of an event)**	**Business model**
Catching (an event)	**Context (of an event)**	**Control (Windows)**
Controller	**Event**	**Event-driven**
Event type	**External event**	**Glue code**
Handle (an event)	**Model**	**MVC**
Publish (an event)	**Publish-Subscribe**	**Register**
Refactoring	**Signature (of event type)**	**Subscribe**
Trigger (an event)	**View**	**Widget**

18-E EXERCISES

18-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

18-E.2 Concept map

Add the new terms to the conceptual map devised for the preceding chapters.

← *Exercise "Concept map", 17-E.2, page 660.*

18-E.3 Efficient Observer

Choosing the appropriate representation of the subscribers list, adapt the implementation of the Observer pattern so that the following operations are all **O** (1): add a subscriber (doing nothing if it was already subscribed); remove a subscriber (doing nothing if it was not subscribed); find out if a potential subscriber is subscribed. The *publish* procedure, ignoring the time taken by subscribers' actual handling of the event, should be **O** (*count*) where *count* is the number of subscribers actually subscribed to the publisher. Overall space requirement for the *subscribers* data structure should be reasonable, e.g. **O** (*count*). (*Hint*: look at the various data structures of chapter 13 and at the corresponding classes in EiffelBase.) Note that this optimization also applies to the Event Library implementation.

← *"The observer pattern", 18.4, page 678..*

18-E.4 Type-safe Observer

Show that in implementing the Observer pattern a type scheme is possible that removes the drawbacks of both "Argument Scheme 1" and "Argument Scheme 2" by taking advantage, as in the last design of this chapter (Event Library), of tuple types and constrained genericity. Your solution should describe how the *PUBLISHER* and *SUBSCRIBER* classes will change, and also present a typical publisher and subscriber classes inheriting from these.

← *"The subscriber side", page 681..*

PART V:

Towards software engineering

In this one-chapter part, the final one before appendices, we take a look at what it takes to move from programming — the focus of the rest of this book — to the professional construction and operation of industrial-quality software development, otherwise known as software engineering.

19

Introduction to software engineering

There is more to software development than programming. This statement is not a paradox, but a recognition of all the factors that affect the success of a software project, and all the tasks, other than writing the program, that we must accordingly worry about. To take just a few examples:

- A program with a brilliant design may end up a failure if its user interface displeases the target audience.
- The best program is useless if it does not solve the right problem. Hence the need for a *requirements* task to capture user needs and decide on the system's precise functionality.
- Aside from technical aspects, projects must tackle *management* issues: setting and enforcing deadlines, organizing meetings and other communication between project members, defining the budget and controlling expenses.

These activities and others discussed in this chapter do not involve programming techniques, but if not taken care of properly they can destroy a project regardless of its technical qualities.

This is typical of what defines moving beyond *programming* to *software engineering*. In the previous chapters we have almost exclusively been concerned with programming, but the picture would be incomplete without a foray into the non-programming aspects of software engineering.

This is a wide-ranging and well-developed discipline. To cover it extensively would require another textbook. Fortunately, several already exist; you will find the references to some of the best in the "Further reading" section. The present chapter has more limited goals: to introduce a few non-programming aspects of software engineering, enough I hope to awaken your interest and make you want to learn further from the rest of the literature.

→ *Page 740.*

An important part of software engineering is the role of *tools*. A survey of some software engineering tools appeared in an earlier chapter.

← *Chapter 12.*

19.1 BASIC DEFINITIONS

The following broad-ranging definition will serve us well:

> ***Definition*: Software engineering**
>
> Software engineering is the set of techniques — including theories, methods, processes, tools and languages — for developing and operating production software meeting defined standards of quality.

Two important properties of software engineering captured by this definition are the restriction to production software, and the focus on quality.

Production software is operational software, intended to function in real environments to solve real problems. Software developed purely as an experiment, or "throw-away" programs used once and not further maintained, generally do not qualify, except if it they are a means towards some broader goal which belongs to software engineering proper. For example, an experiment to evaluate various possible algorithms may not qualify by itself, but this changes if it is performed as part of the development of a production system.

What characterizes production software is the combination of constraints that it must satisfy. They may include:

- Quality constraints (discussed next): for example the guarantee that the system will not crash, will deliver correct results and will perform fast.
- Size constraints: production systems may consist of thousands or tens of thousands of classes and other modules, and hundreds of thousands or millions of lines of code.
- Duration constraints: systems used in industry must often be maintained (that is to say, kept operational and regularly updated) over many years or even decades.
- Team constraints: such systems may involve large teams of developers, and large numbers of users; this raises specific management and communication problems.
- Impact constraints: these systems affect physical and human processes; in particular, if they do not function well, people may be affected — by trains not arriving in time, phones not working, salaries not paid, orders not delivered, or worse. This reinforces the emphasis on quality.

Quality is indeed at the center of software engineering concerns. The definition mentions "defined standards": quality is not just something that someone arbitrarily declares to be present or absent in a software process or product; it should be evaluated as objectively as possible.

The definition also talks of "developing and *operating*" software. Software construction cannot be hit-and-run: along with development you have to set up the actual usage (operation) of the software. Even the development part should not be understood as only the initial production of a releasable system: what comes afterwards is just as important. We have already encountered the technical term for this activity:

Definition: Software maintenance

The **maintenance** of software systems covers all development activities occurring after the first release of an operational version, such as: adaptation to new platforms and environments; correction of reported deficiencies; extensions (addition of new functionality); removal of unneeded functionality; quality improvement.

The term "maintenance" comes from other parts of engineering: think of maintenance for a car, of a coffee machine, of a house. As many people have pointed out, the analogy is misleading: a program does not deteriorate from repeated use; run your program ten, a thousand or a million times, and unlike a car whose tires will inexorably wear out it is still exactly the same program as the first time. As a software term, however, "maintenance" is here to stay. There is no problem in using it as long as this is based on a precise definition as above.

A jargon term will be useful for the discussion:

Stakeholder

A **stakeholder** of a software project is any person who can affect or be affected by the project and the quality of the resulting software.

This encompasses many people: developers, but also testers and other quality assurance personnel; project managers; future users of the system (or others on whom it may have an effect, including — the less pleasant part but definitely a possibility — those who will *not* be users because the system makes their current jobs obsolete); marketing and sales people who will have to find customers in the case of a product to be released to the world; trainers (who will teach users how to benefit from the product); corporate legal departments.

It is an important task of project management to identify all the stakeholders early, and to give due consideration to the needs and constraints of each.

19.2 THE DIAMO VIEW OF SOFTWARE ENGINEERING

To understand the challenges of software engineering, we may view the discipline as consisting of five parts of roughly equal importance. Programming is one of the components of one of these parts (the second part, "Implement"). As a mnemonic for this classification we may use the acronym DIAMO (although it is not an English word, only the prefix of one). The letters stand for Describe, Implement, Assess, Manage and Operate.

D

Describe: many software engineering activities involve understanding and specifying *problems* and systems; the goal here is not to build solutions, but to describe properties of such solutions. We may need the description *before* building the solutions, as in requirements analysis and design specifications; or *afterwards*, as in documentation.

I

Implement: this is the task of building the programs. It includes not just implementation ("programming" in a restricted sense of this term) but also *design*, the task of defining the high-level architecture of a system.

A

Assess: a large part of software processes is devoted to analyzing software. The products being assessed include not just programs but everything else that makes up software, in particular designs and documentation. The most common goal is to uncover deficiencies (or, conversely, to establish correctness); this is where you find such tasks as static program analysis, testing and debugging. It is not, however, all there is to assessment; in particular, an effective software organization often needs *quantitative* properties of both its products (size, complexity…) and its processes (time spent, costs); this is the purpose of *software metrics*.

M

Manage: any serious software project, even with just a few developers, requires management. The management part of software engineering addresses such tasks as communication within the development team — ever more challenging today, as many teams are geographically distributed across countries and continents —, scheduling tasks so that they will meet deadlines, ensuring smooth interaction with customers and other stakeholders, and handling the inevitable requests for change.

O

Operate: when everything has been analyzed, designed, implemented, tested and documented, you are not done yet. You still need to put the system into operation — a step known as *deployment* — and organize its successful operation. The deployment phase can, in industry, be a major undertaking; think of a bank installing software in thousands of automatic teller machines in many countries, with the need to adapt each installation to the local context (display language, security requirements, network connections) and to put in place a sustainable process of future system-wide upgrades.

Programming, of course, retains its fundamental role: no programs, no software engineering. All of the other activities are theoretically dispensable. In practice, any significant project must devote attention to each one of them.

19.3 COMPONENTS OF QUALITY

Quality, the central pursuit of software engineering, is a notion with many different facets, often called *factors* of software quality. Let us take a look at some of the most important ones.

Process and product

Discussions of software engineering address two complementary aspects:

- *Products*: outcomes of the development. The most obvious product is the source code, but software projects frequently add many others such as requirements and design documents, test data, project plans, documentation, installation procedures.
- *Process*: mechanisms used to obtain these products.

The number of errors in a delivered program is an example of a product issue. Whether the program is delivered on schedule is an example of a process issue.

> In each case the other aspect plays a role too: the process determines in part the introduction and removal of errors; and treating timely delivery as the principal goal may affect the product, for example through dropped functionality.

It is convenient to discuss the factors of software quality under three rubrics based on this distinction:

- *Process quality*, characterizing the effectiveness of the software development process.
- *Immediate product quality*, characterizing the adequacy of the product as delivered in a particular version.
- *Long-term product quality*, characterizing the future prospects of the software. In the world of software engineering, where projects may have a long life, this is just as important as the immediate picture.

We will now take a look at the major goals in each area, starting with the most visible property of a software project: immediate product quality. The discussion also includes some comments about why some other factors are less relevant. Two general notes about this review:

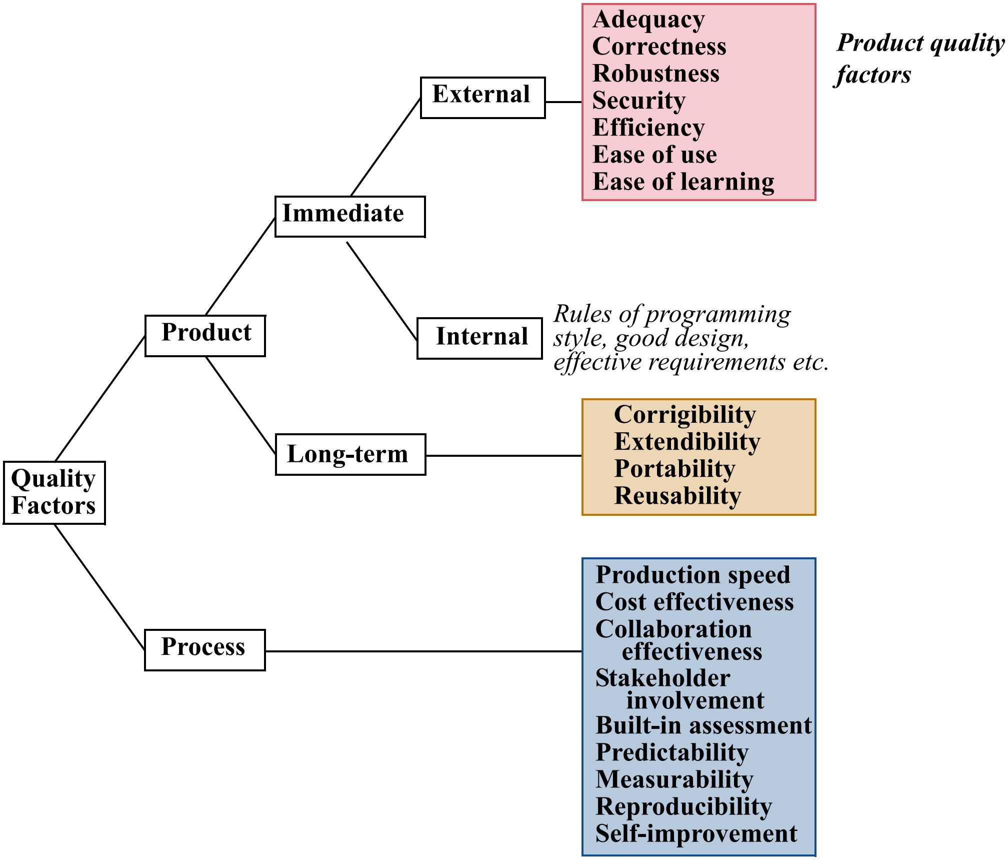
- No explicit definitions are given for self-explanatory quality factors ("ease of use", "ease of learning"). The corresponding terms will not appear in the "New vocabulary" list of this chapter.

→ Page 743.

- You will notice a certain relativism in the definitions: adequacy is satisfaction of *defined* needs, efficiency is *adequate* use of resources. This is not vagueness but in fact the reverse: definitions of software quality goals are only useful as long as they allow the product or process to be **assessed** against these goals. The definitions consequently assume that such goals have been clearly defined. This issue is not just academic: imagine that you are heading a software development project and that you track the number of remaining deficiencies ("bugs"). Should you authorize the release when the number reaches 1000, 500, 200, 0? (In a realistic setup you would have to distinguish between categories of bugs: show-stoppers, serious but non-critical deficiencies, minor issues such as user interface imperfections, missing "nice to have" functionalities that could be deferred to the next release and so on.) This question is essentially impossible to answer unless precise criteria have been stated in advance. We are back to the original definition of software engineering and its insistence on "meeting *defined* standards of quality".

← Page 702.

The following figure shows the overall classification for the quality factors to be reviewed now.



Immediate product quality

Product quality involves the following factors.

- **Adequacy**: satisfaction of defined user needs. In other words: does the software serve the right purposes for its user community? Other factors commonly cited in this area are *completeness* and *usefulness*, but both are less precise and are subsumed by "adequacy": no system ever has "complete" functionality, since someone will always think of another facility that would be nice to have; and "usefulness" is a subjective criterion unless you state precisely *to whom* and *for what needs* the system is, or not, useful enough.
- **Correctness**: to what extent the software functions as prescribed by the specification, in cases covered by the specification. This is clearly a fundamental requirement. It is just as clearly hard to achieve, not only because writing programs that meet a specification is hard, but also because writing the specification itself is tough too — you must think of all cases and end up with a document that is precise yet readable.

  An important consequence of this definition is that correctness is a relative notion. We can never say in absolute terms that a software system is correct or incorrect. We can only discuss its correctness *with respect to* a certain specification. In mathematical terms, correctness does not apply to a program but to a pair: [*program*, *specification*.]
- **Robustness**: how well the system reacts to erroneous cases of use, outside of the specification. That a user pressed the wrong button, a sensor malfunctioned or another program sent bad input is not a good excuse for the system to crash or produce wild results. Robustness assesses error handling and recovery mechanisms.
- **Security**: how well the system protects itself, its data, its users and any affected devices or people against attempts at hostile misuse. Unfortunately it is not just errors we need to worry about, as addressed by robustness; computer systems offer ready targets for people with all kinds of nasty intent, and you cannot write software, especially if it will be available over a network, without considering potential attacks.
- **Efficiency** (often called **performance**): adequate use of time, memory space, and other resources such as bandwidth if the system engages in network communication. "Adequate", not optimal: if your compiled program takes up one megabyte of memory, reducing this to 0.6 MB may be possible, but is not necessarily useful. If you expect your users to have plenty of memory, it is probably more productive to spend your time on other quality factors; but if you are running in a tightly constrained environment, for example with software for small-memory hand-held devices, such space optimization can become critical. What matters once again is to define objectives.

- **Ease of use**: the really difficult challenge here is to make the system easy to use for *various categories* of users. Much of the effort in "usability", as this is also called, goes into facilitating the task of complete novices. But it is just as important to help the experts — who, for example, do not want to go through the same repetitive sequences of clicking "OK" on various informational windows over and again, when they know exactly what to do — and to support the process of progressing from novice to expert. Each of us is a novice in some tools and an expert in others; and each of us was once a novice in each of the systems at which we are now an expert. Ease of use is also about defining that path and helping anyone who wants to travel it.
- **Ease of learning**: closely connected to the previous factor.

Long-term product quality

Some product qualities are of no immediate value to users of a system, but of much interest to those who commission or purchase it. If I am driving, I do not care that the software controlling the brakes or the air bag is easy to modify; I care that it works (an "immediate" factor). But if I am an executive in charge of managing software development or acquisition at Nissan or BMW I have to keep in mind the long-term picture: will the software be easy to upgrade if an improvement is requested? Can the version developed for sedans be transposed at reasonable cost to convertibles?

Descriptions of software products and software issues often talk about "the user"; the term has acquired almost mythical connotations. It is good to think of users, but stakeholders in user organizations also include others with a long-term perspective. The more general term "customer" is appropriate (whether or not the product is commercial) to cover both people using the products now and those interested in its past and future.

Long-term qualities, in the approximate order of when concerns will arise, include:

- **Corrigibility**: how easy it is to update the software to repair deficiencies (of correctness, robustness, security, ease of use…). One of the recipes for achieving corrigibility is *structure*: devising a modular architecture that is easy to understand and reflects the structure of the problem and its solution.

Also "correctibility".

- **Extendibility**: how easy it is to add functionality. Here too, structure is key; the object-oriented techniques we have learned — data abstraction, information hiding, classes, genericity, contracts, inheritance, dynamic binding, agents and so on — facilitate extension. Extendibility is a principal requirement of practical software development, as almost every system undergoes changes of its expected functionality. The reason for change may

Also "extensibility".

be that the initial requirements definition missed some functions; sometimes it is simply the consequence of initial success, as a useful system suggests ideas of what more it could do. A good software process must enforce a discipline on such changes, by defining strict procedures for examining new requests once initial requirements have been approved; but it cannot pretend that the need will not arise.

- **Portability**: how easy it is to transfer the software to other platforms. A "platform" here is a combination of computer architecture and operating systems, plus other resources that the system may need, such as a database management system. The IT industry has experienced considerable standardization in recent decades, making the construction of portable software more realistic than when dozens of incompatible computer brands populated the market. For general-purpose computing, the hardware scene is down to a few architectures (Pentium and compatible, Sparc, PowerPC), and the operating system world to Windows and to Unix variants such as Linux, Solaris and MacOS. As to programming languages, many today are available on numerous platforms.
- **Reusability**: how much of the product can be applied to future developments. Many applications need some of the same functionality, either of a general nature (data structures and fundamental algorithms, GUI mechanisms) or targeted to a particular application domain. *Reusable software* is software that is sufficiently independent from the specifics of a particular project to be of use for subsequent ones. Helped by object technology, reusability in software has made great strides, leading to **software components** that serve the needs of many different developments. (Think of the Traffic library and all the libraries on which it itself relies.) Even if you are not producing software components, you can strive to make your software reusable to facilitate future projects.

In the literature you will see references to a quality factor called **maintainability**, having to do with the ease of continuing to work on a system after its initial release. This important concern is not an independent factor but a combination of the long-term product factors just reviewed, since maintenance may involve fixing errors, adding functionality and adapting to new platforms.

← *"Definition: Software maintenance", page 703*

All the properties reviewed so far are *external* quality factors: they are of direct interest to customers. Quality also involves **internal** factors, characterizing how the software is actually written, and directly meaningful to developers only. You informally know many of them because they correspond to the design and programming advice given throughout this book, telling you to ensure that your software is divided into classes reflecting relevant data abstractions, uses appropriate inter-class relations (client and inheritance), takes advantage of effective design patterns, includes meaningful contracts, applies information hiding, incorporates proper comments and documentation, and is written in a readable style facilitating future extensions. Another example of internal factors is the list of properties defined below for good *requirements* documents, some of which also apply to programs. Internal qualities are fundamental attributes of a software system; so fundamental indeed that external factors can only be achieved through them. Correctness and corrigibility, for example, both boil down to matters of systematic programming, sound software architecture and appropriate contracts.

→ "Fifteen properties of good requirements", page 724.

In the end, however, the **external** factors are the ones that matter, since they directly relate to customer needs.

Process quality

Process factors address the quality of the mechanisms used to produce the software. They include the following:

- **Production speed**: the ability to deliver a product in a short time. Every project has to worry about this; customers are waiting, competitors progressing, shareholders wondering.
- **Cost effectiveness**. This is also a concern for almost all projects. In software (unlike some other fields of engineering) the production cost is usually negligible. *Development cost* dominates everything else (except sometimes the cost of marketing, which can be significant especially for mass-market products); within it, *personnel costs* dominate other aspects such as equipment and office space. For that reason the standard measure of cost is the *person-month*: the average cost of employing one person — employee, contractor — for one month, all-inclusive.

Often also "man-month".

- **Collaboration effectiveness**: the effectiveness of procedures for combining the contributions of all project members and allowing them to communicate. Significant software projects may involve large numbers of people, requiring special attention to coordination mechanisms. Communication in particular is a delicate issue, which beyond a certain team size can overshadow all other aspects of the development. An extreme form of this phenomenon is known as "Brooks's Law" (from the name of the designer of the IBM OS/360 operating system), which states that adding people to a late project delays it further. This is only true of badly managed projects but highlights the need to devote proper attention to communication issues.

→ See book citation in "Further reading", 19.9, page 740.

- **Stakeholder involvement**: the degree to which the project takes into account all relevant needs and viewpoints.
- **Built-in assessment**: the inclusion of mechanisms and procedures in the process, to evaluate quality factors at well-defined steps. Quality is not just decreed and attempted: it must be checked and enforced. A good process integrates this task as one of its components.

- **Predictability**: the inclusion in the process of reliable methods to estimate other quality factors — in particular production speed and cost — ahead of time. Predictability is one of the most important characteristics of a good process; sometimes a guaranteed date is just as important as an early date. The software industry has not had a good record in this area, as many projects are late and over budget; the situation is improving, thanks to better application of software engineering principles and techniques.

- **Measurability**: the availability of sound quantitative criteria to determine achievement of other quality factors, both process and product; for example, techniques for measuring error rates. Effective management needs precise measures of progress. This criterion is closely related to the preceding two, since the ability to make predictions and to assess whether the predictions were met requires the ability to measure.

- **Reproducibility**: the independence of development, management and prediction techniques from unessential attributes of individual projects. In most industrial contexts, a software development does not happen in isolation but as one step in a succession of projects. It is important to carry over information and experience from one project to others, so that success in one particular project can be replicated on future ones. (*Failure* in a project also deserves careful analysis: not to reproduce it, but to learn the lessons.) This means in particular being able to abstract process and product attributes from the circumstantial properties of particular projects, such as the personalities of the developers and the specifics of the customer. Such reproducibility is one of the characteristics of an industrial production process. Because software is an intellectual activity, not assembly-line work, no process will ever achieve total reproducibility, nor would that necessarily be desirable; but a good software process reduces unnecessary sources of non-reproducibility — bad surprises.

- **Self-improvement**: the inclusion, in the process specification itself, of mechanisms to qualify and improve that process. Organizations, like people, can learn from experience. The self-improvement criterion assesses to what extent the process, as defined by the organization, encourages this phenomenon by including built-in evaluation mechanisms, which can be fed back into the process itself for adapting it as a result of the lessons learned.

Process models such as CMMI (studied later in this chapter) take these issues to heart, in particular the last five, to foster a software culture where assessment, predictability, measurability, reproducibility and self-improvement are built-in as core practices.

→ *"Capability maturity models", 19.8, page 735.*

Tradeoffs

While any software development should strive for the highest quality across all factors, the preceding review implies that tradeoffs are inevitable:

- Tradeoffs between process and product factors: a quest for perfection in the product might take too long to achieve, affecting the "production speed" process factor.

- Tradeoffs between product factors: ease of use does not always agree with security, since you will only want to make the product easy for *legitimate* users; passwords are bad for ease of use but good for security. Optimizing for efficiency can conflict with corrigibility (as it may lead to contorted code), and with factors such as extendibility, portability and reusability, all of which call for general solutions rather than techniques narrowly targeted to a particular platform and context.

One of the characteristics of a well-managed project is that it examines these tradeoffs explicitly, and resolves them on the basis of rational analysis. Otherwise they end up being resolved anyway, but not necessarily in the most desirable way; a common example is a misplaced concern for efficiency — extensive optimization where it is not essential — at the expense of other quality factors.

19.4 MAJOR SOFTWARE DEVELOPMENT ACTIVITIES

Software engineering involves a number of tasks. You have learned much about one of them, implementation, and gained a good first idea of others such as design, documentation and specification. We now go through the list of major tasks; the order is, roughly, from tasks closest to customers' concerns to those dealing with internal properties of the software.

Feasibility analysis is the task of studying a customer-related problem and deciding whether it is *possible* and *desirable* to build a software system (or a system involving software) to address it. The second aspect, although not immediately suggested by the name, is just as important as the first; not every system that can be built should.

Requirements analysis defines the functionality of the system. The elements making up a requirements document are of two kinds:

- *Functional* requirements, describing the results or actions of the system: "If the phone user leaves a coverage area to enter another, the connection shall automatically switch to an access point in the new area".

- *Non-functional* requirements, specifying constraints on the system's operation. They include *performance* requirements such as timing ("For an access point less than two kilometers away, switchover shall take no more than one second"), memory and bandwidth usage, security ("all communication with the access point shall be encrypted"). They also cover the impact on the system's environment, and consequences for stakeholders such as employees: the effect on work practices and training requirements).

Specification is the precise description of individual elements of the system. Requirements are customer-oriented; specification translates requirements into a form that is directly usable for the development of the software. The main difference is rigor and precision: the specification must give an unambiguous answer to every relevant question about the operation of the system.

Requirements and specification are sometimes treated as a single activity; the world **analysis** is then used to cover them both. In the lifecycle models that follow we will treat them as separate. Regardless of the exact division, the activities seen so far only address the *problem* to be solved; with the next tasks we enter the world of software *solutions*.

Design, also called **architecture**, builds the overall structure of a software system. It is responsible in particular for defining the principal units, or *modules*, of that system, and the relations between those units.

Implementation is the task of actually developing the program text to produce a usable system. This is also known as *coding*, with just a hint of a derogatory tone — as if writing the program were a menial chore to be performed once the great thinkers have done the analysis and design. (This book uses *programming* in the broad sense of program construction: not just implementation but also design and analysis.)

Documentation is the task of describing various aspects of the system to help its users and other stakeholders, in particular developers. Aside from documents for users, it may include project plans (for managers) and documents describing the results of some of the other tasks: requirements documents, specifications, design plans. The word "document" encompasses more than traditional reports designed for paper; today's documentation takes many other formats such as Web pages, online help files, or explanations included in program texts and processed by specialized tools (such as the header comments in Eiffel classes, or, in Java programs, special comments marked as "Javadoc").

Verification and Validation, or "V&V", is the task of assessing whether the system is satisfactory. The two aspects are complementary:

- Verification is *internal* assessment of the consistency of the product, considered just by itself. A typical example, at the implementation level, is type checking, preventing you for example from declaring a variable as *REAL* and using it as if it were an *INTEGER*.

- Validation is the *relative* assessment of a product vis-à-vis another that defines some of the properties that it should satisfy: code against design, design against specification, specification against requirements, documentation against standards, observed practices against company rules, delivery dates against project milestones, observed defect rates against defined goals, test suites against coverage metrics.

A popular version of this distinction is that verification is about ascertaining that the product is "doing things right" and validation that it is "doing the right thing". It only applies to code, since a specification, a project plan or a test plan do not "do" anything.

"Maintenance", as already noted, is not a separate activity but a combination of some of the tasks listed above; its only distinctive characteristic is *when* it happens: after the initial release.

19.5 LIFECYCLE MODELS AND AGILE DEVELOPMENT

A mainstay of the software engineering literature is the emphasis on lifecycle models: specifications of how to schedule the basic software engineering activities listed above into actual processes. The exercise has its limits, because the models describe idealized processes whereas software development is a human activity with its inevitable elements of unpredictability.

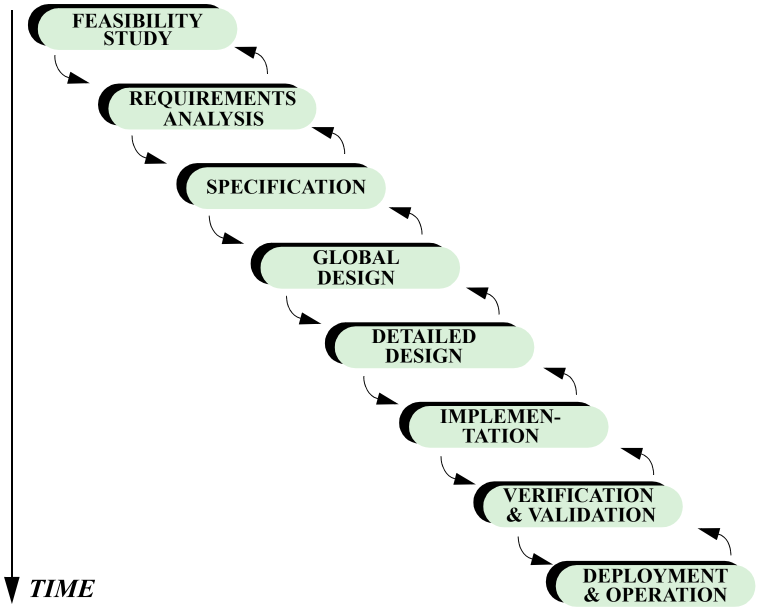
The waterfall

The starting point for all discussions of lifecycle models is the "**waterfall model**", dating back to a 1970 article (which was actually written to criticize the model, but ended up as its reference definition). The waterfall idea is simply to execute the above tasks (with possible variations) in the order given.

By Winston W. Royce, see tinyurl.com/r3jaj.

It has become a common practice — reflective perhaps of the lack of rigor of process model definitions — to represent them in graphical form. I will follow this practice here. The picture on the right is a conventional representation of the Waterfall model.

Like some other elements in this section, it is adapted from chapter 28, *The Software Construction Process*, of my "*Object-Oriented Software Construction*", 2nd edition, Prentice Hall, 1997, which contains a more detailed discussion of process models.



The waterfall model

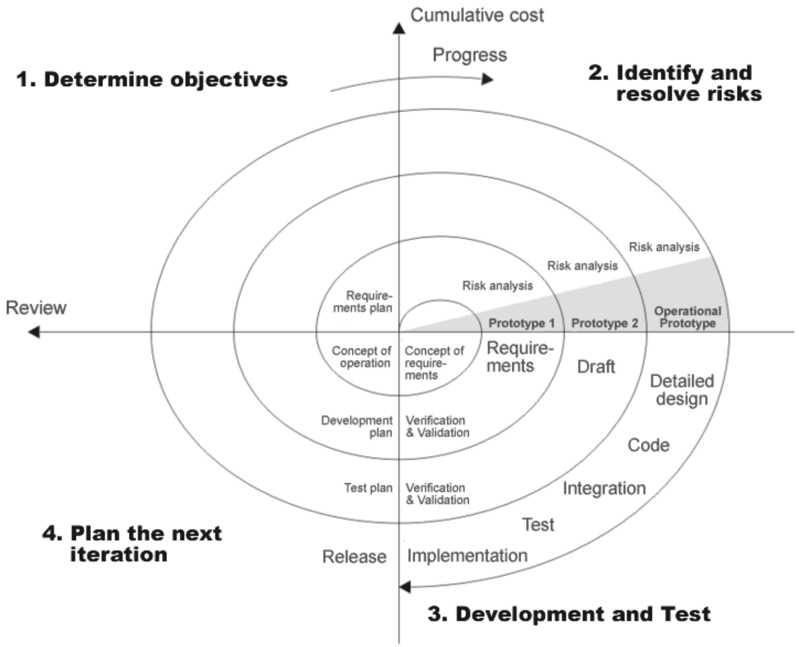
A disadvantage of this model is its rigidity, since it assumes that all activities will proceed synchronously on the entire system. Even more damaging is the late appearance of code (at the "implementation" stage); many problems can become clear only then, even if the previous stages, all devoted to plans and predictions and high-level views, seem to have proceeded smoothly. The inability to translate these hopes into code may be the reckoning.

The spiral model

In his book *Software Engineering Economics* (Prentice Hall, 1988) and other publications, Barry Boehm proposed a model that mitigates some of this risk by adopting an iterative approach, based on writing successive prototypes. This is known as the **spiral model** and is illustrated on the next page.

Each prototype in the spiral model follows a sequence of steps similar to the waterfall, but is intended to try out hypotheses and possible designs rather than producing a working system. Each iteration of the spiral benefits from the lessons of the previous iteration.

The spiral model is more flexible than the waterfall and avoids some of its principal deficiencies. What creates a risk, however, is that a prototype is not a system; often, to build a prototype, one relaxes some constraints (such as performance), which may turn out later to be the most critical and jeopardize the



The spiral model

Figure by Conrad Nutschan (Wikimedia Commons), adapted from Boehm, "Software Engineering Economics", Prentice Hall, 1981.

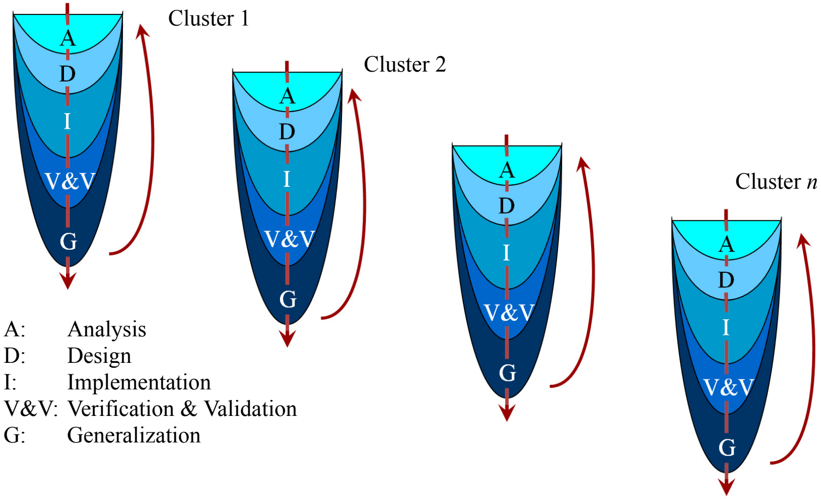
value of any lessons learned from the prototype. As another risk, if a budget is cut or pressure to release becomes untenable, the project will end up shipping a prototype — which was never intended for that purpose.

The cluster model

The model that fits best with object-oriented development as presented in this book is the *cluster model*, which modifies the basic waterfall in a different way: instead of a synchronous process it assumes that the system is divided into a number of subsystems or *clusters*, each with its own mini-process as pictured on the facing page. As appears immediately from the figure, this adds to the sequential dimension a concurrent aspect, since several clusters can proceed in parallel; a consequence for the project manager is the ability to react to surprises in project development: tasks that proceed faster or (more commonly) less fast than expected. It suffices in such cases to hasten or delay the start of another cluster, or of a task of that cluster.

To minimize risk, the development should start with the most fundamental clusters, providing critical functionality, and proceed towards the more user-oriented parts. It can be hard to convince customers of the merits of this order, since they naturally want to see visible results, but it is the soundest approach to ensure success.

The cluster model



The tasks appearing in each cluster's development are the expected ones from the waterfall, with one innovation: G for **Generalization**. The idea here is that once you have satisfactorily implemented a cluster you may not be through yet if you are interested in *reusability*. The goal of the Generalization phase is to remove from the cluster's classes any property that needlessly limits their applicability, such as dependencies on specific parts of the project, built-in size limits, insufficient contracts and imperfect inheritance structures. As a result the cluster's classes may be applicable to future developments in addition to providing an answer to the immediate needs of the project.

The development of each cluster proceeds continuously rather than through a sequence of separate steps (as in the waterfall). This is the idea of **seamless development**, particularly important in the Eiffel approach, where analysis, design and implementation all use the same notation and all build on the same basis, starting for example with high-level deferred classes that describe the problem, so that the design and implementation phases consist of refining and enriching these classes. This facilitates **reversibility**, the ability (symbolized by the reverse red arrows on the figure above) to go back in the process to improve or correct an imperfect first version.

Agile development

In reaction to the rigidity of some process models, *agile methods*, in particular the approach known as "extreme programming", de-emphasize plans and processes and focus instead on elements such as:

- Working code as the principal measure of progress.
- Collaboration between developers but also with customers, who are expected to have a representative in every development team.
- Frequent communication.
- Tests (rather than specifications) to guide the development.
- Small increments of development, to provide a constant feedback loop, and continuous integration (compile and test changes immediately, and integrate them promptly into the project baseline, rather than waiting for weeks or months and running the risk that two parts of the project diverged and have become hard to reconcile).
- Specific practices such as "pair programming" (developers systematically writing code in is groups of two people sharing their thinking aloud, to make programming choices explicit and catch more errors early).

The original introduction of agile techniques in the nineties caused considerable controversy and appeared at the time like a sociological phenomenon — the revolt of the cubicles (the programmers) against the corner office (the managers), or if you prefer the tussle between Dilbert and his pointy-haired boss. Things have cooled down considerably, and many agile practices, such as continuous integration, have been widely adopted. Others, such as the primacy of tests over specifications, remain questionable. But it is becoming clear that a software process can be both structured and agile.

19.6 REQUIREMENTS ANALYSIS

The rest of this book is almost entirely about technology. Without the proper programming techniques — algorithms, data structures, contracts, performance analysis, modular structures, compiler support, tool support and others that we have studied — projects will fail. But technology, however indispensable, is not sufficient. Successful systems are built to serve their stakeholders, in particular users, and must be adapted to their needs. Requirements analysis is the task of achieving a good match between what the users want and what the system does.

This is one of the core tasks of software development; it is hard, but can be quite enjoyable. There is perhaps no better way to dispel the popular view of software developers as introspective nerds than to note how much time, in the daily practice of their job, they spend interacting with users and other non-technical parties.

The following overview of the requirements task describes some of the challenges that it faces and a few principles that you should keep in mind to produce effective requirements.

Products of the requirements phase

A requirements process should produce two concrete results:

- A *requirements document* describing the characteristics of the software to be built.
- A V&V plan (often, just a *test plan*) describing how that future software, once built, will be assessed.

The second product is often neglected, but it is as important as the first: the time to come up with a good test plan, and more generally a good quality assurance plan, is *before* the software has been built. This ensures in particular that the tests assess whether the system satisfies its actual intent: the later the tests are defined, the more likely it is that they will be driven by the chosen design and implementation solutions and less by the original user needs. (In other words the risk is to get a V&V plan that tilts more towards verification, away from validation, and hence fails to assess fulfillment of the most important stakeholder objectives.)

The IEEE standard

A useful resource exists for preparing requirements: a standard of the IEEE Computer Society (together with the ACM, one of the most active professional associations in information technology — we already encountered one of its standards, for floating-point arithmetic). The "*Recommended Practice for Software Requirements Specifications*" standard defines some best practices for requirements, including a universal structure for requirements documents.

→ Precise reference in "Further reading", 19.9, page 740.

← The floating-point standard was cited in "Computing with numbers", page 279

This is a short and simple standard; it is a good idea to read it for its advice, and, if you have to write requirements, to follow the recommended structure, which the industry uses widely. This structure consists of three parts: introduction, overall description and specific requirements. Part 2, overall description, includes: product perspective; product functions; user characteristics; constraints; assumptions and dependencies; apportioning of requirements. Part 3 goes deeper into details of the system, including external interfaces, performance requirements and database requirements.

All this is no more than a checklist of the system properties that requirements may need to address. Because so many of these properties can affect the success of a software development, following the standard helps projects avoid costly upfront mistakes.

Scope of requirements

A software system is almost always part of a bigger system. “Embedded” software, say in a digital camera or a cell phone, is part of a system involving hardware. “Business” software is part of a system involving company processes. One of the first decisions to make when preparing the requirements is to define boundaries: do the requirements cover the software part only, or the entire system? The first answer does not mean you can ignore the rest of the system: you still have to specify the *interfaces* between the software and its operating environment.

Another important distinction, already noted, is between functional and non-functional aspects:

- Functional requirements specify the system’s responses. “*If the input in the social security number field is a valid social security number, the system shall display the first and last names of the corresponding person*” is a functional requirement.
- Non-functional requirements specify all other properties of a system, such as constraints on performance, availability and ease of use. “*Displaying the first name and last name shall take no more than 0.2 seconds in 99.5% of requests*” is an example of non-functional requirement.

Note the terminology: “a requirement” is a unit of specified behavior, functional or non-functional, as in each of these examples; “the requirements” means the collection of all such individual units.

Obtaining requirements

The process for obtaining (or “*eliciting*”) a system’s requirements varies widely. It can be very informal, with a few people laying down the essentials of the system and proceeding quickly to actual development. The “agile” approach mentioned above favors constant interaction with customers rather than a heavy upfront requirements process. Many large industry projects, however, devote considerable effort to getting the requirements right first, while sometimes leaving room for later revision as the construction of the software yields new insights.

This last comment highlights a general feature of requirements gathering. You might think that in an ideal process the system is entirely derived from “user needs”: the requirements team patiently goes around, asking customers what they want; they record all the answers, sort them out, organize them into a requirements document, and hand out the document to a development team which implements the customers’ desires. Things almost never happen this way. Nor should they:

- Various stakeholders often have conflicting views; someone must resolve the conflicts.
- User demands often include a mix of easy, feasible and hard (or impossible) features. Users often do not have a clear understanding of what is easy and

what is hard. Only the development team can assess the technical cost of each requested functionality, an essential criterion in deciding whether to include individual requirements.

- Users tend to think in terms of existing systems (or, at the other extreme, of systems that are impossible to build); often, developers are in a good position to propose functionalities that users would not have imagined. (This is a general feature of technology innovation: few breakthrough products — pick your favorite example, software or not — were designed by just gathering the wishes of a panel of potential users. The technologists listen to the users, but come back with "What if I gave you a device that looks like this?" proposals of their own.)
- Many external factors affect the final choice of functionalities, such as budget constraints, the existence of a previous software system, and the need to interface with other systems; think for example of the common case of a company being acquired by another, which has its own software.

These observations indicate that, beyond the simplistic view of a process that would just gather user needs then implement them, any realistic requirements process is a **negotiation**: users express the desirable, developers describe the feasible, and after a few iterations they agree on a middle ground. Such discussions can be quite creative, the two groups together devising solutions that neither would have initially imagined on its own.

The requirements process must support this model. While there is such a job description as "requirements engineer", the requirements effort should include the principal members of the software development team. They have a special role among stakeholders: if they cannot implement the required functionalities, the requirements document will be worthless. Even if the task is technically feasible, it has little chance of success if the development group is fundamentally hostile. The practical rule is that the development should not start until the requirements document has been endorsed — concretely, signed — by representatives of the two critical groups of stakeholders: the principal decision-maker on the customer side (the person who holds the purse strings) and the head of the development group. Many a software project failure would have been avoided if the project had applied this simple rule.

Sometimes a company will perform a requirements process without having selected a development team, external or internal (hence making this rule inapplicable); the idea is to define the needs first, and then to decide who is best equipped to answer them. This is a risky practice. It can lead to unrealistic requirements, in particular if the company hires external consultants for the requirements analysis. As I have seen too often in industry projects, such a process invites over-ambitious requirements: why strive for realistic demands if you know your responsibility stops there? The temptation is too great to please your customer by making promises that someone else will have to fulfill. In the best cases, requirements have to be redone — inevitably, in the direction of more limited functionality — when a development team takes over; in the worst cases, the development fails to implement the requirements, resulting in either delays or failure.

Techniques for gathering requirements include

- Interviews. You go around and ask representatives of each stakeholder category what they would expect from a new system or an extension to an existing system. The interviews must be carefully prepared, including both questions on predefined issues and open-ended parts allowing stakeholders to describe their thoughts freely. It is common practice to videotape the interviews for later perusal.

- Workshops. Gathering a number of stakeholders in a room for a discussion of desirable features may be a better use of time than conducting many individual interviews. Because the setting encourages discussions, you may avoid the painful process of discovering contradictions between the requests of different stakeholders and resolving them after the fact; different views may come out early and be reconciled through direct interaction.

- Previous systems. Few developments start in a virgin environment. Usually, the company has in the past built or acquired software addressing some of the same needs. Studying these existing systems to understand their benefits and limitations is an important part of requirements gathering. There may even be a technical requirement that the new system perform at least as well as an existing one, or deliver the same results in comparable cases.

- Competing systems. If you are in the business of selling software products, you will need to know what the competition is offering. Even if the development is internal to your company, it can be useful to study how its competitors, who often have similar needs, are addressing them.

The glossary

Every requirements effort should develop, as one of its products, a *glossary*. (In the IEEE standard's recommended structure it is section 1.3, "Definitions, acronyms, and abbreviations".)

Every technical area has its jargon; stakeholders from the area, often called *domain experts*, will use it in requirements interviews and workshops; they might assume that you understand it — and that all other domain experts understand it in the same way they do. Neither assumption necessarily holds. Your first task in a requirements process is to list all the terms and define their precise technical meaning. Collect all such definitions into a glossary, and show it to the domain experts to make sure that they agree — with you, and with each other. This will be one of your principal resources for the requirements process, and also beyond it, since many of the concepts listed in the glossary will need direct counterparts (classes, features and such) in the programs.

Machine properties and domain engineering

An important distinction to keep in mind when writing requirements (emphasized in particular in an important book by Michael Jackson) is between *domain* and *machine* properties. Any system will function in some domain, natural or human, which has its own laws: an electronic system is subject to physical limitations on signal speed; a banking system is subject to banking regulations. The software's development will yield a system — a "machine" — that adds its own rules. Jackson emphasizes the need to distinguish between the resulting two categories of requirements elements:

→ *Reference in "Further reading", 19.9, page 740.*

Jackson (2004)

- "*No transfer shall be accepted if it would cause the account balance to become less than the approved overdraft limit*" is a domain property: it is imposed by the environment, here business or legal rules.
- "*A transfer attempt that would bring the balance below the approved overdraft limit shall result in the sending of an email to the account manager*" is a machine property, describing a particular decision being made for the system. (In an actual requirements document, it would have to be stated more precisely.) This particular machine requirement follows from the preceding domain requirement, but not all machine requirements will have this direct relationship to the domain; some are purely system-related decisions.

In the short text describing the Paris metro, the statement that "*It is a property of the metro network that such a route always exists between any two stations (in mathematical terms, the graph is connected)*" was a domain property: any software system dealing with the metro must take it for granted. The south-to-north station numbering rule (explicitly introduced as a way "*to make our life easier*"), is a machine property, describing a particular convention we chose to model the metro.

← *"Touch of Paris: Welcome to the Metro", page 52* and *"Conventions: Line numbering", page 58.*

Out of this need to understand the domain emerges a new software engineering task, distinct from requirements engineering: *domain engineering*, devoted to modeling the general properties of an application domain. Domain engineering is not tied to a particular project, but helps the requirements process of all projects in the chosen domain. For example a company that regularly develops train control software may invest in a non-project-specific effort to model the general properties of railway systems.

Requirements are the combination of domain constraints and machine properties. Too often, requirements documents fail to distinguish between the two kinds; then readers, in particular developers, do not immediately see what is a consequence of external circumstances (the speed of light, for example, will not change) and what might be reconsidered in the future evolution of a system. For this reason, it is important for the requirements document to specify the nature, domain or machine, of every individual requirement.

Fifteen properties of good requirements

Let us now complete this overview of the requirements phase by taking a look at the properties — fifteen of them! — that good requirements should satisfy: *requirements on requirements*. I should note that I have never seen a requirements document that satisfied all of them; but they provide a clear set of objectives for any requirements writer. Some, but not all, appear in the IEEE standard; they are marked with an asterisk * below.

Requirements should be **justified**. Every individual requirement should have its source in some identified and explicitly stated stakeholder need.

They should be **correct***: any system that satisfies the requirements should meet stakeholder needs. This is very hard to guarantee formally; informally, you should make sure that all stakeholders know and approve the requirements affecting them, which brings up the next point.

Requirements should be **complete**: they should cover *all* approved stakeholder needs. In principle this is impossible to ascertain, since the immediate question arises: complete with respect to what? Any answer would have to refer to some higher-level description of the intent; but that would just be another requirements document, meaning that we would only have pushed the completeness problem one level further. In practice, a useful heuristics exists, based on concepts introduced early in this book:

← *"Features, commands and queries", page 26.*

Touch of Methodology: Sufficient completeness

A requirements document should define the effect of every command of the system on every query of the system.

Like a class, any system provides some *commands* and some *queries*: you can ask the system to perform some actions, and you can ask it for information. A requirements document will describe both the commands and the queries; the information should be sufficient to enable the reader to determine how executing any one of the listed commands will affect any one of the available queries.

"Sufficient completeness" is a technical term, introduced in a 1978 article by Guttag and Horning to characterize properties of *abstract data types*, the theoretical basis for object-oriented programming.

→ *Precise reference in "Further reading", 19.9, page 740.*

Requirements should be **consistent**: they should not include contradictions. This is surprisingly difficult to achieve. The difficulty comes in part from the size of many industrial requirements documents, which can run into hundreds or thousands of pages for complex systems. Inconsistencies will slip in, page 325 stating that the system must close the door *before* sounding the beep that signals the train will be moving and page 1232 implying the reverse. But of course the programmers will have to implement one or the other, so the requirements phase is the time to detect and correct such inconsistencies.

> Note the difference between consistency and correctness: consistency is internal to the requirements document; correctness addresses its satisfaction of some external constraints. This is the same distinction as between verification and validation.

← "Major software development activities", 19.4, page 712.

Requirements should be **unambiguous**. What makes this goal challenging is that most requirements documents are written in a natural language, with its risk of imprecision and misinterpretation. Consider this example:

> *The Background Task Manager shall provide status messages at regular intervals not less than 60 seconds.*

It is easy to think of many ways the system's developers could understand this, some leading to results that users will find highly unsatisfactory. The requirements expert who cites this extract proposes as a replacement (making some guesses about the intention, to be confirmed with customers):

Example from Wiegers's "Software Requirements", see reference in "Further reading", 19.9, page 740.

> *1. The Background Task Manager (BTM) shall display status messages in a designated area of the user interface.*
>
> *2. The messages shall be updated every 60 (plus or minus 10) seconds after background task processing begins and shall remain visible continuously.*
>
> *3. If background task processing is progressing normally, the BTM shall display the percentage of the background task processing that has been completed.*
>
> *4. The BTM shall display a Done message when the background task is completed.*
>
> *5. The BTM shall display an error message if the background task has stalled.*

This is far more precise and in a typical style for industrial requirements document (of which the word "shall", already present in the first variant, is a fixture, encouraging requirements writers to be clear about what the system must do). But the example, in its obsession to leave no stone unturned, also illustrates the difficulties and limits of requirements specification, and helps understand why carefully written requirements documents can run, as noted above, into the thousands of pages.

To achieve precision and remove ambiguity, natural language is often inadequate. This is the reason why considerable work has attempted to use mathematics for requirements; this is known as *formal specification*. An article in the bibliography discusses some of the benefits and challenges.

→ "On Formalism in Specification", see "Further reading", 19.9, page 740.

Requirements should be **feasible**. It is all too possible, especially (as noted) if those who define the goals are not those who will implement them, to produce pie-in-the-sky requirements that will never be met. The task of a serious requirements process includes limiting ambition and emphasizing the possible.

Requirements should be **abstract**. A common pitfall in requirements preparation is to start defining design and implementation choices. Such *overspecification* prematurely narrows the realm of possibilities and betrays the mission of requirements, which should limit themselves to defining the *what* and not encroach on the *how*.

←See "Order overspecification", page 151.

Requirements should be **traceable***. In other words, it should be possible to keep track of the consequences of every individual requirement in the code and all other software products. This makes it possible not only to check that a proposed implementation takes all requirements into account but also, for any requirements change, to track down all the software elements that may be affected.

Reminder: the asterisk identifies requirements listed in the IEEE standard.

As an example of a traceability mechanism, EiffelStudio includes a facility known as EIS (Eiffel Information System) supporting the definition of links, both ways, between individual elements from a requirements document and individual classes or features of the Eiffel software. In principle, every software element should follow directly or indirectly from a requirement, and every requirement should have some counterpart in the software. EIS enables you to add links to (for example) a PDF or Microsoft Word document, so that clicking the link will open EiffelStudio on the designated class or feature; and to add links to the Eiffel code that, in the same way, lead to the appropriate parts of the document. EIS is a direct implementation of the traceability principle, intended in particular to facilitate requirements change.

See docs.eiffel.com/book/eiffelstudio/eiffel-information-system.

Requirements should be **verifiable***. It is useless to state a requirement unless a clear criterion exists to decide whether a proposed system meets it. An extreme — but unfortunately common — example of non-verifiability is a requirement of the form "the system shall respond in real time" (to certain commands or queries). What is real time to me may be an eternity to you; real-time response for a banking system may be 2 seconds, for a network device it may be 100 microseconds. The document should specify the expected response time precisely and distinguish between average and maximum, normal and degraded operation etc.

Requirements should be **delimited**. It is important to state not only what the system must do but also what lies beyond its purview.

Requirements should be **interfaced**: they should precisely specify the system's connections to other systems — software, hardware or human.

Requirements should be **prioritized***. Sometimes circumstances prevent a project from implementing all that was hoped; typical causes include budget

The IEEE standard says "ranked for importance and/or stability".

cuts, unexpected difficulties (which delay the project and lead to trimming some functionality to release the product in a reasonable time) and the appearance of a competing product (forcing an early release). The choice of what to remove should not be left to the time of such project hiccups; instead, the requirements should specify the importance of each functionality and constraint relative to other elements of the requirements. This enables project management to make choices on the basis of pre-agreed priorities.

Requirements should be **understandable**. The drive for precision and detail can result in formidable documents. Unless the requirements are easy to consult and understand, they will not play their due role.

Requirements should be **modifiable***. Circumstances evolve, companies get merged, customers change their mind. Like any other software product, the requirements should be designed for change.

Finally, requirements should be **endorsed**. There is so much room for misunderstanding and conflicts in project development that one should not start without a clear, formal understanding involving at least (as already noted) the signatures of the main representatives on the customers' and developers' sides.

I hope I did not scare you with this long list of requirements criteria. It is in fact possible to write good (although perhaps not perfect) requirements documents, reflecting stakeholders' needs and providing a sound basis for development and V&V. This is an important part of software development, and a great opportunity to combine technology with business and human aspects.

19.7 VERIFICATION AND VALIDATION

The first rule of V&V is that it would be nice not to have to do it. The purpose of all the rules of design and programming methodology in this book (and I trust you will apply every one of them, on every single occasion) is to ensure that you produce software that works the first time and every time. But you still have to convince the rest of the world that it does; besides, you might still make mistakes; or, if this sounds insulting, just consider that your assignment might be to enhance or modify software written by another, less enlightened programmer. In practice, Verification and Validation are a major part of the software development effort, often consuming more time than software construction proper.

We will limit ourselves here to a quick overview of some of the basic ideas. The discussion mostly considers programs — even though, as noted, V&V also applies to non-program artifacts such as documentation. The term "software quality assurance" will be used as a synonym for V&V.

This is a slight abuse of language since quality assurance includes *a priori* techniques for building quality software, in addition to techniques for assessing the quality of software once built.

Varieties of quality assurance

To many people, V&V only evokes testing and debugging. The range of techniques is in fact broader.

Testing is the main kind of *dynamic* technique. It consists of executing the system (hence "dynamic") on selected inputs, to try to uncover deficiencies.

Static techniques analyze the program text without executing it. They include code reviews, static analysis, program proving and model-checking.

We will examine testing first, then static techniques. The following terms are useful for this discussion; it comes from yet another IEEE standard, on software engineering terminology:

IEEE Std 610.12-1990, tinyurl.com/3w57pk (1990 text, but much of it still useful).

- A program execution that does not function as expected (it crashes, or produces a wrong result) causes a **failure**.
- The failure is (except in the rare case of a hardware malfunction) due to a **fault**: a characteristic of the software that is not what it should be. Note that the fault is not necessarily in the implementation (the code) but might be at any other level, such as specification or design.
- The fault is due to a **mistake** made by a software developer. (The term "*error*" is also common, but the standard recommends "mistake" because "error" also has the meaning of a discrepancy between the actual result and the expectation, as in, for example, "numerical error".)

The term *bug* is not part of this official terminology, although it is commonly used to denote either faults or mistakes, and figures in *debugging*, the task of correcting the mistakes to remove the faults and stop the failures. (Many people have pointed out that "bug" evokes some creeping creature insinuating itself into the program, and may be an attempt to shirk responsibility by pretending that it wasn't the programmer who inserted the fault in the first place.)

Testing

We start with the most commonly applied technique, testing. The first observation is one of modesty: while it is tempting to think of testing as a way to assess quality, it is not very useful in this role. The reason was expressed by Edsger Dijkstra in one of the most quoted sentences in the history of computing science: *testing can show the presence of errors, never their absence*. A failed test reveals a fault; a successful test says little, since any realistic program has a cosmic number of cases to be tested. Even a program to multiply two 64-bit integers yields 2^{128} cases.

"Errors" is used here for faults.

Dijkstra's comment is accurate, but should not be taken as an indictment of testing. "Showing the presence of errors" is extremely useful in practice, enabling us to find faults before our users do. This is what testing is: a technique to make programs fail.

Testing technology has considerably progressed in recent years. The evolution has been towards more automation. Frameworks now exist, for all important programming languages, enabling developers to record tests and run test campaigns automatically; they are often known generically as "XUnit", following the original JUnit framework for Java. They have enjoyed a wide success since the alternative — manually managing and running the tests — is increasingly unrealistic given the ambition of today's programs and the resulting high numbers of tests to run. Computer power indeed makes it possible to carry out many tests, but the process requires automated support.

Automation is particularly necessary for the task known as *regression testing*. It is a fact of software development, often surprising to newcomers, that corrected faults may resurface in later releases (indicating that the software has partly regressed to an earlier state, hence the name). Causes of regression include:

- Insufficiently thorough corrections, which remove the symptom but not the cause (the original mistake).
- A pattern of mistaken reasoning that has caused several faults and may come back even after some of them have been corrected. The debugging advice of Tom Van Vleck in his delightful cartoon (see overleaf) is, unfortunately, not applied widely enough:

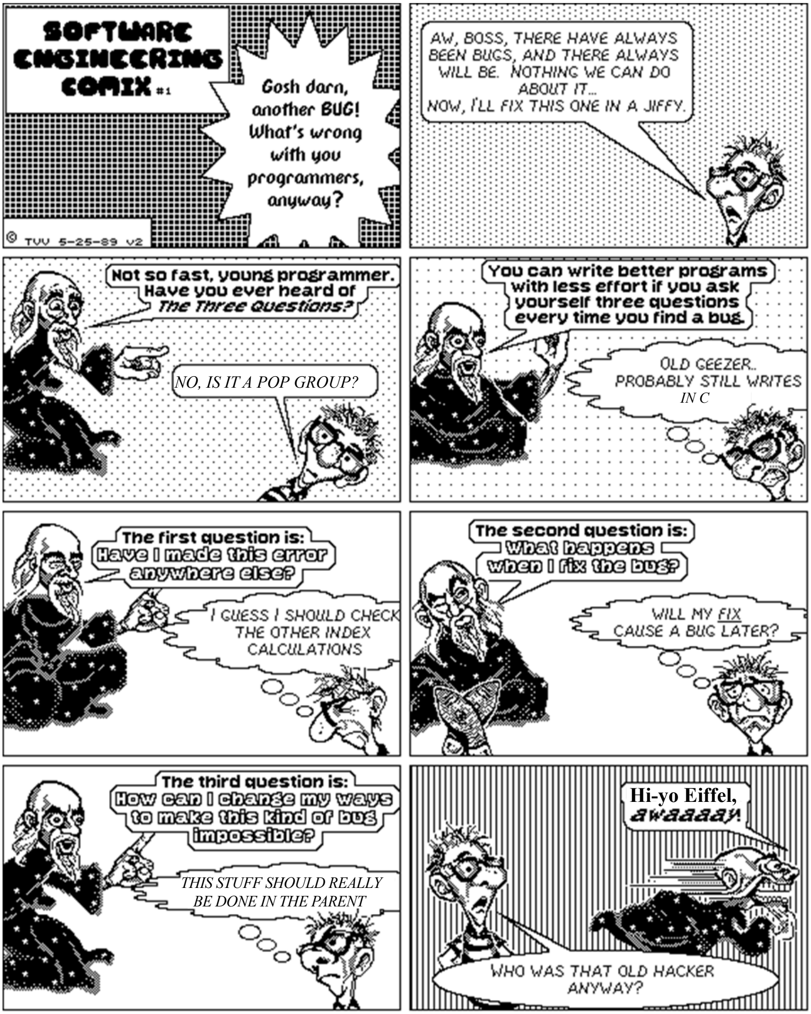
Regression testing tries to catch such cases by running all the tests that previously failed. Every serious project runs a regression test prior to releasing any new version. This can be expressed as a principle:

***Touch of Methodology*:**
The Failed Test Principle

Every failed test must become part of the regression test suite, and remain in it for the entire life of the project.

Recent research is taking test automation even further. As an example of what is now becoming possible, take a look at the Eiffel Test Framework which (since version 6.3) has been an integral part of EiffelStudio. You will notice, in addition to standard "XUnit" mechanisms, two advanced facilities:

- *Test synthesis from failure*: every failed execution, in accordance with the Failed Test principle, automatically produces a test. The novelty here is the automation. Many of the most important potential tests come from interactive executions that failed during development, but in usual approaches they are lost after correction and do not contribute to the regression test suite. Here the process of turning a failure into a reproducible test is automatic.


SOFTWARE ENGINEERING COMIX #1
Gosh darn, another BUG! What's wrong with you programmers, anyway?
© TVV 5-25-89 v2
AW, BOSS, THERE HAVE ALWAYS BEEN BUGS, AND THERE ALWAYS WILL BE. NOTHING WE CAN DO ABOUT IT... NOW, I'LL FIX THIS ONE IN A JIFFY.
Not so fast, young programmer. Have you ever heard of *The Three Questions?*
NO, IS IT A POP GROUP?
You can write better programs with less effort if you ask yourself three questions every time you find a bug.
OLD GEEZER.. PROBABLY STILL WRITES IN C
The first question is: Have I made this error anywhere else?
I GUESS I SHOULD CHECK THE OTHER INDEX CALCULATIONS
The second question is: What happens when I fix the bug?
WILL MY FIX CAUSE A BUG LATER?
The third question is: How can I change my ways to make this kind of bug impossible?
THIS STUFF SHOULD REALLY BE DONE IN THE PARENT
Hi-yo Eiffel, awaaaay!
WHO WAS THAT OLD HACKER ANYWAY?

Three questions you should ask about each bug you find

(Tom Van Vleck, Software Engineering Notes, vol. 14, no. 5, July 1989, slightly adapted)

- *Test generation from specifications*: you can ask the Testing Framework to test a class for you without having to provide input values. The tool will exercise all routines of the class, using values and objects that it creates automatically. The process can happen in the background while you are developing your software, making itself heard only if it causes a failure (remember, the purpose of testing is not to ascertain quality but to uncover faults). Failures in this case are postcondition or invariant violations.

Testing is a vibrant research area, and you can expect to see many more tools and facilities in future development environments.

Coming back to today's testing technology, a few more notions are worth noting (and looking up, for details, in software engineering textbooks and the testing literature).

Testing occurs at several levels of granularity. *Unit testing* covers individual modules — typically classes or clusters in object-oriented development — and is usually carried out by individual developers. *Integration testing* assesses how a group of modules or subsystems perform when combined; it is generally the task of the development group — possibly handled by a specialized subset of that group, the "test team" or "quality assurance team". *System testing* tests the system as a whole; often the term denotes a step that is still performed by the development group or its test team, unlike *acceptance test*, which determines the acceptability of the system from the customer's viewpoint, and is the responsibility of the customer organization, or of a joint customer-developer group.

For unit testing, it is common to distinguish between the *white-box* and *black-box* approaches. In white-box testing, the program text is available to guide the testing process, whereas black-box testing relies on the program's specification only. Black-box testing is the only solution if you acquire components from an external provider, do not have their source text, and want to assess their applicability to your development; but it may be interesting even for software of which you could consult the source if you wanted to. As an example of this last case, the just-noted mechanism for automatic test generation from specifications, in the Eiffel Test Framework, does not use the implementation but works on the sole basis of the class API including contracts.

Finally, we note the concept of *test coverage*, mostly applied to white-box testing in the current state of the art. Coverage is a measure of the quality of a *test suite* (a collection of test cases), attempting to estimate how much of the functionality has been tested. Coverage measures include:

- *Instruction coverage*: what percentage of the program's instructions does the test suite exercise?

Also called "statement coverage".

- *Branch coverage*: what percentage of branches (elementary paths of the program, for example the two branches of a Conditional) are exercised?

Many other coverage criteria exist, although in the end the only one that really counts is how many faults a test suite uncovers (which may or may not correlate with elementary coverage measures). A black-box generalization of the notion of coverage would be to define *specification coverage*, estimating how many of the cases permitted by the specification have been tried.

Static techniques

We conclude this review of V&V by taking a look at static techniques.

Design and code reviews, also known as inspections, are a manual process designed to uncover faults and other deficiencies. The target is some software element, typically under the responsibility of one developer: a program unit such as a class, but also possibly (since reviews can also apply to products other than programs) some part of a design document or even a chapter from a user's manual. The text is circulated in advance, and discussed in a meeting whose purpose is to discover possible problems. The meeting has no other goal: it is not intended to *correct* these problems (that will be the developer's responsibility, after the meeting), or to evaluate the developer.

This is the description of the classical idea of code reviews; with the advent of the Internet and the increasing practice of using geographically distributed development teams, the process can take advantage of remote interaction techniques. One of the lessons of such experiences is that code reviews are more effective if conducted partly in writing; the process starts ahead of the meeting, with participants annotating a common document (using Web document-sharing technology that only became widely available in recent years). It turns out that in most cases the original developer and the critics agree. The meeting (in practice, a conference call) can then be devoted to the most interesting issues, those on which disagreement remains.

My article "*Design and Code Reviews in the Age of the Internet*" (*Communications of the ACM*, vol. 51, no. 9, September 2008), describes the process in more detail.

Available on the ACM site and at se.ethz.ch/~meyer/publications/acm/reviews.pdf.

We cannot expect to use reviews as an effective tool for systematic detection of faults: reviews are a time-consuming human process, which does not scale up. Rather, they are *spot checks*; they will discover some faults (it is indeed a good idea to apply reviews to critical modules), but the main benefit of performing reviews is to assess an organization's or team's overall design and code practices, especially practices that can damage quality. It is therefore important, whenever a review has identified a deficiency, to probe further into its causes and ponder what techniques can be used to avoid similar mistakes in the future.

A more effective static analysis process requires automated tools. If you have used a compiler for any statically typed language you have used a static analyzer, since part of the compiler's role, as we have seen, is to enforce the type system. Beyond such direct implementation of programming language rules, **static analyzers** look for code patterns that might be faulty even if they do not explicitly violate the language definition. Examples include:

← "The compiler as verification tool", page 338.

- Variables that can, on some program paths, be accessed before they have been set (in a language that does not include automatic initialization rules).
- Variables that are not used (not necessarily a fault, but an anomaly).
- Void calls (if the language does not enforce void safety).

The ultimate form of static analysis is **program proving**, the most ambitious but also the hardest approach. It uses the term "proving" in the mathematical sense and hence assumes that the properties of the software have been mathematically, or "*formally*", specified. Eiffel's contracts give an idea of how such specifications may look like: every software element is characterized by a precondition and a postcondition (for routines) or an invariant (for a class). These are abstract specifications of functionality. For full program proofs, the specifications must be more detailed, but the general idea remains applicable. "Proving" a class then means establishing through mathematical techniques that every implementation satisfies the relevant specification: every routine, started in any state satisfying its precondition and the class invariant, will terminate its execution in a state satisfying its postcondition and again the invariant.

> This form of specification is in line with the observation, earlier in this chapter, that the correctness of a program can only be defined with respect to a stated specification. Here the specification takes the form of contracts, and correctness means that the implementation is consistent with the contract.

← *"Immediate product quality", page 707.*

Because of the many details involved in such proofs, and also because human-written proofs are subject to error and would not necessarily be trusted, the process must (beyond academic examples) rely on automatic tools, known as *program provers*. Many program provers run on top of *theorem provers*, which are able to perform general mathematical reasoning. Work on theorem and program provers has proceeded for decades, and has received new impetus in recent years thanks to advances in proof technology and a better understanding of the issues. This is a very active research area.

Some of the most impressive progress has been brought by techniques that generally do not attempt full proofs of functional correctness, but focus instead on identifying specific faults, the way testing does. *Model checking* takes advantage of computing power to explore the state space of the program, or more realistically of a simplified version of the program; if it succeeds in reducing the state space to a tractable size, it can determine whether any of the states violates the property of interest, often a correctness or security property. This approach integrates some ideas from testing (exploration of many cases, focus on uncovering faults rather than establishing full correctness) but is a static technique and in fact a form of proof. *Abstract interpretation* defines an abstract version of a program to which it applies advanced static analysis techniques; one of its success stories is the proof that large safety-critical programs embedded in the Airbus A330/340 and A380 planes will not produce any run-time failures.

What does this practically mean, you may ask, for the daily practice of programming? It really depends where you work. For a long time, "formal methods" — as proofs and related techniques are known — were considered an intellectually attractive idea not applicable to industry developments. (Calling an approach "academic" is often the kiss of death.) This view is simply no longer tenable today. With the steady improvement in both theory and tools, and the increased awareness of the risks of malfunctioning software systems, a number of industry developments have used formal methods and tools. Some of the lessons are encouraging and some sobering:

- On the positive side, formal tools work. It is possible to develop realistic systems equipped with full guarantee of correctness. Note, by the way, that such proofs do not mean that the software is perfect, only that it meets the *specified properties* under *specified assumptions* (for example, that the hardware works right); they make no other claim. Still, they are solid enough to remove the need for certain kinds of test; there is generally no point in testing correctness properties that have been mathematically proved.
- The limitation is that such impressive results can only be obtained, in the current state of the art, through a special development process and by specially trained development teams. In addition, they generally assume a drastically reduced programming language; often you have to renounce most of what makes life worth living: classes, inheritance and its consequences (polymorphism, dynamic binding), genericity, dynamic object creation, recursion…

All these features of modern programming language technology are there for a reason: facilitating the construction of large programs with elegant architectures open for extension and reuse, and enhancing the programmers's power of expression. As a result, the use of formal methods has largely been confined so far to areas where such criteria have to yield to one crucial goal: correctness. This is the case with life-critical systems, such as train or plane control systems, where everything must be done to avoid malfunctions. The Airbus software is a representative example.

The rest of the industry is generally not willing to adopt the kind of asceticism that such techniques impose on their followers. Considerable research is in progress to make them more applicable to more mainstream developments; Tony Hoare has initiated a "*Grand Challenge*" to encourage a concerted international attack on the problem of producing verified software. We can indeed hope that, within a few years, fully formal tools will benefit even those of us who do not have the privilege that their programs, if they malfunction, will kill someone.

19.8 CAPABILITY MATURITY MODELS

Our last topic for this chapter covers a general organizational approach that companies have increasingly applied in recent years. It is in line with the ideas behind the lifecycle models discussed earlier in this chapter, but extends them to a more general framework.

Assume you are in an organization that needs to contract out some development to a software company. There is no product yet to assess, so all you can evaluate is the process. The company tells you they have everything under control, but how do you know?

In the early nineties this need for objective evaluation of companies' software processes led the US Department of Defense (DOD), the world's largest consumer of software services, to ask the Software Engineering Institute, a DOD-funded center at Carnegie-Mellon University in Pittsburgh, to develop a model for the level of industrial "maturity" of software organizations. The resulting "Capability Maturity Model", further developed into a more comprehensive set of models known as CMMI ("I" for "Integration"), has exerted a profound influence on several segments of the software industry, in particular:

- US defense contractors, its initial target.
- Indian software companies, probably not part of the initial plan; India's nascent outsourcing industry saw in the CMM, as it was then called, a critical tool for obtaining outside certification that would reassure the Western customers they were trying to attract. Soon after the model was released, Indian companies started to account for a significant share of CMM certifications.

CMMI is also used outside of these communities. As a sign that it has extended its reach beyond its initial target group, the proportion of defense contractors and military organizations in CMMI assessments went down to 40% in 2004 and continues to decrease.

> Some companies seeking process improvement and qualification prefer other models. The 9000 series of standards from the International Standards Organization (ISO) is the software-oriented branch of a set of international standards for industrial quality in general. SPICE (Software Process Improvement and Quality dEtermination) combines some elements of the other two. This overview only considers CMMI.

CMMI scope

CMMI and friends examine only the process. They are technology-neutral, language-neutral and tool-neutral. All they assess is whether the organization has a set of clear procedures in place, applies them, controls that it applies them, measures their effect, and strives to improve them. In terms of the earlier discussion of software quality, the emphasis is on *process* factors, especially the

← *"Components of quality", 19.3, page 705.*

last five on our list. Think of the pilot and copilot going through their check-list prior to a flight: what matters is that they consider every single item on the list, tick it off if it's OK, and follow the predetermined action (such as calling aircraft maintenance) if not. Because of this emphasis on formal procedures at the expense of technology, some people dismiss process models as merely a way for managers to "cover their bottoms" in case the project fails, by showing that they did everything by the book. Indeed there have been cases of project failures in organizations with high CMMI or ISO qualifications. But such dismissal is a classical case of confusing necessary with sufficient: software projects, especially large ones, need both high process quality and excellent technology. While you can still mess up if you have a perfect process, process qualification helps companies *not* mess up.

← *"Process quality", page 710.*

Key to CMMI is the notion of assessment. Organizations wishing to establish their "maturity level" as discussed next may get themselves evaluated — in the military's passionate acronym culture this yields an example of acronym nesting, SCAMPI for "*Standard CMMI Appraisal Method for Process Improvement*" — by assessors officially accredited by the Software Engineering Institute: 179 "SEI partners", organizations rather than individuals, as of 2005. Assessed organizations may publish the results of the assessment — typically, to boost their attractiveness if they are software companies — or keep them for themselves. Between April 2002 and September 2004, the SEI was notified of 424 appraisals affecting 206 companies, half of them outside the US.

CMMI disciplines

As the I in the acronym attests ("Integration"), CMMI outgrew the original CMM to cover a range of models that extend beyond software; their four "disciplines" include software engineering but also:

- "Systems engineering". This concept covers non-software aspects of a system; indeed, software is often part of a bigger system — think of the software in your car, music player or refrigerator — which has its own process involving hardware, software and other aspects.
- "Integrated product and process development".
- " Supplier sourcing": selecting, controlling and coordinating all the suppliers that contribute to a project. Large projects often involve the participation of many suppliers; in some cases, for example a government customer with no software development department of its own, a project is entirely outsourced. Supplier sourcing is the process of overseeing outsourced work.

An organization interested in implementing CMMI and being assessed may select from these disciplines, depending on its activity and needs.

Goals, practices and process areas

The essence of CMMI is to define **goals** and recommend **practices**:

- A goal is a desirable property of a process. For example, every project should have good requirements, describing user needs; this observation yields goals such as "*Develop customer requirements*" and "*Analyze and validate requirements*" (that is to say, it is not enough just to produce requirements for a project, but one should also have formal procedures to check that they are feasible and satisfy the stakeholders).
- A practice is a technique that has been shown to help achieve a goal. Examples are "*Establish a definition of required functionality*" and "*Analyze requirements to establish balance between stakeholder needs and* [project] *constraints*".

As the examples indicate, every practice must be related to a certain goal; using software terminology, the goal is a specification and the practice an implementation (carried out by humans) of that specification.

Such goals and the corresponding practices are grouped into collections called **process areas**. The preceding examples are part of the process area "*Requirements development*".

> The term "area" is not intuitive, so to understand the rest of the discussion you must remember that a "process area" is exactly what this definition says: a collection of goals and of practices supporting those goals.

Two models

CMMI exists in two variants: *staged* and *continuous*. The difference is scope:

- The staged variant addresses the maturity level of an organization as a whole. This has the merit of yielding a single, global figure ("Our division just achieved CMMI qualification at level 4!") but ignores the differences between various activities and specialties; for example an organization might be very good at software construction but not have mastered requirements yet. Staged description is in the tradition of the original CMM, and is still the dominant practice.
- Continuous description allows assessment of individual process areas and hence provides more flexibility.

Common to both variants is the notion of assessment level. CMMI enables you to qualify your organization — all of it if staged, some of its process areas if continuous — at one of **five levels**, labeled 1 to 5 in order of increasing closeness to the Nirvãna of total control. (The continuous representation adds a level 0, "incomplete", for process areas not applied.)

In the staged variant, each level is characterized by a set of process areas: you reach that level if you apply the applicable practices and meet the corresponding goals. For example, reaching level 2 assumes that you satisfy *Requirements management* and other process areas listed below. In addition, each level has one **generic goal** and an associated set of **generic practices** not belonging to any process area; for example level 2 has the generic goal "*Institutionalize a managed process*", meaning a company-wide definition and enforcement of a development process, and associated generic practices such as "*Plan the process*" and "*Provide resources*".

As a consequence of these concepts, the goals and practices are divided into two categories:

- **Generic**: characterizing a CMMI level, but not belonging to a particular process area.
- Those belonging to a process area, called **specific**.

Assessment levels

Here is the general characterization of the levels, in the staged variant. The more precise definition comes from the table on the facing page, which identifies the generic and specific goals of each. There are, as noted, five levels:

1 **Initial**: this characterizes an organization with little process definition or enforcement. Some projects succeed, others not, but no one quite knows the reason. It is like going for mushrooms in the woods on a rainy day in October: this oak has lots, that one has none, but why? To me they look just the same. In software development this is sometimes known as the "heroic" stage: success depends too much on the people involved, their willingness to make extraordinary efforts, and the poorly controlled circumstances of each project.

Note for Argentinian, Australian, Brazilian and South African readers: for "October", read "April".

2 **Managed**: at this level there is a real process; the organization has defined policies that include a description of the process and plans for carrying it out; it has allocated resources and defined responsibilities to meet these plans; application of the process is monitored, reviewed, and reported to higher management; stakeholders are defined and involved; and a mechanism is in place for configuration management. In other words, the process has been defined and is carefully carried out.

3 **Defined**: this is a managed process (from now on each level assumes the preceding ones) with more systematic procedures. The main difference with the previous level is the mix of *generality* and *tailorisation*: there is a global but customizable process model for the organization as a whole, and the process for any project is customized from it.

Not to be confused with Taylorisation *(which is how critics would characterize the whole thing).*

4 **Quantitatively managed**: in addition to the previous requirements, the process makes extensive use not only of quantitative data (such as measures of costs, development time, reliability and service quality) but of statistical quality control techniques to analyze the data in depth and use the results as part of the process.

5 **Optimized**: this level adds a feedback loop that uses data collected about the projects to question the process and improve it continually, both incrementally and through more innovative changes.

The following table describes, more precisely, what must be achieved at each level (starting at 2 since by definition there is nothing to report at level 1).

Level	**Name**	Generic practices	Process areas
2	**Managed**		Requirements management Project planning Project monitoring & control Supplier agreement management Measurement & analysis Process & product quality assurance Configuration management
3	**Defined**		Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management for IPPD Risk management Integrated teaming Integrated supplier management Decision analysis & resolution Organizational environment for integration
4	**Quantitatively managed**		Organizational process performance Quantitative project management
5	**Optimized**		Organizational innovation & deployment Causal analysis and resolution

The CMMI defines, for each level, a precise set of goals and practices. We will not go into these here, but perhaps this overview will have given you the incentive (and courage, see the comments below) to go to the CMMI literature and learn the details by yourself. In the process, you will encounter a technique known as the *Personal Software Process*, which applies some of the same ideas to the work of individual developers.

19.9 FURTHER READING

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli: *Fundamentals of Software Engineering, 2nd Edition*, Prentice-Hall, 2002.

A well-known software engineering textbook, providing excellent coverage of the field. Other good textbooks are by: S.L. Pfleeger and J. Atlee (3rd edition, Prentice Hall, 2005); and Roger Pressman (6th edition, McGraw Hill, 2005).

Ghezzi (2008)

IEEE Computer Society (Software Engineering Standards Committee): *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830-1998, available online at ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=15571 (at the time of writing, access requires personal or institutional membership).

A short standard describing best practices for writing requirements documents, included a recommended document structure that is widely applied in industry.

Mandrioli (2008)

Bertrand Meyer: *On Formalism in Specifications*, in IEEE Software, vol. 3, no. 1, January 1985, pages 6-25. Available online at se.ethz.ch/~meyer/publications/computer/formalism.html.

An old article explaining why it is useful to rely on mathematical techniques to express specifications (requirements).

John V. Guttag and James J. Horning: *The Algebraic Specification of Abstract Data Types*, in *Acta Informatica*, vol. 10, pages 27-52, 1978.

A seminal paper on the theory of abstract data types, underlying object technology. Introduces the notion of "sufficient completeness".

Karl E. Wiegers: Software Requirements, Microsoft Press, 2003.

A repertoire of useful rules for writing good requirements documents.

Michael Jackson: *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*, ACM Press, Addison-Wesley, 1995,

An excellent discussion of requirements challenges and techniques.

Horning (2007)

Axel van Lamsweerde: *Requirements Engineering*, Wiley, 2009.

Another excellent book on requirements, the most recent, by one of the authorities in the field. Strong on both theory and examples.

Bertrand Meyer and Jim Woodcock (editors): *VSTTE (Verified Software: Theories, Tools, Experiments)*, LNCS 4171, Springer-Verlag, 2008.

Proceedings of a 2005 conference at ETH Zurich, which launched Tony Hoare's "*Grand Challenge*". Provides a good assessment of the state of the art in program verification.

Frederick P. Brooks: *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, Addison-Wesley, 1995 (the original edition is from 1975, same publisher).

At IBM Fred Brooks directed the development of OS/360, one of the first examples of a complex operating system available across a whole line of computers. This book, where he summarized his experience through short individual essays, has to be mentioned here since it is widely considered a classic in software engineering, although that is more for its folksy advice than for any contribution of substance.

Brooks (2007)

Software Engineering Institute: *Capability Maturity Model Integration (CMMI) Overview*, online document at www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf.

Presentation slides providing a short overview of CMMI.

Software Engineering Institute: *Capability Maturity Model® Integration (CMMI^{SM}), Version 1.1, CMMI^{SM} for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1) Staged Representation CMU/SEI-2002-TR-012 ESC-TR-2002-012*. Sorry, I do not make those titles. Available online at tinyurl.com/kf9uy (shorthand for www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf#search=%22cmmi%20staged%20representation%22).

This is the official, detailed description of CMMI, staged representation. (Continuous variant at tinyurl.com/gjla9; the two documents share a large amount of material.) You will need to gear yourself up for the delicate charm of Government-Committee English, an acquired taste and probably not quite what your creative writing instructor had in mind when exhorting you to be concise, concrete and clear. A typical sample:

> *The plan for performing the organizational process focus process, which is often called 'the process-improvement plan,' differs from the process action plans described in specific practices in this process area. The plan called for in this generic practice addresses the comprehensive planning for all of the specific practices in this process area, from the establishment of organizational process needs all the*

> *way through to the incorporation of process-related experiences into the organizational process assets*"

Wow! Once you get used to the style you will in fact find, like gems in the rubble, a concentrate of some of the best project organization practices that have emerged from four decades of software project management experience.

Watts S. Humphrey: *PSP: A Self-Improvement Process for Software Engineering*, Addison-Wesley, 2005

Describes the Personal Software Process, a personal discipline for programmers applying sound rules of engineering practice and derived in part from CMMI ideas of accountability and reproducibility.

Humphrey (2007)

19.10 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Software engineering encompasses programming but also all the other activities, technical or not, involved in producing software systems. It focuses on industrial software production with defined standards of quality.
- Software engineering involves five major task categories, as captured by the acronym DIAMO: Describe, Implement, Assess, Manage and Operate.
- Issues of software engineering affect both the development *process* and the resulting *products*.
- Product and process quality involves many factors, from correctness and efficiency to cost effectiveness and reproducibility.
- A software project should have a clear view of who its *stakeholders* are and which goals are important for each category of stakeholders.
- Software development includes a number of clearly defined tasks, which *lifecycle models* attempt to organize sequentially. *Agile methods* put less emphasis on the process and more on working code and human interaction.
- The analysis of system requirements is an essential task for any project. Requirements analysis calls for precision, for a description of system properties free of any early commitment to implementation, for a clear view of stakeholders' needs, for realism and for traceability.
- System requirements include functional aspects, specifying the system's functions, and non-functional aspects such as performance constraints. An IEEE standard exists for structuring requirements documents.
- Domain properties reflect external constraints; machine properties express decisions about system properties.
- Verification and Validation can use dynamic techniques, particularly testing, and static techniques such as design and code reviews, static analysis, correctness proofs and model checking.

- The purpose of testing is to cause failures, revealing faults.
- Capability Maturity Model Integration (CMMI) defines five levels of maturity for an organization's process. At level 5, the highest, the process is defined, documented, measured, reproducible and self-improving.

New vocabulary

Adequacy	**Built-in assessment**	**Correctness**
Correctibility	**Cost control**	**Efficiency**
Extendibility	**Factor (of software quality)**	**Goal (CMMI)**
Lifecycle	**Maintenance**	**Measurability**
Portability	**Practice (CMMI)**	**Predictability**
Process (vs product)	**Process area (CMMI)**	**Product (vs process)**
Production software	**Reproducibility**	**Reusability**
Robustness	**Security**	**Self-improvement**
Software engineering	**Stakeholder**	

The names of some of the quality factors (ease of use, production speed…) retain their meanings from non-technical usage and do not figure in this list.

← *"Components of quality", 19.3, page 705*

Acronym collection

CMM	**CMMI**	**DIAMO**
DOD	**ISO**	**SCAMPI**
SEI	**SPICE**	

19-E EXERCISES

19-E.1 Vocabulary

Give a precise definition of each of the entries in the above vocabulary and acronym list (including each acronym's expansion).

19-E.2 Stakeholders

Are *competitors* stakeholders in a software project? Discuss what part they, or concerns about them, may play in building the software and managing the project.

19-E.3 Better wrong or better late?

The overview of CMMI listed under "Further reading" attributes this comment to an unnamed senior manager (and criticizes it): "*I'd rather have it wrong than have it late. We can always fix it later*". Discuss this statement from a software engineering perspective.

PART VI:

Appendices

This part complements the main material of the book by providing an introduction to four important programming languages: Java, C#, C++ and (briefly) C. The presentations are specifically designed for the readers of this book: they do not describe the four languages from scratch, but assume you are familiar with the concepts developed in the preceding chapters, and emphasize the differences with the Eiffel notation that they use.

The descriptions do not cover the languages in their entirety, but include enough to enable you to start programming in these languages if you have mastered the concepts of the rest of this book. For practical use of a language system (compiler, interpreter, integrated development environment) you will of course need the corresponding reference information, from books or online.

Since not every reader will want to learn every one of the languages, each of the first three appendices can be read independently; this also means, since Java, C# and C++ share a number of characteristics (due to the influence of C and, for the first two, C++), that if you do read all three you will encounter some repetitions. The fourth appendix is a short overview of C presented as a subset of C++, and hence assumes that you have read the C++ appendix.

The last appendix presents basic information about using the EiffelStudio environment, to help with the examples of this book and the Traffic software.

The appendices are followed by the list of picture credits and the index.

A

An introduction to Java (from material by Marco Piccioni)

A.1 Language background and style

Gosling (2007)

Java was introduced in 1995, the result of an internal research project at Sun Microsystems led by James Gosling (other key contributors include Bill Joy, Guy Steele and Gilad Bracha). The language came at just the right time to benefit from two separate phenomena:

- Widespread dissatisfaction, after initial enthusiasm for object technology in the late eighties, with the C++ language (see appendix C), particularly its complexity and the limits of its "hybrid" approach retaining compatibility with the non-object-oriented C language.
- The spread of Internet access and the advent of the World-Wide Web, which seemed to call for a universal mechanism to execute programs securely from within browsers.

The Java project, initially intended for "set-top boxes" and network appliances, was ready to support such programs, called *applets*. As noted in an earlier chapter, applets never became the dominant computing model as prophesied at the time, but Java usage quickly expanded to many other application areas.

← "Virtual machines, bytecode and jitting", page 333.

The following properties characterize the Java programming model:

- A close connection between the programming language and a computing platform based on a virtual machine, called a JVM (Java Virtual Machine).
- An emphasis on portability captured by the slogan "Write Once, Run Anywhere" and based on compilation to the JVM's bytecode, which can then be interpreted, or compiled to machine code, on many platforms.
- Syntax, general language style and basic instructions taken from the C-C++ line of languages.
- A strongly typed object-oriented model, with many of the mechanisms studied in this book: classes, inheritance, polymorphism, dynamic binding, genericity (added in recent versions). Some missing elements are multiple inheritance (except for "interfaces", as we will see), contracts and agents; in addition, the O-O part of the type system does not include primitive types.
- Beyond the language proper, a rich set of libraries supporting software development in many application areas.

A.2 OVERALL PROGRAM STRUCTURE

We first review the general structure of a Java program, starting with an overview of the Java Virtual Machine.

The Java Virtual Machine

A Java Virtual Machine is a software system providing mechanisms to support the execution of Java programs. (We may talk of "*the*" JVM as the general specification of these mechanisms, and "*a*" JVM as one particular implementation.) Here are the principal mechanisms:

- A **class loader** manages classes and libraries in the file system and dynamically loads classes in bytecode format.
- A **verifier** checks that bytecode satisfies fundamental constraints on reliability and security: type safety (non-null references always lead to objects of the expected types); information hiding (feature access observes visibility rules); branch validity (branches should always lead to valid locations); initialization (every data element is initialized before use).

← "Null" is the same as "void": see "Void references", 6.3, page 111.

- An **interpreter**, the software equivalent of a CPU in a physical computer, executes bytecode.
- A **Just In Time compiler** (JIT compiler or "jitter") translates bytecode into machine code for a specific platform, performing various optimizations. The most widely used JIT compiler is "Hot Spot" from Sun.

← "Virtual machines, bytecode and jitting", page 333.

Packages

Java programs, like those in other object-oriented languages, are structured into classes, but Java offers a modular structure above the class level: the package. A package is a group of classes (like Eiffel "clusters", which are not a language mechanism but an organizational concept).

Packages fulfill three main roles. The first is to help you structure your systems and libraries. Packages can be nested, and hence make it possible to organize classes in a hierarchical structure. This structure is conceptual, not textual; in other words, you will not declare a package as such (with its constituent classes), but instead declare classes and in each of them indicate the name of its package if any:

```
package p;
class A {… Declarations of members of A …}
class B {… Declarations of members of B …}
… Other class declarations …
```

If this is the content of a source file, all classes given belong to package p. The package directive, if present, must be the first line in the file. Nested packages use the dot notation: p.q is the sub-package q of p.

The package directive is optional; in its absence, all classes in a file will be considered to belong to a special default package.

The second role of packages is as compilation units. Rather than compiling classes individually, you can compile an entire package into a single "Java Archive" (JAR) file.

In their third role, closely related to the first, packages provide a *namespace* mechanism to resolve the class name conflicts that may arise when you combine libraries from different providers. When referring to a class A belonging to a package p, you may always use the fully qualified name: p.A. This technique also applies to classes from subpackages, as in p.q.r.Z. To *avoid* full qualification, you may use the import directive: writing

```
import p.q.*;
```

allows the rest of the file to use the classes from p.q without qualification, as long as this does not create any conflict. (The asterisk * means "all classes from the package", not including subpackages.) Fully qualified notation remains available to resolve ambiguities.

The package mechanism comes with some methodological recommendations. One recommendation is to use it in its explicit form: include every class in a named package (in other words, do not rely on the default package). Another follows from the observation that packages and namespaces only push the name clash problem one step, since you can still have clashes between *package* names. To minimize the likelihood that this will happen, a standard convention for packages uses names that start with the institution's Internet domain, listing components in reverse order; for example a package originating with the Chair of Software Engineering at ETH Zurich (our group, domain name se.ethz.ch) might be called

```
ch.ethz.se.java.webtools.gui
```

Program execution

From a command line, the command to start the execution of a Java program is:

```
java C arg1 arg2 …
```

where C is the name of a class and the optional arguments arg1 arg2 … are strings. The effect is to execute a method (routine), which must be present in C under the name main:

```
public static void main(String[] args) {
    … Code for main program…
}
```

Unlike in Eiffel, this does not create an object since a "static" method (as explained below) does not need an object. Of course main will usually create objects, or call other methods that create objects. The optional formal argument is an array of strings (String[]), corresponding in the above call to arg1 arg2 … The public qualifier, also studied below, makes main available to all clients.

← "System execution", 6.8, page 130.

→ "Static members", page 753.

A.3 BASIC OBJECT-ORIENTED MODEL

We now take a look at the basic object-oriented mechanisms of Java; the discussion assumes familiarity with the concepts of the preceding chapters.

The Java type system

Most Java programmer-defined types will, as in most examples of this book, be reference types, each based on a class. At the top of the class hierarchy stands a class called Object (think of *ANY*, but there is no equivalent to *NONE*).

← "Overall inheritance structure", 16.10, page 586.

A major difference with the type system assumed in the rest of this book affects basic types. In Eiffel, and in C# as studied in the next appendix, every type is based on a class; this includes basic types describing arithmetic, boolean and character values. Java, in contrast, follows C++ in treating a number of basic types as predefined outside of the object-oriented type system. Java has eight such types, known as "primitive types":

- boolean.
- char, representing 16-bit unicode characters.
- Integer types: byte, short, int and long, respectively representing 8-bit, 16-bit, 32-bit and 64-bit integers.
- Floating-point types for real numbers: float (32-bit) and double (64-bit).

You cannot use the corresponding values, such as integers and characters, directly as objects, for example in a data structure described by a generic class which could be used with arbitrary generic parameters. You will have to wrap, or "box", the values into objects. Java provides a set of boxing classes: Boolean, Character, Byte, Short, Integer, Long, Float, Double. (The language is case-sensitive, so Byte is distinct from byte.) So with the declarations

```
int i;                    // Primitive
Integer oi ;              // Wrapper
```

you can convert back and forth between the primitive and object forms:

```
oi = i;          // Abbreviation for oi = Integer.valueOf(i);
i = oi;          // Abbreviation for i = oi.intValue()
```

As the comments indicate, the assignments require calls to conversion functions between primitive (unboxed) values and their boxed equivalents, but you do not need to use these functions explicitly; this is known as *autoboxing*.

The expression oi.intValue(), in the expanded form of the last example, illustrates another difference with the concepts of this book: Java does not apply the Uniform Access principle. A function without arguments, such as intValue in Integer, will be called with an empty argument list as above, clearly distinguishing it from an attribute.

← *"Touch of Methodology: The Uniform Access Principle", page 246.*

Classes and members

A class contains *members*, the Java term for the features of a class. A member can be a field (attribute), a method (routine), or a constructor (creation procedure). A class text may also contain an initializer: an anonymous block of code invoked at initialization time. The following class text contains examples of all these categories:

```
class D {
    String s;                    // Variable field
    final int MAX = 7;           // Constant field

    T func (T1 a1, T2 a2){
            // Method with two arguments of types T1 and T2
            // returning a result of type T.
        … Code for func …
    }
    void proc(){
            // Method with no arguments.
        … Code for proc …
    }
    D(){
            // Constructor: same name as the containing class, no
            // return type.        … Code for constructor …
    }
    D (T1 a1){
            // Another constructor, with one argument
        … Code for constructor …
    }

    {
            // Initializer
        … Initializer code…
    }
}
```

Information hiding

A class member has an export status, which must be one of the following four, listed in order of decreasing accessibility:

- public: available to any client.
- protected: available to the class itself, others in its package, and descendants of the class, but not to any other classes.
- package (not a keyword, but the default): available to classes in the package.
- private: available only to the class itself.

These qualifiers also apply to classes, in particular because Java supports class nesting. For a top-level class (not nested in another), the only possibilities are the default (the class is accessible to others in the same package) and public.

Because of the absence of support for Uniform Access, the export status does not mean the same as what we have seen in this book. Exporting a field member, in any of the first three cases above, gives the corresponding clients *writing* privileges as well as read access. This means that you can directly access fields of remote objects:

```
x.a = b;
```

This contradicts the principle of information hiding, and leads to the common methodological practice of *never* exporting fields, keeping them private instead and equipping each of them with a getter function and a setter procedure.

← “Setters and getters”, page 248.

Static members

Another Java concept departing from the strict object-oriented style used in this book is the support for static members.

To access a class member you will normally need a target object, and will use the standard object-oriented notation target.member (possibly with arguments) where target denotes an object. The keyword this denotes the current object (**Current** in Eiffel).

Java also makes it possible to declare *static* members, which do not require a target object and are called with the syntax C.member where C is the name of a class. The declaration of a static method may not use any non-static methods or fields (since they would require a target object, which the calling method does not provide).

The main program, main, must be static as noted above; the reason is that at the start of execution no object exists yet to call a method (unlike in Eiffel, where execution consists of creating an object and calling a creation procedure on it).

← Page 750.

Abstract classes and interfaces

You may mark a method as abstract to indicate that the implementation will be provided in descendant classes. The class in which such a declaration appears must also be declared abstract:

```
public abstract class Vehicle {
    public abstract void load (int passengers);       // No method body.
    …Declarations of other members (abstract or not) …
}
```

This corresponds to Eiffel’s **deferred** features and classes, without the ability to equip the features with contracts.

Another difference with the deferred class mechanism is that abstract classes — like other Java classes, as we will see in reviewing the Java inheritance mechanism — can only participate in *single* inheritance: a class may inherit from at most one other, abstract or not. This makes it impossible, using classes only, to combine two or more abstractions into one. To ease the restriction, Java provides another form of abstract module: the **interface**. An interface is equivalent to an abstract class whose members are all abstract methods (constants and nested types are also permitted). The declaration of an interface looks like this:

← See the discussion in "Deferred classes and features", 16.5, page 565.

```
interface I {
        // Constants
    int MAX = 4;

        // Abstract methods
    void m1(T1 a1);
    String m2();
}
```

Note that the declarations only specify names and signatures, plus values for constants. All methods of an interface are automatically abstract and public, and all attributes public and static constants.

Classes may *implement* one or more interfaces, as in

```
class E implements I, J{
    void m1(T1 a1) {… Implementation of m1 … }
    String m2() … Implementation of m2 … }
    … Implementations of the members of J (assumed to be another interface)…
    … Other members of E …
}
```

Overloading

It is possible for a Java class to have two methods with the same name as long as their *argument signatures* differ: they have a different number of arguments, or the same number with at least one different type, or the same types in a different order. This is known as method overloading.

The convention for object creation, discussed next, fundamentally relies on overloading: all the *constructors* of a class (its creation procedures) have the same name, which is also the name of the class.

Outside of constructors, it is preferable to stay away from overloading: within the same scope, different things should have different names.; additionally, in a language supporting inheritance, overloading interferes with redefinition (overriding).

Run-time model, object creation and initialization

The Java run-time model is similar to the model discussed in this book; in particular, Java is designed for automatic garbage collection.

A reference not attached to any object has the value null, which is also the default initialization value for reference variables.

The keyword void is used for something else in Java, as already illustrated: it serves as the return type for methods which do not return a result (procedures).

Programs create objects dynamically through new expressions, as in

```
o = new D (arg1);        // Referring to the earlier class D, specifically
                         // its second constructor from page 752.
```

where o is of type D. If this is the initialization of o, it is common to combine the declaration of o and its creation, since Java does not enforce Eiffel's separation between declarations (static) and instructions (dynamic):

```
D o = new D (arg1);
```

Unlike Eiffel's **create**, a creation through new always needs to repeat the class name.

A new expression such as the above refers to one of the *constructors* of the class. As noted, constructors do not have their own names (as other members of the class do) but all use the class name, disambiguated through overloading. Class D as given earlier has two constructors: one with no arguments; one with ← *Page 752.* a single argument of type T1, which the above creation instruction will use provided arg1 is of type T1 or a descendant (otherwise the instruction is invalid).

The reliance on overloading can be constraining; for example it is impossible to achieve the equivalent, in a class representing points in two-dimensional space, of two creation procedures with different semantics, *make_cartesian* and *make_polar* (to define a point through cartesian or polar coordinates), which happen to have the same signature. You would need to add an argument for the sole purpose of disambiguating calls.

It is possible for a class not to declare any constructors; in this case it is considered to have a “default constructor” with no arguments and an empty body.

The creation process is complex. The full effect of a creation instruction such as the above is to execute the following sequence of steps:

I1 Allocate space for an object of type D.

I2 Recursively perform steps I3 to I8 with respect to D’s parent. (D as given has no explicit parent, hence the implicit parent Object, but if it named a parent class the steps would be executed for that class including, recursively, for its own ancestors up to Object.)

“Parent” in the singular because of single inheritance

I3 Set all static fields to their defaults.

I4 Set static fields to values, if any, stated in their declarations (as in static int n=5;).

I5 Execute all static block initializers.

I6 Set all non-static fields to their defaults.

I7 Set non-static fields to values, if any, stated in their declarations.

I8 Execute all non-static block initializers.

I9 Invoke a parent constructor.

I10 Execute the body of the constructor.

Step I9 reflects the Java rule that every object creation must invoke a parent constructor in addition to a constructor of the given class. (Either or both of these constructors may be a default constructor.) The rule is recursive, so this chain of constructor calls goes all the way up to Object. The choice of the parent constructor is as follows:

- The local constructor’s text may have, as its first instruction, a call to the special method super, with arguments if needed. The keyword super denotes the parent class, so this will result in a call to the appropriate constructor, chosen through overloading resolution.
- Otherwise, the constructor is understood to start with super ();. In this case the parent must have an argument-less constructor (one that it declares explicitly, or the default constructor); the effect of the implicit super instruction is to call that constructor.

  The reason for these rules is unclear. The intent is probably to make sure that an instance of a descendant type also satisfies the consistency constraints defined by proper ancestors. The constructor chain mechanism may be an attempt at achieving such consistency, in the absence of a notion of class invariant to express the constraints explicitly.

The initialization of fields in steps I3 and I6 uses default values, as in Eiffel. Unlike in Eiffel, the rules only apply to fields; local variables must be initialized manually. Compilers must issue a warning if you fail to do so.

Arrays

Java arrays are objects, allocated dynamically as we have done in the rest of this book. To define an array, simply use a declaration such as

```
int[] arr;                          // An array of integers
```

To create the array object, you may use

```
arr = new int[size];
```

where *size* is an integer expression (not necessarily a constant). Unlike in Eiffel, arrays are not resizable.

Array access uses the bracket notation, as in arr[i]; be sure to note that indexing starts from 0, so in the above example the valid indexes range from 0 to *size* – 1. You may assign a value to an array element, as in

```
arr[i] = n;
```

The expression arr.length (length is a read-only field) denotes the number of elements of the array; after the above allocation its value will be *size* + 1 (since *size* will actually determine the highest legal index, and indexing starts at zero) A typical iteration on an array, using the for loop detailed below, is

```
for (int i=0; i < arr.length ; i++)
    {… Operations on arr[i] …}
```

where i++ increases the integer i by 1. Note that the continuation condition i < arr.length reflects that the last legal index is arr.length – 1.

You can have multi-dimensional arrays, in the form of arrays of arrays:

```
int[][][] arr3;                          //Three-dimensional array
```

allowing access of the form

```
arr3[i][j][k]
```

Exception handling

An exception is an abnormal run-time event that interrupts the normal flow of control. Typical causes of exceptions include null pointer dereferencing (void call: x.f where x is null) and integer division by zero. In Java can also trigger a *developer exception* explicitly through

← “An introduction to exception handling”, 7.10, page 200.

```
throw e1;
```

where e1 is of an exception type, which must be a descendant of the Throwable library class. More specifically, this type is in most cases a descendant of Exception, the heir of Throwable that covers programmer exceptions. The other heir, Error, covers system-related run-time errors.

See the inheritance diagram on the adjacent page.

A Java program may *handle* an exception in the following style:

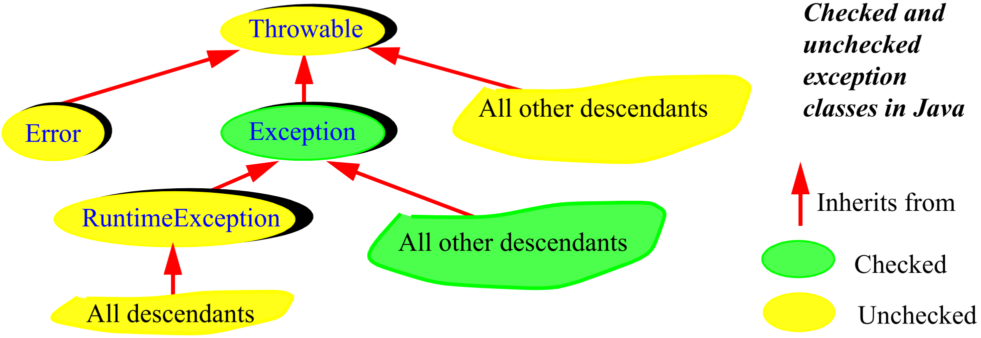
```
try {
    … Normal instructions, during which an exception may occur …
} catch (ET1 e) {
    … Handle exceptions of type ET1, details in e …
} catch (ET2 e) {
    … Handle exceptions of type ET2, details in e …
}… Possibly more cases…
finally {
    … Processing common to all cases, exception or not…
}
```

If the try block triggers an exception of one of the types listed, here ET1, ET2 …, execution will not complete the try block but continue with the corresponding catch block. The finally part is executed in all cases, exception or not; its typical purpose is to release resources, for example to close open files, before moving on.

Any occurrence of an exception produces an *exception object* — an instance of the appropriate descendant of Throwable. The program can access this object, in the corresponding catch clause, through the specified exception name (e in all the catch clauses above). Examples of properties that you can obtain in this way are the human-readable name of the exception and the state of the call stack, although ordinary exception handling seldom needs such information.

If an exception occurs whose type does not match any of the listed types — or it is triggered outside of a try block —, it is passed on to the caller for application of this policy, until a caller up the chain has a suitable catch; if none does, the program terminates in error.

Java introduces an interesting distinction between “checked” and “unchecked” exceptions. The place of exception types in the Throwable hierarchy determines which exceptions are checked, as illustrated by the following figure:



Checked and unchecked exception classes in Java

Checked exceptions provide a contract-like mechanism: the rule is that if a method can throw a checked exception, it must declare it, and then all callers are required to handle it.

To specify that it can throw an exception, the method will use the throws keyword (do not confuse with throw, used by the instruction that actually triggers the exception):

```
public r(…) throws ET1, ET2 {
    …Code for r, including instructions
                throw e1;              // For e1 of type ET1
                throw e2;              // For e2 of type ET2
    …
}
```

If r includes throw e3; for e3 of a checked type ET3, and e3 does not appear in its throws clause, the method is invalid — unless its body contains a try block with a branch of the form catch (ET3 e), ensuring that the exception will be processed within the method rather than passed on to the caller.

With the above declaration, any call to r in another method must be in a try block containing catch clauses for the listed exception types, here ET1 and ET2.

This carefully designed mechanism has attracted praise but also some controversy. A limitation is that one can only force the use of throws specifications for programmer-defined exceptions, whereas the most damaging cases often come from system-related events (void call, integer division by zero…). When the rules do force callers to use a try block, it is easy for a lazy programmer to write a perfunctory catch clause that pacifies the compiler but does nothing, thereby defeating the purpose of the mechanism. This is probably why the C# exception mechanism, otherwise almost identical to Java's, did not retain checked exceptions. Still, checked exceptions encourage a disciplined approach to exception handling; you should take advantage of them if you use programmer-defined exceptions in Java.

→ "Exception handling", page 790 (about the C# exception mechanism, in the next appendix).

A.4 INHERITANCE AND GENERICITY

The original Java design had single inheritance and no genericity. Since then the language has added genericity ("generics" in the usual Java terminology); the limitation to single inheritance remains.

Inheritance

To make a class inherit from another, use the keyword extends. This is distinct from the syntax for inheriting from interfaces, which uses implements. The two can be combined, with extends coming first:

```
public class F extends E implements I, J {…}
```

A class with no extends clause is considered to inherit from Object.

You can declare a class final so that no other is permitted to inherit from it:

```
final class M …
```

There is no equivalent in Java to the **rename** mechanism for resolving name clashes. If two methods inherited from a class and an interface, or two interfaces, have the same name and different argument signatures, this will be considered a case of overloading; if the signatures are the same, the two methods are in conflict and there is no simple way to resolve the issue.

Redefinition

The redefinition of a method is called "overriding". The overriding method may not be static; it must have the same argument signature as the original.

You have to be careful about keeping an identical signature: any change in type or number of arguments would be considered overloading, and so would not produce a compilation error (unless it clashes with another overloading of the same method). This requires particular attention since both overriding and overloading are silent (there is no equivalent to the **redefine** clause): you simply declare in the new class a member with the same name, and depending on how you declare the arguments it could be an override, an overload, or a validity error.

The return type is not part of the argument signature and plays no role in the overloading rules. For an overridden method, it will generally be the same as the original's, but it can also be a descendant of the original. This is known as *covariant* redefinition. (Eiffel has covariant redefinition for both result types and arguments, which raises special issues for the type system.)

The equivalent of the **Precursor** mechanism for accessing the original version of a redefined method is the super construct, which we have already seen for constructors. For example:

```
public display(Message m) {          // A redefinition of an inherited method
    super(m);                        // execute the original's body
    … Other operations, specific to the redefinition…
}
```

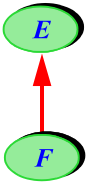
For fields (attributes), using the same name and type in a descendant shadows the original version.

The redefinition of a member may extend its visibility status (going up in the earlier order, for example from private to public), but not restrict it, since clients could then cheat the restriction by going to the parent to access the redefined feature through polymorphism and dynamic binding.

← "Information hiding", page 752.

Polymorphism, dynamic binding and casts

Polymorphism and dynamic binding are the default policy, as presented in the rest of this book. In other words, if e1 is of type E, f1 is of type F, and F is a descendant of E, you may use the polymorphic assignment



```
e1 = f1;
```

after which calls of the form e1.r () will use the F version of m if F overrides the method r.

Polymorphic assignments such as the above are known as *upcasting*. The other way around, the mechanism for *downcasting* (forcing a specific type on an object known through a more general type, as studied in detail in the discussion of inheritance) uses the C syntax for "casts":

← "Uncovering the actual type", 16.13, page 599.

```
f1 =(F) e1;
```

If e1 is attached to an object of type F, this operation will attach f1 to that object; if not, the cast will cause an exception, in accordance with the Casting Principle. You may plan for the possible exception through a try block, but it is better to avoid it through the instanceof operator:

← "Touch of Methodology: Casting Principle", page 601.

```
if ( e1 instanceof F )
     {f1 = (F)e1;}
else
     {… Handle case in which e1 did not denote an F object…}
```

This achieves an effect similar to Object Test, without the notion of scope.

← *"The object test", page 602.*

Genericity

Java genericity ("generics") concepts will be familiar from the discussion of unconstrained genericity. Generic parameters are enclosed in angle (rather than square) brackets <…>. If you declare

← *"Generic classes", page 365.*

```
public class N<G, H> {
     …Class body …
}
```

class N has two generic parameters. A generic derivation (or "instantiation") also uses angle brackets:

```
N<T, U>
```

Like classes, interfaces can be generic.

The closest equivalent to constrained genericity is the ability to declare a formal generic parameter as

```
<? extends V>
```

which means that the corresponding actual generic parameter must be a descendant of V. This is known as a "bounded wildcard".

A significant extension to the genericity mechanism, not present in Eiffel, is that methods, as well as classes, can be generic. You can for example declare

```
public <G> List <G> repeated (int n, G val) {…}
```

With the appropriate body this could be a method that takes as argument an integer n and a value of arbitrary type G, and returns a list containing n items all equal to that value. With this declaration, the expression

```
<String> repeated (27, "ABC")
```

denotes a list of 27 strings, all of them with the content "ABC".

A number of restrictions govern generics:

- You cannot use the primitive types such as boolean and int as actual generic parameters. Instead you have to use their object-oriented counterparts, such as Boolean and Integer, and wrap the corresponding values into objects.

← *“The Java type system”, page 750.*

- Exception classes cannot be generic.
- No static context is possible for generic types.

A.5 FURTHER PROGRAM STRUCTURING MECHANISMS

The control structures of Java are largely inherited from C and C++.

Conditional and branching instructions

Conditional instructions have the form:

```
if(boolean_expression){
      …
}
else{
      …
}
```

You may omit the braces for the then or else part if it consists of a single instruction. (It is better to leave the braces anyway, since adding an instruction later on could lead to unintended semantics.)

There is no equivalent to elseif; you have to use nesting, but the clauses do not have to look nested due to the absence of an end keyword and, visually, the use of comb-like indentation:

← *“Comb-like structure, figure on page 179.*

```
if (expression) {…}
else if (expression) {…}
else if (expression) {…}
…
else statement
```

The closest to a multi-branch instruction is the switch instruction, although it is a multiple-target goto instruction rather than a one-entry, one-exit conditional. The multi-branch instruction has the form

← *"Multi-branch", page 195.*

```
switch (expression) {
    case value: statement; break;
    case value: statement; break;
    …
    default: statement
}
```

where *expression* is of a character type (char or the boxed variant Character) or an integer type (short, byte, int or their boxed variants), and each *value* is a compile-time constant of a compatible type. If the value of the *expression* does not match any of these constants, the instruction executes its default branch if present, nothing otherwise. (In Eiffel, in the absence of an **else** clause in an **inspect**, this case produces a run-time error.)

long *is not permitted. The boxed variants were seen in "The Java type system", page 750.*

To obtain the effect of a multi-branch conditional you must include break; instructions as shown. If you omit them control will flow, when a branch terminates, to the next branch. The break instruction is also applicable to other control instructions: if conditionals, and loops as seen next.

Another branching instruction is continue; you may include it within a loop body to cause the execution to ignore the rest of the body and continue with the continuation test and, if positive, the next iteration.

> As we saw in the discussion of control structures, it is usually best to stay away from such goto-like constructs

← *"The goto puts on a mask", page 189.*

To ensure that a break or continue breaks or continues to the intended place, you may use the form break *label* or continue *label*, where *label* is an identifier; this assumes that you have labeled the corresponding structure:

```
label: … The control structure (if, switch or loop) …
```

While the unlabeled break and continue only break out of the immediately enclosing structure, the labeled form enables you, in a nested control structure, to jump out any number of levels. If you use break and continue at all, make sure to use this labeled form to decrease the likelihood of errors.

Loops

Java provides three kinds of loop:

```
while (boolean_expression) statement
do statement while (boolean_expression);
for (init_statement ; boolean_expression ; advance_statement) body_statement
```

In all these variants, the *boolean_expression* serves as a *continuation* condition; this is the reverse of the convention used by the **from** … **until** … **loop** … **end** form of the rest of this book, where the **until** clause uses an *exit* condition. To go from one style to the other, just negate the condition.

The difference between the two while variants is that the second one always executes the body (*statement*) at least once, since it tests the expression before executing the body (as in a **repeat** … **until** …, with the condition reversed); the first form can have zero executions of the body.

← *"Other forms of loop", page 192.*

The for loop is the most general and the most commonly used. The purpose of the *advance_statement* (included in the body in the Eiffel syntax) is to advance the iteration. So the equivalent of

```
from i := 1 until i > n loop
    …
    i := i + 1
end
```

is, in Java:

```
for (int i = 1; i <= n; i++)
    {…}
```

The language also provides an enhanced form of for loop, simplifying the writing of loops that iterate through containers

```
for (variable: collection){
...
}
```

The equivalent in Eiffel would be obtained through the iteration mechanisms of container classes, relying on agents.

← *"Four applications of agents", page 621.*

A.6 ABSENT ELEMENTS

A number of mechanisms on which we have come to rely are not directly present in Java. Let us explore how to achieve their effect.

Design by Contract

Java has no direct support for Design by Contract (preconditions, postconditions, class and loop invariants). Java 1.4 introduced an assert instruction, used in the form

```
assert boolean_expression;
```

which (similar to Eiffel's **check**) evaluates the *boolean_expression*, does nothing else if the value is true, and otherwise throws an exception of type AssertionError — a descendant of Error and hence not checked. This instruction can be used to insert assertion monitoring at specific points, for example at the beginning of a method body (where a precondition would appear) and at the end (postcondition). But of course this provides only a small subset of the Design by Contract mechanisms, in particular missing the applications to documentation and inheritance and the whole notion of invariant.

← *"Exception handling", page 758.*

Recognizing the importance of this deficiency, many groups have proposed extending the language to add contracts; usually these extensions have experimental implementations that use a preprocessor (a tool that processes the extended language and translates it into standard Java). There are dozens of such proposals; the most widely used is JML, the Java Modeling Language, which has served as the basis for important work on software verification (see the bibliographic section for a reference).

Multiple inheritance

The ability for a class to implement multiple interfaces means that you can combine several abstract types. But you cannot combine several classes through inheritance. We will see below a way to remedy this limitation in part through inner classes.

Agents

Java has no equivalent to the notion of agent as used in this book or to similar mechanisms in other languages, such as C#'s "delegates" studied in the next appendix. Because the strongly typed nature of the language also precludes the use of function pointers as in C and C++, this leaves Java at a disadvantage for applications that need to treat operations as objects: event-driven programming, some numerical applications such as integration, iteration and others that we saw in the discussion of agents.

← *Chapter 17.*

Java of course offers alternatives for such needs. We saw that loops now have a built-in iteration mechanism; for most other cases the recommended Java solution is to use inner classes (as detailed, for the example of GUI programming, in the next section).

For a long time, the Java community denied that anything else was necessary; a 1997 white paper provides a fascinating insight into that view (and more generally into issues of programming language design). In its eagerness to demonstrate that delegates — which were then being proposed for Java, but only made their way into the future C# — are a superfluous mechanism, it provides a number of examples written side-by-side with inner classes and delegates, and in the end succeeds in demonstrating the superiority of the very mechanism that the paper is attempting to dismiss. This will not be a surprise if you remember the earlier discussion of how the absence of agents complicates programs.

java.sun.com/docs/white/delegates.html.

← “A world without agents”, page 623.

Almost a decade and a half later the designers finally relented: it has been announced that Java 7 will include an agent-like mechanism known as *closures*.

A.7 SPECIFIC LANGUAGE FEATURES

Java provides a number of mechanisms that are absent from the basic object-oriented framework presented in this book.

Nested and anonymous classes

It is possible in Java to declare a class within the text of another:

```
public class O {
      class Inner {
            … Members of Inner …
      }
                  … Other members of O, can use class Inner …
}
```

This is known as a “nested” or “inner” class. Within class O, class Inner is just like a member of O — except that it is not a field or attribute, just the name of a class that other members can use, for example to declare variables, and calls methods of Inner on the corresponding objects.

It is even possible, if you only need a class within a specific context, to declare it without a name. Such nested classes are called *anonymous*; we will see an example below.

While you may be wondering about the usefulness of these possibilities, they turn out to have a role in emulating some of the missing mechanisms noted above: multiple inheritance and agents. Let us examine both applications.

Assume first that you would like to write a class R as heir to two classes P and Q. The language permits you to choose only one of them, say P, as the class that R officially extends. For Q, the usual solution is to use the client relation instead; inner classes offer a slight improvement over this technique. You can add to R an inner class S that extends Q; this enables other members of R to use the members of Q through class-qualified notation, S.some_member_of_Q:

```
public class R extends P {
    class S extends Q {              // Class S is inner to R
        … Members of S, including possible overriding of members of Q …
        }
    …Other members of R …
    // Declarations here can use members of S (including those from Q)
    // under the form S.f(…)
}
```

If R is really to offer the functionality of Q, it is often appropriate in practice to add to R, for every public method f of Q, a simple method of the form

```
T f (T1 a1, …)          // Same argument and result signature as Q.f
    {S.f (a1, …)}                                                    [1]
```

The benefits and limitations of this emulation technique are clear. On the positive side, it provides direct access to all the members of the spurned parent (here Q), and allows overriding them. But it can lead to code duplication (if you use scheme [1] above) and does not achieve the symmetry of multiple inheritance. One of the parents is the real one; the other is more like a poor uncle whom you charitably put up in your attic. In particular, you cannot polymorphically use an instance of R as an instance of Q; only P has that privilege. The uncle is not allowed to forget that he does not quite belong.

Consider now the emulation of agents, for example in GUI programming. Assume you want to specify that any occurrence of an event of a certain type, say left-mouse-button click, on a certain GUI element such as a button *OK_button*, should trigger the execution of a certain routine *perform* of your system, which takes as arguments two integers for which it will use the mouse coordinates. We have seen how to do this with agents:

← *"Using agents: the event library", 18.5, page 686.*

```
OK_button.left_click.subscribe (agent perform)                      [2]
```

To understand this discussion you should indeed be familiar with the concepts developed in the earlier discussion of event-driven design.

← *Chapter 18.*

Let us see how to achieve the effect of [2] in Java, using the general approach of Java GUI libraries such as Swing, although not their precise terminology (the goal is not to teach you how to use a particular library — you will find many Web pages for that — but rather to show you, as in the rest of this book, how things actually work under the hood). What you need first is an interface associated with the event type; for example:

```
interface ClickListener {
    void process(ClickEvent e);
}
```

There will be one such interface for every event type. Here it needs only one method, which we call process, denoting the operation to execute in reaction to an event of the given type. The argument e represents an event; when an event occurs, the event producer will create an event object of the appropriate type, here ClickEvent, containing the event arguments, such as the mouse coordinates. We assume that these arguments are accessible, for an event e, as e.args.

As the name of the interface suggests, its instances (that is to say, instances of its descendant classes) represent objects that have subscribed to the corresponding event type; as we saw, "listener" is just another term for what is also called an *observer* or a *subscriber*.

← *"Definitions: Trigger, publish, publisher, subscriber", page 667.*

Internally, the overall mechanism ensuring that subscribers are notified of events will be the same as what we saw for the Observer pattern and, in a simpler form, the Event Library: for each event type, keep a list of subscribers; when an event occurs, traverse the subscriber list so that each can execute its subscribed operation. All that we need to see is how such a class, say U, subscribes its own operation to the event type. This is where we can use an anonymous nested class:

```
class U {                                                        [3]
    Button b;
    build () {
        okButton=new Button(…);
        okButton.addListener(
            new ClickListener(){
                public void process(ClickEvent e){
                        // Code to be executed for e, for example:
                    perform(e.args);
                }
        );
    }
    …Other members of U …
}
```

The basic scheme for adding a listener is the same as in the earlier chapter: add an element to the list of observers (listeners), here under the form okButton.addListener (obs). In the Observer pattern, obs was an instance of an observer class, and we had to define such a class for every possible kind of event type and observer. The agent mechanism simplified this drastically by enabling us to write obs as just **agent** *perform*, where *perform* encapsulates the routine we want to call, together with any associated objects. The scheme here is closer to the Observer mechanism, but with a significant improvement: we do not need to encumber our system with a new class fulfilling a local role only; instead, we may use an anonymous nested class.

← *"The observer pattern", 18.4, page 678.*

Normally, new would not be applicable to ClickListener since this is an interface; we would have to define a class that implements ClickListener, and make obs an instance of that class. Since we only need such a class in the context given, we define it inline and instantiate it right away; this is what the highlighted part of the preceding code achieves. Since the class will not be used anywhere else it does not need a name.

To be a valid descendant class of ClickListener (an *effecting* in the terminology of earlier chapters), all the class needs to do is implement the interface's single method, process. Our implementation of process calls the desired routine from our system, perform, passing it the event's arguments.

← *"Deferred classes and features", 16.5, page 565.*

The benefit is that you can use the needed class within the desired scope as you would a normal class, but without polluting the global name space. In addition, inner classes have access to all the members of the enclosing class, including private members; this can be useful for example if U is part of the application's GUI and the subscriber code, here process, needs to change elements of the user interface.

This technique, however, essentially addresses one of the limitations of what was earlier called "a world without agents": the "Many Little Wrappers" syndrome, forcing you to wrap operations into small classes. Here we do not need to give first-class status to these classes since they remain local and anonymous; but we still need to define one (typically very small) interface per event type.

← *"A world without agents", page 623. On Many Little Wrappers see page 626.*

In addition we cannot directly reuse classes from the application (the "model" in terms of the earlier discussion), such as perform; we have to wrap them into "glue code" such as process. We also lose the flexibility of mixing open and closed arguments in an agent.

← *"The model and the view", page 675.*

It is not necessary, however, to belabor the point: a mere look at the texts of [2] and [3] shows eloquently enough the difference in expressiveness— and, as noted, the Java community seems by now to have got the message.

Type conversions

Java provides a uniform framework for converting values between different primitive types.

Where there is no loss of precision — in converting from byte to short, short to int, int to long, char to int, int to double and float to double — you can use a plain assignment, for example l = s; where s is of type short and l of type long.

This possibility also applies (as in most programming languages) to some cases in which the loss of precision is considered acceptable: int to float, long to float, long to double.

For cases with a more significant loss of precision, such as converting from a floating-point type to an integer type (which requires rounding or another approximation), you must use explicit cast syntax as a way to assert that you know what you are doing. As an example:

```
float s;                        // Single-precision floating-point
double d = 9.9;                 // Double-precision floating-point
s = (float)d;
```

Autoboxing, as we saw, provides automatic conversions between primitive and wrapper types. Unlike in Eiffel, it is not possible to define conversions between arbitrary types.

← *"The Java type system", page 750.*

Enumerated types

Enumerated types make it possible to define variables that range over a finite set of predefined values, as in

```
enum CardColors {Spades, Hearts, Diamonds, Clubs}
```

You can denote the values by prefixing them with the type name and a dot, as in CardColors.Spades.

Internally, a type such as CardColors is a class that extends the library class Enum and has four instances. Since it is a class, you may add constructors, attributes and methods.

Varargs

From version 5.0 on, you can design a Java method to handle a variable number of actual arguments, or “varargs”, without explicitly using an array or a collection (the standard technique in the absence of a specific language mechanism). The convention is simple:

```
public void m(T1 a1, T2 a2, String … s)
```

which indicates that after the first two arguments come any number — zero, one or more — of String arguments. A method may have at most one such “varargs” argument, and it must appear at the end of the argument list.

The effect is the same as if that last argument were an array (here an array of String); the routine can then use array features: s.length to find out how many values were actually passed in a particular call, and s[i] (for i ranging from 0 to s.length) to access them.

Annotations

Also introduced in version 5.0, annotations provide a mechanism for adding structured information to Java programs. The intent is similar to that of Eiffel’s **note** construct, and of C# attributes studied in the next chapter.

→ “Attributes”, page 801 (on the C# mechanism).

Information contained in annotations should not affect the basic semantics of the program (that is the purpose of the other language constructs), but it can be of interest to other tools, for example project management tools.

An annotation is based on an interface, declared with the keyword variant @interface. You might for example want a standard way of equipping classes with basic version control information: an author name, a modification date, and optionally a revision number, all strings. You may define an annotation interface:

```
public @interface ChangeInfo {
    string author;
    string last;
    string revision;
}
```

Then you can equip a class or (here) a method with version information:

```
@ChangeInfo{
    author=     "Caroline",
    last=       "24 December 2009",
    revision=   "6.7"
} public void r {…}
```

As part of Java's *reflection* library, which provides programs with information about their own structure, you may then access annotations associated with a software element through the expression

```
x.getAnnotations()
```

where x (obtained through reflection) represents a class or a method.

A.8 LEXICAL AND SYNTACTIC ASPECTS

Java uses the Unicode character set. Like Eiffel and most other modern languages, Java is "free-format": break characters (blanks, tabs, new lines) are all equivalent and only serve to keep tokens separate.

← *"Breaks and indentation", page 45.*

Unlike in Eiffel, identifiers are case-sensitive. They can be of arbitrary length but may not start with a digit, include / or –. As a style custom rather than a language rule, Java identifiers generally use "camel case", as in aCamelCaseName.

← *"Touch of Style: Choosing your identifiers", page 45.*

Comments have two forms: //, used liberally in the examples of this chapter, introduces a comment that extends to the end of the line; you can also write a comment over any number of lines by enclosing it in /* and */. In this second form, you may start a comment by /** to indicate that it is specifically intended for the Javadoc program documentation tool.

Just in case you have not noticed (then you should probably start reading this appendix again, more carefully this time), Java retains two basic syntax conventions from C and C++: ending every declaration and instruction with a semicolon (a terminator rather than a separator); and using = for assignment, whereas == is the symbol for equality.

Keywords

The following names are reserved in Java:

> abstract, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

The names const and goto appear in this list even though the language does not currently use them.

Operators

These are the Java operators:

Access, call: . [] ()	Postfix: expr++ expr--
Other unary: + - ~ ! new ()	Prefix: ++expr --expr
Arithmetic: * / %	Additive: + -
Shift: << >> >>>	Relational: < > <= >= instanceof
Equality: == !=	Logic: & ^ | && ||
Ternary: cond ? expr1:expr2	Assignment: = += –= *= /= %= &=
Assignment: ^= |= <<= >>= >>>=	

A.9 BIBLIOGRAPHY

James Gosling, Bill Joy, Guy Steele and Gilad Bracha: *The Java Language Specification*, third edition, Addison Wesley, 2005.

> As is often the case, this description of the language by its main designers beats most derivative works. A must-read for anyone interested in Java.

Joshua Block: *Effective Java*, second edition, Prentice Hall, 2008.

Bruce Heckle: *Thinking in Java*, fourth edition, Prentice Hall, 2006.

Cay S. Hearthstone and Gary Corneal, *Core Java, Volume 1 (Fundamentals)*, eighth edition, Prentice Hall, 2007.

> Three widely used textbooks.

java.sun.com/reference/docs

> The online documentation from Sun.

www.eecs.ucf.edu/~leavens/JML/

> This is the home page for JML, from which you will find references to numerous articles on this Design by Contract extension for Java.

B

An introduction to C# (from material by Benjamin Morandi)

On introducing C# in 1999, Microsoft presented the language as follows:

> *The ideal solution for C and C++ programmers would be rapid development combined with the power to access all the functionality of the underlying platform. They want an environment completely in sync with emerging Web standards and providing easy integration with existing applications. Additionally, they would like the ability to code at a low level when the need arises.*
>
> *C# is a modern, O-O language that enables programmers to quickly build a wide range of applications for the new .NET platform, which provides tools and services that fully exploit both computing and communications.*
>
> *Because of its elegant O-O design, C# is a great choice for architecting a wide range of components — from high-level business objects to system-level applications. Using simple C# language constructs, these components can be converted into XML Web services, then invoked across the Internet from any language running on any operating system.*

See tinyurl.com/dkeeur (archive of msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp). For a more recent introductory description see msdn.microsoft.com/en-us/vcsharp/bb466176.aspx.

Most readers of this book presumably prefer English, so here is a translation: "C# is Java plus *delegates* (routine objects, in the spirit of agents) and a few low-level mechanisms brought over from C++." C# was Microsoft's response in its rivalry with companies supporting Java, particularly Sun Microsystems and IBM, and the language was extremely close to Java.

← Agents were discussed in chapter 17.

This characterization remains largely applicable today, although C# has evolved as a design of its own and introduced a number of interesting innovations with no direct Java counterparts. C# in its version at the time of writing (3.0) is a rich language, of which we will only survey the essentials.

To learn C#, knowledge of Java is useful but not required; this appendix covers C# on its own and does not assume that you have read the description of Java in the previous appendix. (As a consequence it repeats some of its observations, when covering shared concepts.) Like other language-specific appendices, it does not start from scratch but assumes that you are familiar with the programming concepts introduced in this book, occasionally describing C# by comparison with corresponding Eiffel mechanisms.

Anders Hejlsberg (C# designer), 2007

B.1 LANGUAGE BACKGROUND AND STYLE

C# (pronounced *C sharp*, as in music) is closely connected with Microsoft's .NET environment, a platform for developing and running software using a virtual machine. In an earlier discussion we saw the role of virtual machines and their benefit for implementing higher-level languages.

← *"Virtual machines, bytecode and jitting", page 333.*

.NET, the CLI and language interoperability

While the Java Virtual Machine was designed specifically to support the Java language (although it was later used to implement other programming languages), it was a central design goal of the .NET platform, from the beginning, to support several languages; the name of the virtual machine, *Common Language Runtime* (CLR) and its supporting interoperability API, *Common Language Infrastructure* (CLI, now an international standard), reflect this decision.

CIL (Common Intermediate Language) is the name of the .NET bytecode.

Part of the reason was that Microsoft, pre-.NET, already provided implementations of several languages, notably Visual Basic (a mass-market offering often used by non-professional programmers to develop simple applications, as well as a few less simple ones), C++, and JScript for client Web applications. The company could not ask all the corresponding programmer communities to drop their favorite languages and move to a brand new one. It could, however, provide a common base for interoperability and future evolution. .NET and the CLR/CLI were able from the start to provide implementations of four Microsoft-supported languages (the three ones mentioned plus C#) as well as languages developed by third parties, including Eiffel (from the very introduction of .NET in 1999) and Cobol, the latter a legacy language still important for many business applications.

Language openness goes beyond the availability of compilers for several languages: it implies a high degree of interoperability between programs written in these languages. This is the role of the Common Language Infrastructure: to provide a standard set of mechanisms that can be used from any language. The CLI is, more specifically, an *object model*, akin to an object-oriented language without the syntax. The object model specifies a set of mechanisms in a precise way; these are the object-oriented mechanisms studied in this book — classes, features (members), inheritance, genericity, a type system, objects, a policy for dynamic object creation and garbage collection — for which the CLI design makes a number of specific design choices.

As long as .NET languages do not depart too much from those decisions, they can reach a degree of interoperability unheard of in pre-.NET days; in particular, classes written in various object-oriented languages can easily interface with each other, through both the client and inheritance relation; for example an Eiffel class can inherit from a C# class and conversely. This simply assumes that the compilers enforce the CLI rules for "producers" and "consumers" of *assemblies* (the target modules produced by .NET compilers).

This interoperability scheme has proved quite successful (in spite of a recent and worrying tendency of software producers to ignore the CLI compliance rules). It allows each language to retain its specificity as long as it can map its object model to CLI; for example, Eiffel's implementation must emulate multiple inheritance — not directly supported by the CLI — through special use of CLI mechanisms (multiple inheritance from *interfaces*, a concept discussed later in this appendix).

The favorite son

In this society of languages some are more equal than others. C# is the favorite son, with an object model closely reflecting the CLI. (Visual Basic .NET, which resembles previous versions of Visual Basic in syntax only, is a close contender for the title. "Managed C++", the CLI-compliant version of C++, departs significantly from the usual C++ to participate in the .NET interoperability game.) C# indeed took its essential semantics from the CLI, although subsequent versions extended it considerably. The syntax is in the C-C++-Java tradition, with semicolons as instruction terminators (rather than separators) and braces to enclose program blocks.

B.2 OVERALL PROGRAM STRUCTURE

The basic elements of a C# program are classes and structures (or "structs"), organized in a number of program files.

Classes and structs

C# classes and structs are descriptions of a set of possible run-time objects to which the same *members* (features) are applicable. The general form of a declaration is

```
class name {
    …Member declarations …
}
```

with the keyword struct instead of class in the case of a structure declaration.

A member is one of the following: field (corresponding to Eiffel attributes); constant; method (routine); property (field equipped with a getter and setter); operator (function with operator syntax); constructor (creation procedure); destructor (associated with garbage collection, and applicable only to classes, not structs); event (in connection with delegates and event-driven programming); indexer; and nested type. We will review the most important below.

A struct is a simplified form of class, with no possibility of inheritance. The rest of this discussion focuses on classes, but most non-inheritance-related properties also apply to structs.

Classes and structs are grouped into *assemblies* (corresponding to the notion of cluster in Eiffel).

Program execution

Every executable program must have at least one method, called Main and marked as static (a notion explained in the next section). Program execution consists of executing that method. You can write the proverbial "Hello world" as a single class with a Main method:

```
public class Program {
    static void Main(string[] arguments) {
        System.Console.WriteLine("Hello world!");
    }
}
```

Main may have no arguments or, if the execution needs user-provided arguments, it should (as in this example) take an array of strings (string[]) as its single argument. It may return no result as here, or it may be a function returning an integer, typically a status code indicating possible errors.

B.3 BASIC OBJECT-ORIENTED MODEL

Many of the concepts of C# resemble what we have seen throughout this book, but C# introduces a number of variations.

Static members and classes

One of the C# concepts departing from the strict object-oriented style used in this book is the support for static members and classes.

To use a class member you will normally need a target object, and will use the standard object-oriented notation target.member (possibly with arguments) where target denotes an object. To denote the current object (**Current** in Eiffel) use the keyword this.

C# also makes it possible to declare *static* members, which do not require an object and are called with the syntax C.member where C is the name of a class. The definition of a static member, for example a static method, may not use any non-static member.

A class as a whole can be declared static class… if all its members are static; then it is not possible to create instances of the class. A static class may be convenient, for example, to group a set of object-independent general facilities, such as mathematical functions which do not use an object-oriented design.

It was noted above that Main must be static; the reason is that at the start of execution no object exists yet to call a method.

Eiffel addresses the problem by defining execution as the creation of a "root object" to which it applies a "root procedure".

Export status

For information hiding, every type and member has an accessibility level, defining clients' access rights. The goal is the same as Eiffel's information hiding mechanism, including selective exports, but with a coarser granularity since in C# you cannot make a feature accessible to a specified list of classes. The three possible qualifiers are:

- public: accessible by any code.
- internal: accessible by code in the same assembly.
- private: accessible by code in the same class or struct. This is the default for class members.

Some restrictions apply: a type can only be internal (the default) or public, unless it is a nested class (a class declared within another), which can also be private; destructors may not have access modifiers; a user-defined operator must be public; the accessibility of a member must be at most equal to the accessibility of the types used in its declaration. It is not hard to see the justification behind each of these rules.

Fields

C# fields correspond to attributes.

The rest of this book has used more specific terminology: a field (dynamic notion) is a constituent of an object; it corresponds to a feature of the generating class, called an attribute (static notion). In C# "field" covers both.

The declaration of a field gives the type (before the field name, as in T f, rather than *f*: *T* in Eiffel); it can optionally include an initial value, using the assignment symbol =. Such an initial value may not refer to other instance fields of the current object, because they might not be initialized yet. Here is an example with two fields:

```
class A {
    public string s1 = "ABC";
    public readonly string s2 = "DEF";
    … Other member declarations …
}
```

Note the semicolon terminating all declarations and instructions. The readonly qualifier in the second example prohibits assignments to the field, except in its declaration as here or in a constructor (creation procedure).

Unlike in Eiffel, exporting a non-readonly member (through public or internal) gives the corresponding clients writing privileges as well as read access. For a reasonable application of information hiding, this means that fields should usually be private, with associated setters and getters. C# simplifies the writing of setters and getters through the notion of *property* studied below.

← *"Setters and getters", page 248.*

Basic types

C# provides a number of built-in types:

- bool, representing booleans.
- char, representing 16-bit Unicode characters. A character constant is written in single quotes, as in 'A'.
- string, representing sequences of zero or more Unicode characters. A string literal constant is written in double quotes, as in "ABC".
- Integer types: sbyte (signed 8-bit), byte (unsigned 8-bit), short (signed 16-bit), ushort (unsigned 16-bit), int (signed 32-bit), uint (unsigned 32-bit), long (signed 64-bit), ulong (unsigned 64-bit).
- Real (floating-point) types: float, double and decimal, representing 32-bit, 64-bit and 128-bit IEEE floating point numbers.

The type object is ancestor to all class types (think *ANY* in Eiffel).

← *"Overall inheritance structure", 16.10, page 586.*

The void (null reference) is written null.

References and values

Every C# type is either a reference type or a value type; this is the same distinction as between reference and expanded types in Eiffel. A variable of a value type directly denotes a value, which may be a simple value as just seen (the built-in types except string and object are value types) or a complex object; a variable of a reference type denotes a reference to an object.

You can go from a value to a reference through an operation called *boxing*:

```
int i; object o;
i = 1;
o = i ;  // Boxing: create an object containing the integer 1, and attach o to it.
```

As this example indicates, boxing happens automatically on assignment of a value to a reference. Unboxing is explicit, using the "cast" operator (…):

```
i = (int) o;   // Unboxing: get the integer value in o, and assign it to i.
```

Constants

A field can be declared const to indicate that it has the same value, specified in the declaration, for all instances of the class. The value can be a literal constant (a manifest integer, string etc.) or an expression that only involves previously defined constants:

```
public const string s3 = "ABC- ";
public const string s4 = s3 + "DEF";      // Value: "ABC-DEF"
```

Since a constant's value is independent of any target object, you can access it using the class name, as in A.s4 if the above declarations appear in a class A.

Note the difference between const and readonly fields: a read-only field is, like other non-constant fields, attached to an object, and its value does not necessarily appear in the program (it may be set by constructors).

Methods

Routines are called methods in C#. Methods cover both procedures (with no result) and functions; a result-less method uses void as its return type. Here are a few example declarations illustrating some important possibilities:

```
class B {
    public void p(int arg1, ref int arg2) {… arg2 = 0;}       // Procedure
    public string f() {… return "ABC";}                      // Function
    public static string sf() {… return "DEF";}              // Static function
}
```

By default arguments are passed "by value" (as in Eiffel), meaning that the formal argument represents a copy of the actual argument; what is copied may be a reference or an object depending on the type. But you may also declare a formal argument ref ; in that case assignments to the argument, such as the assignment to arg2 in p above, will also modify the corresponding actual.

The actual argument corresponding to a ref formal argument must also be specified as ref in the call, as in

```
B v = new B();
int x = 1;
int y = 1;

v.p(x, ref y);        // Does not change x, sets y to 0
```

Local variables are not declared in a separate clause; the declaration of a local simply appears in the body of a method, prior to its first use. The name must be different from the names of formal arguments and other locals.

Local variables and formal arguments may have the same name as members of the class, over which they take precedence. This explains why you will encounter the pattern

```
int a;                          // a is a field
r (int a) { this .a = a;}       // a is also a formal argument to r
```

where a, within the same class, denotes both a field and a method argument; the method can still access the field through the notation shown, this .a. This is actually a common occurrence, especially for constructors (reusing the name of a field for the constructor argument that serves to initialize it). It is better to stay away from it and choose different names for each purpose.

Overloading

C# permits method overloading: two methods of a class may have the same name as long as their argument signatures (number and types of arguments, including whether they are ref) are different. The result type plays no role.

> The previous methodological comment applies here too: names are not a scarce resource. Overloading, however, is a pervasive practice in languages such as C# and is required in the case of multiple constructors as discussed below.

Properties

The export policy, as noted, does not distinguish between read and write access. This means that it is almost never appropriate to export a field a, since this would allow clients to perform direct field assignments x.a = v in direct violation of information hiding principles. The object-oriented solution in this case is to provide a getter function (not necessary in Eiffel since you can export the attribute, giving read access only) and a setter procedure. C# standardizes the writing of setters and getters through the notion of property. The pattern for shadowing a secret attribute a with a property is:

```
class C {
    private string a;               // The secret attribute

    public string ap {              // The property
        get {return a;}             // The getter
        set {                       // The setter
            a = value;              // The field's value
            … Possibly other instructions …
}   }   }
```

This mechanism uses the three keywords get, set and value. It declares two special methods with names get and set, enabling the getter and the setter to access the attribute with, in the setter case, assignment syntax:

```
C x = new C();
string b;
b = x.ap;             // Uses the getter
x.ap = "ABC";         // Uses the setter
```

The effect is similar to what is achieved in Eiffel with an **assign** specification; the Eiffel mechanism does not require writing the getter, which in practice is almost always of the above form return a. (The setter, for its part, commonly includes more than an assignment, for example a log update, so it is normal to request writing it explicitly.)

← *“Bracket notation and assigner commands”, page 384.*

Constructors

Creation procedures, used to initialize objects, are called constructors in C#. A constructor can be:

- An instance constructor, initializing objects created dynamically.
- A static constructor, initializing static data members.

The following class contains an example of each kind:

```
class D {
    public D (string a) {                      // Constructor 1: instance
        … Field initializations, typically using a …
    }

    static D () {                              // Constructor 2: static
        … Field initializations …
    }
}
```

Constructors do not have their own names but use the name of the class, here D, relying on overloading, and disambiguation through the signatures, if there is more than one constructor.

This convention does not work in some cases, such as a class POINT that would require two creation procedures make_cartesian and make_polar: both would have the same signature, two arguments of type float.

A constructor declaration does not specify a return type, not even void. In the case of a constructor marked static, there may not be any other modifier.

The creation of a new object relies on the new operator (think **create** in Eiffel) and an instance constructor, as in:

```
D x = new D("ABC");
```

C# does not enforce the distinction between declarations (static) and instructions (dynamic); *statement* is the term that covers both of these concepts, as well as their combination as in this example which both declares and initializes x. Execution of the new expression will create a new instance of D and call the associated constructor, here the one that was marked Constructor 1.

This is not, however, the full story on instance constructors and instance creation. The detailed specification (given below in the discussion of inheritance) implies that the constructor you call may itself call other constructors, resulting in a constructor chain that *must* involve constructors from every ancestor class.

→ *“Inheritance and creation”, page 798.*

A static constructor — rather, *the* static constructor of a class, since there can be at most one, with no argument as illustrated above — gets executed before the first occurrence, if any, of either a creation of an instance of the class or an access to one of its static members. This facility makes it possible to initialize properties associated with a class rather than specific instances (as you might do in Eiffel through a once function). Imagine for example an error reporting system, where errors are recorded into a special log file. The first creation of an error object will create and open that file.

Destructors

C# is designed for garbage collection: while object creation happens explicitly through new, reclamation of unused objects’ memory space is the responsibility of an automatic mechanism, the garbage collector (GC).

Sometimes you may want to ask the GC to perform a specific operation whenever it reclaims an object. The typical example is a file object, associated with an actual file: whenever the object gets collected, you will also want to close the associated file. (In Eiffel you can, for this purpose, define a *dispose* routine which the GC will call on reclaiming the object.)

C# destructors fulfill this need. A destructor is a method of name ~C where C is the name of the class. No overloading here; a class may have only one destructor, without arguments, return value, or modifiers (such as static):

```
class File {
    … Other class members including constructors …
    ~File() {
        … Instructions to close the associated physical file…
    }
}
```

Operators

An operator is a static method with an operator name, as in

```
class E {
    public static E operator +(E a, E b) {
        … Computation of result into exp…
        return exp;
    }
}
```

which can then be called (like an Eiffel feature with an operator alias) in operator syntax, as in x + y with x and y of type E in this example. Operators are built-in for predefined types such as int and float; you can use the same operator names (unlike in Eiffel, you cannot define your own) to define operators for your own classes such as E above. The main available unary operators are

← *"The ubiquity of calls: operator aliases", page 134.*

```
+     –     !     ~     ++    ––
```

where ! is negation for booleans and ~ is negation for integers (replacing every 0 by a 1 and conversely in the binary representation). ++ and –– are side-effect-producing operations: x++ returns the value of x, then increments x by one; ++x also increments x, but returns the incremented value; and similarly for x–– and ––x. As you know, expressions with side effects are not a good idea; use them at your own risk. Binary operators are:

```
+     –     *     /     %          // Arithmetic (% is remainder)
<     >     <=    >=               // Relational (should yield boolean result)
==    !=                           // Also relational (equality, inequality)
&     ^     |                      // Boolean, strict (and, xor, or)
<<    >>                           // Bit-shift (left, right)
&&    ||                           // Boolean, semistrict (and, or)
```

← *"Semistrict boolean operators", 5.3, page 89.*

As in other C-based languages, = is reserved for assignment, requiring a special notation == for equality. Some of the above operators handle integers in their binary representation: the strict boolean operators are applicable not only to booleans but to integer operands, to which they apply the corresponding operations to every bit in the representation; the left and right bit-shift operators shift the bits of the first operand (m in m << n) by the number of positions given by the second operand (n), discarding the bits that fall off the right or left edge, and filling the freed positions with zeros.

You can use any of the operators listed, except semistrict boolean operators, for your own types, by relying on the overloading mechanism. In the case of comparison operators, overloading must come in pairs: if you overload < you must overload >; similarly for <= and >=, as well as == and !=.

In addition, C# supports the following non-overloadable operators:

```
[…]                                   // Array indexing
(…)                                   // Cast
+=    –=    *=    /=    %=    ^=      // Operation cum assignment
&=    |=    <<=   >>=                 // Operation cum assignment
```

+= and such combine an arithmetic or other operation with an assignment: x += 1 is a shortcut for x = x + 1.

Arrays and indexers

To declare an array, with one or more dimensions, use bracket notation:

```
string[] a;                 // One-dimensional array
string[ , ] b;              // Two-dimensional arrays
```

For more than one dimension, use commas as illustrated. Arrays are indexed from 0: b [1, 1] is the element in the second row and second column. Because of this condition you must take care to avoid indexing errors.

The declaration does not specify bounds; the array will be allocated dynamically as in Eiffel. The assignment

```
a = new string[4];
```

creates an array with four elements, initialized to standard default values. Alternatively, you can directly specify the values of these elements:

```
a = new string [] {"A", "B", "C", "D"};// Creates an array with four elements
```

For multiple dimensions, use commas to separate the brace-enclosed lists:

```
b = new string[2, 3];
b = new string[,] {{"A", "B"}, {"C", "D"}, {"E", "F"}};   // Dimension: [3, 2]
```

In addition to these standard rectangular arrays, C# offers *jagged* arrays, which are simply arrays of arrays, as in

```
string[][] c;
```

Each row may have a different size (hence “jagged”). A typical initialization is

```
c = new string[][] {  new string[] {"A"},
                      new string[] {"B", "C", "D"},
                      new string[] {"E", "F"}};
```

resulting in rows with one, three and two elements.

Access and modification use bracket syntax, as in

```
b[0, 0] = "Z";
c [0] = new string[] {"Y", "Z"};      // Changes the first (zero-th) row
c [0][0] = "Z";
```

To allow bracket-like notation for accessing structures other than arrays, as in

```
Table t = new Table (); string n;
…
n = t [1, 1];       // Accesses the very first element (see implementation below)
```

where Table is one of your own classes, you should equip that class with an *indexer*, performing a role similar to Eiffel’s bracket alias. The definition of an indexer resembles the definition of a property:

← *“Bracket notation and assigner commands”, page 384.*

```
class Table {
    private string[,] rep;          // Initialization of rep omitted
    public string this[int i, int j] {
        get {return rep [i – 1, j – 1];}
        set {rep [i – 1, j – 1] = value;}
    }
}
```

The indexer has the name this, defined with a getter and a setter. In this example the implementation of Table provides indexing from one in both dimensions, relying internally on a private array rep indexed from zero.

Genericity

C# genericity concepts (the more common term for Java and C# is "generics") will be familiar from those of Eiffel; generic parameters are enclosed in angle (rather than square) brackets <…>. If you declare

```
class F<G, H> where H: T, new() {
    …Class declaration …
}
```

class F has two generic parameters. The second one, H, is constrained by T, meaning (as in **class** *C* [*G*, *H* –> *T*] in Eiffel) that any actual generic parameter must conform to *T*. The inclusion of new() means (as in Eiffel if the declaration read *C* [*G*, *H* –> *T* **create** *make* **end**] for a creation procedure *make* of *T*) that *T* must provide a public constructor without arguments; this allows methods of F to create instances of T.

A generic derivation (or "instantiation") also uses angle brackets:

```
F<V, W>            // W must conform to T, and provide a default constructor
                   // with no arguments.
```

Basic statements

Here is a review of the principal kinds of instructions (executable statements).

Assignment, as we have seen, uses an equal sign, as in

```
var = e ;
```

Note the use of the semicolon as a *terminator* (rather than a separator), required at the end of every basic statement.

A **method call** uses the name of the method and a list of arguments. Unlike routine calls in Eiffel, it always includes a parenthesis pair even if the argument list is empty, as in methodWithNoArgument ().

The **return** instruction has no direct equivalent in Eiffel. It is written just return in a procedure (method with void type) and, in a function (method with an actual result type), specifies a value to be returned:

```
return some_expression;
```

A return terminates the current method execution and, in the second case, delivers the value of the expression as the function's result. This means that C# does not enforce the one-exit rule.

← *"Structured programming", page 188.*

Control structures

A **conditional** has the following structure:.

```
if (c1) {
    …
} else if (c2) {
    …
} else {
    …
}
```

with zero or more else if parts, and zero or one else part. Conditions must appear in parentheses. There is no elseif keyword; technically the use of else if (two keywords) leads to nesting the else parts, but this is not visible because of the absence of an **end** keyword, and is visually supported by the indentation style.

← *"Comb-like structure, figure on page 179.*

The **multi-branch** instruction has the form

```
switch (expression) {
    case value: statement; break;
    case value: statement; break;
    …
    default: statement; break;
}
```

← *"Multi-branch", page 195.*

where *expression* is an integer or boolean expression and each *value* is a compile-time constant. If the value of the *expression* does not match any of these constants, the instruction executes its default branch if present, nothing otherwise. (In Eiffel, in the absence of an **else** clause in an **inspect**, this case produces a run-time error.) The switch instruction by itself is not a one-entry, one-exit conditional but a multi-target goto; to obtain the effect of a multi-branch conditional you must include break; instructions as shown. Explicit transfer is mandatory at the end of every case, to avoid the common bug, in the C++ and C version of switch, of mistakenly continuing from one branch to the next.

← *For methodological comments on the use of such constructs see "The goto puts on a mask", page 189.*

Several forms of **loop** are available. The most general one, coming from C and easy to relate to the Eiffel **from** structure, is.

```
for (initialization; exit; modification) {
    … body …
}
```

This executes the initialization and stops if *exit* is true, otherwise executes the *body* and the *modification* until *exit* becomes true. The *modification* typically includes the part of the iteration that advances to the next step, such as incrementing an index or advancing a cursor. (In Eiffel it would be integrated with the *body*.)

For simple while- or until-style loops you may use one of

```
while (condition) {statements}
do {statements} while (condition)
```

Surprisingly, C# also offers a **goto** statement: goto *l*, where *l* is a label; you can label a statement by prefixing it with *l*: (a label followed by a colon).

Exception handling

An exception is an abnormal run-time event that interrupts the normal flow of control. Typical causes of exceptions are null pointer dereferencing (void call: x.f where x is null) and arithmetic overflow. It is also possible in C# to trigger a *developer exception* explicitly through

← *"An introduction to exception handling", 7.10, page 200.*

```
throw e;
```

where e is of an exception type, which should be a descendant of the Exception library class.

A C# program may *handle* an exception in the following style:

```
try {
    … Normal instructions, during which an exception may occur …
} catch (ET1 e) {
    … Handle exceptions of type ET1, details in e …
} catch (ET2 e) {
    … Handle exceptions of type ET2, details in e …
}… Possibly more cases…
finally {
    … Processing common to all cases, exception or not…
}
```

If execution of the try block triggers an exception of one of the types listed, here ET1, ET2, … execution immediately transfers (without completing the try block) to the corresponding catch block. The finally part is executed in all cases, exception or not; its typical purpose is to release resources, for example to close open files, before moving on.

Occurrence of an exception creates an exception object, accessible to the program in the corresponding catch clause through the specified exception name, such as e. This makes it possible to access such properties as the human-readable name of the exception and the state of the call stack, although ordinary exception handling seldom uses this facility.

If an exception occurs whose type does not match any of the listed types — or it is triggered outside of a try block —, it is passed on to the caller for application of this policy, until a caller up the chain has a suitable catch; if none does, the program terminates in error.

The reader familiar with Java will have noted that the preceding description applies to both languages (which both got the mechanism from C++). C#, however, departs from the Java model by ignoring the rule that a routine should specify the user-defined exceptions (those coming from an explicit throw instruction) that it could trigger. As a consequence there is no equivalent to Java's throws specification. This is a source of controversy in both language communities.

Delegates and events

C# offers the *delegate* mechanism to describe high-level routine objects, as addressed by Eiffel's agents. An associated mechanism, *events*, complements delegates for event-driven programming. (In Eiffel event types were described as ordinary objects and hence did not need any specific language construct.)

← See chapter 17 about agents and chapter 18 about event-driven design.

The basic way to use delegates, which will encapsulate methods, is to declare the corresponding delegate type, as in

```
public delegate int DT (string s);                                [4]
```

This is a type declaration; it defines a type DT representing functions of one string argument returning an integer result. To define an actual delegate and associate it with a given method — say int lettercount (string s), a function in the current class returning the number of letter characters in a string — you may use

```
DT d = new DT (lettercount);                                      [5]
```

although you can make the constructor call implicit by using the method name directly:

```
DT d = lettercount;                                               [6]
```

In general, modern programming languages (other than functional languages) do not allow using a routine as argument to another routine; the agent and delegate mechanisms, as well as the function pointers of C++ and C, are precisely devised to avoid this need by providing special objects, usable as arguments, which serve as wrappers around functions. To produce a C# delegate from a method, however, you must pass the method itself as argument to the new constructor of delegates as in [5]. Conceptually this is the only case in which the language allows treating a method name as a value. In practice C# relaxes the rule by allowing assignments such as [6], or passing the method name as argument to another method, but these are just syntactical shortcuts; the value actually passed is a delegate, here new DT (lettercount).

You can call a delegate like any other method: the call

```
n = d ("A");                                                     [7]
```

has the same effect, after the above assignment [5] or [6], as n = lettercount ("A"); — a direct call to the original function. Of course, in a call such as [7] we usually do not know which particular function d represents; this is particularly the case if we used [7] in a method r, of which d is the formal argument, and instead of an assignment such as [6] we pass the delegate as actual argument:

```
r ( lettercount );                                               [8]
```

This is, as noted, an abbreviation of r (new DT (lettercount)).

The Eiffel equivalent of the combination of [4] and [5] would be the single instruction *d* := **agent** *lettercount*, with no need for a type declaration such as [4]. This form has no direct equivalent in C#, but you can use delegates on "anonymous methods" (the equivalent of inline agents), as in

← *"Inline agents", 17.7, page 652.*

```
r ( delegate (string s) {return lettercount (s);} );
```

where the highlighted part is the anonymous method. Note that such an anonymous method declares the argument signature but not the result type. In a similar vein, there is no direct equivalent to the notion of open argument (the **?** syntax in Eiffel), but you can achieve it through anonymous methods in the illustrated style.

An alternative notation for the equivalent of inline agent is the **lambda expression**, as in (int x, int y) => x + y, corresponding in mathematical lambda notation to $\lambda\, x, y : INTEGER \mid x + y$.

← *"Lambda calculus", 17.6, page 640.*

A C# delegate is not constrained to represent a single method; if a and b are declared of the same delegate type, a + b denotes another delegate of that type; executing that delegate means executing the methods associated with a and b, in order. You can directly add and remove components to such a *multicast* delegate through the operators += and −=.

For event-driven programming you may define event types and associate them with delegates. In fact an event type is always associated with a delegate type, as in the declaration:

```
public event DT1 click;
```

defining click as an event type such that the associated events will be treated by delegates of type DT1.

You must read such a declaration to the end: public event does not qualify DT1 (which must be defined separately as a delegate type); instead, this is a declaration of the last item listed, click, with three qualifiers — public, event and DT1.

This example uses a new delegate type DT1 rather than the above DT since delegate types handling events usually return void whereas DT was a function delegate. If we intend click to represent mouse-click events with two associated integer coordinates, DT1 will be declared as

```
public delegate void DT1 (int x, int y);
```

To understand the mechanism it is necessary to know that the C# implementation represents an event type member, such as click, as a list of delegates corresponding to the various methods subscribed to the event. This explains how to subscribe to an event:

```
click += r;
```

where r is a routine with the appropriate signature: void r (int x, int y). The += operator, overloaded for lists, adds an element to a list. Note that you can use the routine r directly rather than explicitly wrapping it into a delegate; what gets added to the list, however, is a delegate. To remove a subscriber, use −=.

This takes care of subscription. To publish an event, use the scheme

```
if (click != null) {click (h, v);}
```

(In this example, h and v are two integers giving the mouse coordinates.) Once again you need to know about the list implementation to understand the need for the test click != null: if no delegate has been subscribed to click, the list will be null and the call click (h, v) would cause a run-time error.

The recommended style for handling event arguments in the .NET framework does not directly use individual arguments such as x and y in this example, but an event type class declared as a descendant of the library class EventArgs. In this example you may define a class IntPairArgs inheriting from EventArgs and declaring two integer fields x and y; then the routines you subscribe will be of the form

```
private void rHandler (object sender, IntPairArgs e) {r (e.x, e.y);}
```

where — as part of the same recommended style — the first argument sender represents the target object; the second argument e represents the event arguments. The advantage is that all event handling schemes look the same; it is not clear, however, that it justifies the resulting complication: instead of directly reusing existing model routines such as r you have to wrap them into special glue code such as rHandler.

B.4 INHERITANCE

The C# model of inheritance, essentially taken from Java, does not support multiple inheritance from classes but only from a special kind of abstraction called the *interface*.

Inheriting from a class

Here is how a class declares another as its parent:

```
class L : K
{… Declarations of members of L …}
```

This makes L an heir of K. The following example shows the order of specification elements if the inheriting class is also generic, here with a constraint:

```
class M<G> : K where G: T
{… Declarations of members of M …}
```

You may only specify one parent class, here K.

Abstract members and classes

Corresponding to deferred features and classes in Eiffel (without the contracts), you may define both members of a class and the class itself to be abstract:

```
abstract class N {
     public abstract void r();      // Note no implementation
     public abstract int s();       // Note no implementation{…}
… Other members (of which some may be abstract and some not)…
}
```

You may not instantiate an abstract class (as in new N (…)). An abstract method may only appear in an abstract class; non-abstract descendants must provide an overriding (redefinition) that is not abstract. You may not declare a method or class both abstract and static.

Properties and indexers may also be abstract, subject to the same rules.

Interfaces

An interface is similar to a class that would be *completely* abstract. Interfaces specify abstract functionality that each descendant will implement in its own way. Interfaces provide a restricted form of multiple inheritance since a class may, as noted, inherit from any number of interfaces. This enables the language to avoid providing a mechanism for resolving clashes between inherited features (we saw that this problem has a simple solution through renaming), at the price of renouncing some of the power of inheritance.

← See the discussion in "Deferred classes and features", 16.5, page 565.

← "Renaming features", page 590.

A typical interface would read as follows:

```
interface IOrdered <T> {
    bool lesser (T other);
    bool greater (T other);
    bool lesserEqual (T other);
    bool greaterEqual (T other);
}
```

The recommended style convention gives interfaces names starting with I. The example illustrates that interfaces can be generic, and also that their members are not declared public or private: they are implicitly public. This interface defines the notion of objects comparable through the usual comparison operations.

It also illustrates the limits of the notion of interface: because members cannot have any implementation, you cannot specify that a.greater (b), for example, should always be implemented as b.lesser (a). You could do this in an abstract class, but then a concrete (non-abstract) class that inherits from it could not have any other parent, even though comparability generally is only one among several properties applicable to a class.

A class can inherit from one or more interfaces, providing implementations (effectings in our earlier terminology) of its members:

```
class TennisPlayer: IOrdered<TennisPlayer>, IAnotherInterface {
    public int ranking;
    public bool lesser (TennisPlayer other) {return (ranking < other.ranking);}
    … Similarly for greater, lesserEqual, greaterEqual …
    … Implementations of members of IAnotherInterface …
    … Other members of TennisPlayer …
}
```

Name clashes do not prevent such multiple interface inheritance; there is no renaming mechanism as in Eiffel, but the inheriting class may disambiguate clashing names through the dot notation, as in IAnotherInterface.clashingname.

Accessibility and inheritance

Inheritance brings in new accessibility (export) modifiers for class members in addition to the ones seen earlier (public, internal, private):

← *"Export status", page 779.*

- protected: accessible in the class and its descendants.
- protected internal: accessible in classes of the same assembly and in descendants (combines protected and internal).

When you override an inherited member as discussed next, you cannot change its access status. This is different from the Java policy which allows extending (but not restricting) accessibility in descendants. In addition, descendants may not have greater accessibility than their ancestors.

Overriding and dynamic binding

You may generally override (redefine) an inherited member. The conventions, however, are different from those of other modern object-oriented languages. As in C++ and unlike in Eiffel and Java, binding in C# is — surprisingly for a language first released in 1999 — static. In other words, the version of f to be executed in a.f follows from the declaration of a, not from the dynamic type of the object attached to a. To obtain dynamic binding you must declare the member virtual:

← *"Redefinition", 16.6, page 570.*

```
class P {
    public virtual void f (…) {…}
    …
}
```

and override it in descendants by keeping the same argument signature (exactly) and replacing virtual by the override modifier:

```
class Q: P {
    public override void f (…) {…}
    …
}
```

You may access the original version of an overridden feature — as with **Precursor** in Eiffel — through the keyword base and dot notation; for example, the implementation of f in Q can use

←*Page 573.*

```
base.f (n);
```

to take advantage of the original P implementation.

The new declaration will then be considered an *overriding* of the original, and cause dynamic binding. These rules require that you exert particular care since a new declaration that does not exactly follow the above scheme (original method specified virtual, new one specified override) will sail through the compiler but have a different result. Consider the following variant:

```
class R {public void f (int i) {…}}
class S: R {public void f (int i) {…}}
R r1  ;   S s1 = new S()  ;   int n;
r1 = s1;
```

The original f is not declared virtual in R, but you can still use a new declaration in the descendant S. It is not an override but simply declares a new method that shadows the inherited one on targets of (statically declared) type S, so that s1.f(n) will use this version; but r1.f(n) will use the R version regardless of the run-time value of r1. This is rather dangerous. To protect yourself against the risk of error, you are expected in this case to mark the overriding member as new:

```
class S: R {public new void f (int i) {…}
```

but this is not required; forgetting new only causes a compiler warning. In the same way, no dynamic binding will occur for p1.f (n), with p1 declared of type P above and dynamically of type Q, if you omit the override modifier in Q; this is not an error, simply a case that applies static binding.

An overriding declaration can use the modifier sealed to prohibit any further overriding in descendants:

In Eiffel you may similarly declare a feature or class **frozen**.

```
class Q1: P {
    public sealed override void f (…) {…}
    …
}
```

This is only permitted in overriding redeclarations (marked override); the reason is that you do not need it for original declarations since any new member not marked virtual is non-overridable. You can also declare a class sealed (sealed class T …) to prohibit its use as parent of others.

Declaring classes and members sealed is a more common practice in .NET libraries than object-oriented methodology would normally command, probably because in the absence of contracts this is the only way to prevent descendants from messing up the original intent.

The language's choice of static binding by default is clearly wrong. The consequence of this observation is a methodological rule for C# programming: **always declare methods virtual by default**; only remove this modifier in the rare case of a method that you want to seal.

Inheritance and creation

C# enforces a special rule on constructors in the presence of inheritance: any constructor of a derived (heir) class must call a constructor of the parent. As a result, a constructor call will always result in a chain of constructor calls, going all the way to object's default constructor. This effect can be achieved either explicitly or implicitly:

- The constructor can call a parent constructor of its choice through the base notation, as in base (n); since constructors do not have individual names (only the name of the class), the argument signature determines, after resolution by overloading, which parent constructor is intended.
- In the absence of such a call, the parent must have a default constructor, which will automatically execute before the heir's constructor starts its execution.

One of these cases must apply: the absence of both a default constructor in the parent and an explicit parent constructor call in the heir's constructor is a validity (compile-time) error.

As noted in the discussion of the corresponding Java mechanism, the reason for these rules is not entirely clear. The intent is probably to make sure that an instance of a descendant type also satisfies the consistency constraints defined by proper ancestors. The constructor chain appears as an attempt to ensure such consistency in the absence of a notion of class invariant to express the constraints explicitly.

← "Run-time model, object creation and initialization", page 755.

Run-Time Type Identification

← You should be familiar with "Uncovering the actual type", page 599, which describes the issue and introduces object test.

To force a type U on an expression exp of type T (as with Eiffel's object test) you may use two mechanisms:

- An explicit cast, letting you write (U) exp, an expression of type U regardless of the declared type of exp. If the value of exp is indeed an object of type U at execution time, you may use this expression to refer to that value under that type. The downside is that if the dynamic type does *not* match U an attempt to evaluate the expression will cause an exception, which you should handle for safe programming.
- As a more gentle technique, you can use the boolean expression exp is U.

In the second case exp is still statically of its declared type T; but then you may combine the two mechanisms with the guarantee that the cast will work:

```
if (exp is U) {r ( (U) exp)};     // With the method declaration r ( U  x) {…}
```

B.5 FURTHER PROGRAM STRUCTURING MECHANISMS

C# introduces a number of program structuring facilities beyond the basic object-oriented paradigm.

Namespaces

In combining software from different sources you may run into type name clashes: two providers give you types, usually classes, with the same name. C# makes it possible to resolve such clashes by defining an extra level of naming for types, the name space — usually written in one word: namespace.

By default all type names belong to a global namespace. You can define your own namespaces and include type declarations in them:.

```
namespace N1 {
… Declarations of classes (and other types if desired) …
}
```

Then a client needing to access several classes called C can disambiguate its intent through the notation N1.C.

An important predefined namespace is System, which contains fundamental library classes.

You may nest namespaces; this can mean nesting the texts physically, but you may also use dot notation to define the innermost namespace separately:

```
namespace N1.N2 {// Sub-namespace of N1; contents accessible as e.g. N1.N2.V
… Declarations of classes including V …
}
```

A client that uses many names from a namespace can avoid repeating the qualified notation (as in N1.N2.C, N1.N2.D and so on) through a using directive:

```
using N1.N2;
using SomeOtherNamespace;
… Here we can use V as a shortcut for N1.N2.V …
```

As this example suggests you can have any number of using directives. If the corresponding namespaces define types with clashing names, you must go back to dot notation to disambiguate them: here if SomeOtherNameSpace defines a type called V the above is no longer valid and you must use N1.N2.V.

Extension methods

Assume you want to extend the concept covered by an existing class X. The normal object-oriented mechanism is to declare a new class that inherits from X. But inheritance may be inconvenient or inapplicable. For example X might be sealed (a common practice, as noted, in .NET libraries). More generally, defining a new class means defining a new type; even if you are only adding methods and no fields, the type system prevents you from applying its operations to existing objects of the original type X, for example objects that have been stored in a file or database.

To address this issue, C# provides the interesting mechanism of *extension methods*: methods added, from the outside, to an existing class.

Extension methods are essentially a syntactic simplification, since one could solve the extension problem through static methods: in any class you may write a static method sm to be called as sm (x1, other_args) with x1 of type X. What we want, however, is a method m that will be called in the same style as if it were a method of X:

```
x1.m (other_args);                                                    [9]
```

even though m is declared somewhere else. The syntactic trick is to mark the first argument of m with the modifier this:

```
public static class Y {
    static void m (this X x, int arg1, int arg2) {…}
    …
}
```

making [9] valid (with two integer values to replace other_args in this case).

Attributes

By nature a programming language is restricted to the semantic mechanisms foreseen by its creators. Sometimes new applications call for the inclusion of supplementary properties to characterize program elements; such properties will not necessarily be relevant for the compilation process and obviously should not change the existing semantics — if it did, the proper solution would be to change the programming language and update its compilers — but may be of interest to other tools, for example documentation or serialization tools.

← *To* serialize *an object structure is to map it to external storage. See "Uncovering the actual type", page 599.*

It is prudent for a programming language to provide an open-ended mechanism supporting such extensions, known as **metadata** (supplementary information added to a document, separate from its essential content). In Eiffel you may have noticed the **note** clauses associated with classes and features, which serve precisely that purpose. .NET and C# provide metadata support in the form of **attributes** (not to be confused with the object-oriented notion of attribute as used elsewhere in this book, for which the C# term is "field").

Some attributes are predefined, but programmers may also define their own, known as *custom* attributes.

An example of predefined attribute is Serializable, which you can attach to a class declaration to indicate that its instances can be converted to an external representation suitable for external storage:

```
[Serializable]
public class Z {… Class declaration as usual …}
```

The syntax for adding an attribute to a program element — here an entire class, but the convention is the same if you want to attach the attribute to a class member — is simply to prefix its declaration with the attribute name in brackets.

A custom attribute is defined by a class, which must be a descendant of the library class System.Attribute (that is, class Attribute in the predefined namespace System). As an example, assume we want the possibility of equipping classes with basic version control information: an author name, a modification date, and optionally a revision number, all strings. We use:

```
public class ChangeAttribute: System.Attribute {
    private string author;
    private string last;
    public ChangeAttribute (string a, string l)
        {author = a ; last = l;}
    public string revision;
}
```

Note the recommended style of giving custom attribute classes a name that ends with Attribute.

Then we can equip a class or (here) a method with version information:

```
[ChangeAttribute ("Caroline", "24 December 2009")] public void r {…}
```

In using the attribute we must pass arguments for the chosen constructor. It is also possible (but optional) to specify values for public, writable fields of the custom attribute class, such as revision in this example:

```
[ChangeAttribute ("Caroline", "24 December 2009", revision = "2.1" )]
public void r {…}
```

ChangeAttribute as declared is applicable to any program element: class, struct, method, field, delegate and several others. You can restrict the applicability of an attribute by attaching to its own declaration the AttributeUsage attribute:

```
[System.AttributeUsage(System.AttributeTargets. Class )]
public class ChangeAttribute: System.Attribute {… The rest as before …]
```

Instead of Class you may use such other possibilities as All (the default), Assembly, Delegate, Event, Interface, Field, Method, Parameter (argument), Struct. You may specify two or more permitted target types, separating them by |.

Within an attributed element you can obtain the values of the attributes through built-in "reflection" mechanisms. For o denoting an object, the call

```
o.GetType().GetCustomAttributes(true);
```

returns array containing the attributes that have been defined for the generating class of the object, with their values.

B.6 ABSENT ELEMENTS

C# misses a number of object-oriented mechanisms to which we have grown accustomed in this book, notably contracts and multiple inheritance from classes.

The Spec# experimental language, from Microsoft Research, extends C# with contracts. It is based on an earlier version of C#, version 2.0.

research.microsoft.com/en-us/projects/specsharp/.

B.7 SPECIFIC LANGUAGE FEATURES

It is useful to be aware of a few supplementary C# constructs.

Unsafe code

C# combines a strong emphasis on type safety with a desire to let programmers work on low-level aspects that would normally be handled in C or assembly language. The “unsafe” construct supports a clear separation between normal, type-checked parts of the program and data, and the unsafe elements.

Declaring a method unsafe enables it to manipulate a data area outside of the C# heap (the set of objects created by new operations), performing direct pointer manipulations similar to those of C++ and C and hence bypassing the normal type rules on that part of the data.

Enumeration types

Enumerated types make it possible to define variables that range over a finite set of predefined values, as in

```
enum CardColors {Spades, Hearts, Diamonds, Clubs}
```

The values are actually integers; the underlying type is int by default, but you can specify another integer type, as in enum T:long {…}. Values normally start at zero, but you can specify explicit values, as in

```
enum CardColors1 {Spades = 1, Hearts, Diamonds, Clubs}
```

where values not specified are incremented by one from those preceding. In all cases the resulting values must all be different. You can denote the values by prefixing them with the type name and a dot, as in CardColors.Spades. You can treat the values as integers through explicit casts.

Linq

A set of mechanisms introduced in C# 3.0 has attracted considerable attention. Known as Linq, it enables the programming language to cover database and web manipulations that are generally handled through separate formalisms, typically SQL for databases and XML for web aspects. Object-oriented environments usually support such manipulations through libraries that provide the interface to the underlying mechanisms, such as relational databases and the HTTP protocol. The originality of Linq is to make everything expressible without leaving the programming language at all; for example you can run a database query as

SQL (Structured Query Language) is the standard language for manipulating relational databases.

```
from e in Employees where e.salary > median select e.rank
```

referring to a relation Employees from a database; it produces the list of ranks of all employees making more than a certain salary. This is the same style as SQL queries, but integrated in the programming language; the elements involved are normal objects, such as lists, of the program.

B.8 LEXICAL ASPECTS

C# identifiers follow conventions similar to those of Eiffel. They may, however, start with an underscore (not just a letter), although this possibility is not normally used in application programs. An important difference is that identifiers are case-sensitive: anIdentifier, AnIdentifier and anidentifier are all considered different. The underlying character set is Unicode.

Comments in C# programs can be:

- Single-line: any part of a line starting with //.
- Multi-line: starting with /∗ and ending with ∗/.

Single-line comments starting with /// (with an extra slash character) are intended to contain XML code for use as special documentation by automatic documentation processing tools.

B.9 BIBLIOGRAPHY

Judith Bishop and Nigel Horspool, *C# Concisely*, Addison-Wesley, 2003.

An introduction to the basic C# mechanisms (covering an earlier version of the language).

Online documentation at msdn.microsoft.com/en-us/vcsharp/default.aspx.

The best place to get detailed specifications of language mechanisms. The site also includes a number of tutorials.

C

An introduction to C++ (from material by Nadia Polikarpova)

It is customary to characterize object-oriented languages as either "pure" (fully and solely implementing O-O concepts) and "hybrid" (mixing O-O and non-O-O features). The most prominent representative of the *hybrid* class is C++, designed to make some O-O ideas acceptable to C programmers. C++ has a general flavor quite different from the Eiffel notation used in the rest of this book; it is also more complex.

You will find many books and introductory articles about C++, usually starting from scratch or assuming knowledge of C. The role of this appendix is different: it is directly targeted to you, the patient reader of this book, who have made it to page 805 and hence master object-oriented programming in its "pure" form . The goal is to enable you to apply what you have learned if you have to program in C++. For that reason, many C++ constructs will be explained — as was done for Java and C# — in the style "*This is what you would do in Eiffel, and here's how to obtain the same or a similar effect in C++*".

C, on which C++ is based, is itself an important language, briefly covered in the next appendix.

C.1 LANGUAGE BACKGROUND AND STYLE

Today the idea of using an object-oriented language hardly scares anyone, but in the late eighties it was sulfurous; many programmers and managers in industry and academia, while intuitively attracted by O-O concepts, were suspicious of their applicability. From 1979 on Bjarne Stroustrup, then of Bell Laboratories, designed and implemented a language initially called "C with classes", which extended C with concepts imitated from Simula 67, the first object-oriented language; code was translated into plain C through a "preprocessor". The language soon proved a hit, as it promised a reassuring transition path for C programmers.

Stroustrup (2007)

In the following two decades the language was progressively extended with constructs such as templates (a form of genericity with wide -ranging applications) and multiple inheritance.

C.2 OVERALL PROGRAM ORGANIZATION

In a fully object-oriented approach a program is made of a set of classes. C++, in keeping with its hybrid spirit, does not enforce this rule. A program can include classes but also other elements that do not belong to any class: functions (corresponding to routines in Eiffel, that is to say, including procedures as well as true functions), variables, constants and non-class types.

As an example of stand-alone function, every program includes a function called main, which defines the program entry point. In Eiffel this role was played by the root creation procedure.

← *"System execution", 6.8, page 130.*

Rather than a set of classes, a C++ system is a set of *translation units*, each contained in a source code file which can be processed independently by a C++ compiler. Each translation unit can contain *declarations* of classes and other types, functions, variables and constants.

The declaration of such an element may be a *definition*, meaning that it contains the full description of the element, but it may also be a non-defining declaration, such as

```
class Person;
enum Week_day;
```

with the understanding that the rest of the definition of the given elements — here a class and a type — appears in another unit, or later in the same unit. This makes it possible to use the element without knowing its detailed properties. For a variable or constant, a non-defining declaration will use the keyword extern to indicate that the definition appears elsewhere:

```
extern bool has_error;
extern const double pi;
```

A function definition includes the name, signature and implementation:

```
int factorial (int n)
    {
        if (n > 1) {
            return n * factorial (n - 1);
        } else {
            return 1;
        }
    }
```

A class definition contains the list of class *members* (features).

```
class Person { // Objects that represent persons
    string name; // Person's name
    Date birth_date; // Person's birth date
    void set_name (string s)
        {name = s;}
    void set_birth_date (Date d)
        {birth_date = d;}
    … // Other members
};
```

A variable definition indicates the type (coming before the variable, instead of the Eiffel convention *variable_name*: *TYPE*):

```
int n;
bool has_error = false;
```

The second of these has an *initializer* part, which sets an initial value. The definitions above do not contain the extern keyword: it means that the variables can be used only within the given translation unit.

Defining a variable implies that memory will be allocated for it, as with Eiffel's expanded types.

A constant definition always includes an initializer:

```
const double Pi = 3.14159265358
```

Unlike in Eiffel, order of declaration matters: names declared in a translation unit, become usable only after the declaration. To use an element ahead of its definition — as needed for example in the case of mutually recursive definition — use a non-defining declaration first.

To avoid a translation unit becoming too large, you may split it into several files and rely on the #include directive, as in:

```
#include "filename"
```

which has the effect of making all the definitions contained in the source file filename available to the current translation unit. A common use of this facility is to use *header files* (with conventional filenames of the form *name*.h), containing declarations of elements used by many units, which all #include it.

A #include directive, like any other starting with #, is intended for the C++ *preprocessor*, a tool that is run on program files before the compiler processes them. Other uses of the preprocessor include *conditional compilation*, which enables you to set preprocessor variables and include certain code only if the corresponding variables have been set. A typical example is

```
#ifdef LINUX
    … Linux-specific code …
#endif
```

to include code on the Linux platform only, depending on the LINUX variable. Such variables, unrelated to the program's variables, are set or unset as options, prior to preprocessing and compilation.

C.3 BASIC OBJECT-ORIENTED MODEL

Like in Eiffel, every C++ variable and constant has a type, but unlike in Eiffel not all types are based on classes, and consequently not all values are objects. A type is one of: built-in; derived (with possible combinations of derivation mechanisms); user-defined.

Built-in types

Built-in types are preset in the language. They include: bool for boolean values; char, short int, int and long int for integer values with various range; float, double and long double for floating point numbers with different range. Type char also serves to store characters.

Each integer type also has a signed and an unsigned versions, such as signed short int and unsigned short int. Unsigned versions only include positive values. By default all integer types except char are signed, making types such as signed int redundant; whether char is signed is platform-dependent.

The built-in type void does not have values. It serves as a return type for functions that do not return a value (procedures in Eiffel), such as

```
void set_name (string s);
```

Derived types

Derived types are constructed from already existing types using one of five type transformations: *constant, pointer*, *reference*, *array* and *function*. The following examples apply these transformations to obtain new types from an existing type T.

The type const T represents immutable values of type T. For example, a constant n defined as

```
const int n = 5;
```

will have type const int. You may assign an expression of type const T to a variable of type T, but not the other way around.

T* is the type “pointer to T ”. A value of this type denotes the address of the memory location where a variable of type T is stored. To obtain a pointer to a variable x, use & as in &x. The other way around, if p is a pointer, the dereferencing operator *, as in *p, yields the value stored at the location to which p points; if that value is an object, you can access a field f through dot notation applied to the dereferenced pointer, as in (*p).f. The special syntax p–>f is a synonym.

Because a pointer is simply a memory address, you can subject it to arithmetic operations, in particular add or subtract an integer to a pointer to get a new address, as in *(p + n) which yields the value stored at the (n * sizeof T)-th byte after p, where p is of type T* and sizeof T yields the number of bytes occupied by an instance of type T. This is known as *pointer arithmetic* and is widely used in low-level C programs to access specific memory locations. The mechanism is error-prone (since it is hard to guarantee the type of the value that will appear at such a dynamically computed address) and should generally not be used in C++ applications.

All pointer types conform to the special pointer type void*, making it partly similar to the Eiffel type *ANY*, but only partly since this conformance is built-in and not induced by inheritance. As you cannot have variables of type void, you cannot dereference void* pointers, nor can you perform any other operations on them. To use such pointers you should perform an explicit *typecast* (a “run-time type identification” operation) to attach the value stored in the pointer to a variable of another type. There are multiple typecast operations in C++; they have different results in case of unsuccessful cast.

← “Uncovering the actual type”, page 599. A typecast is also known as a dynamic cast

T& is the type “reference to T ”. Like a pointer, a reference represents an address, but any use is automatically dereferenced. References are the most direct way to obtain the effect of ordinary (reference) class in Eiffel. With an Eiffel class class *PERSON* … end, the equivalent of a client routine

```
call_her_izzy (p: PERSON)
    do p.set_name ("izzy") end
```

This is Eiffel, not C++..

will be, in C++:

```
void call_her_izzy ( Person& p)
    {p.set_name ("izzy");}
```

with a call of the form call_her_izzy (Isabelle) if Isabelle has type Person. The effect is to change the *name* field of the referenced object. If the argument in the C++ version had the type Person, the call would create a copy of the Person object; the routine would work on that copy, but with no useful effect since the copy is lost on return.

To implement the reference behavior with pointers, you must make the use of the address and the dereferencing explicit:

```
void call_her_izzy (Person* p)
     {p->set_name ("izzy");      // or (*p).set_name ("izzy");
     }
```

The call in this case will be call_her_izzy (&Isabelle). It is considered good C++ style to use pointers rather than references in such cases, as it makes the reference semantics explicit.

Another difference is that a reference is enforced to have an initializer, which attaches a value of type T to a T& reference, whereas pointers can have zero value (also called NULL and corresponding to Eiffel's Void). This does not, however, provide the benefits of Eiffel's attached types, since it is possible to assign a dereferenced T* pointer to a variable of type T& or T, causing a run-time error if the pointer is null.

← *"Appendix: getting rid of void calls", 6.9, page 136.*

More generally, reference types enforce a stricter type discipline; in particular, they do not allow pointer arithmetic.

Now for array types. The array type T[size], where size is an integer constant known at compile time, represents sequences of values of type T stored in size adjacent memory locations. If a is of that type, you can use [*i*], where *i* is an integer expression, to access the *i*-th array element. A C++ array is really just an address at which enough locations are assumed to be available to store size elements. For safe array handling with index control avoiding out-of-bound array access, you may use library classes.

Finally, function types. They actually are *pointer to function* types. If R, A1, …, An are types,

```
R (*f) (A1, …, An);
```

declares a variable f whose values are pointers to functions with return type R and argument types A1, … An. For example, with the declaration

```
void (*f) (Person*)
```

you can assign to the variable a value denoting a function pointer:

```
f = call_her_izzy; //or f = &call_her_izzy;
```

and then perform an indirect call:

```
f (Isabelle);                          // or (*f) (Isabelle);
```

After the previous assignment this will have the same effect as a direct call call_her_izzy (Isabelle); the difference is that f is a variable, and could be assigned pointers to many other functions.

A function pointer can denote not only a standalone functions, but also a member function of a class, as in

```
void (Person::*p) (string)
…
p = Person::set_name;
```

which declares and assigns p as a pointer to a function from class Person, taking one argument of type string. You may call that function through the .* operator:

```
(Isabelle.*p) ("Izzy");
```

As you have surely started to reflect, pointers to functions are closely related to two object-oriented mechanisms that we have studied, both of which rely on the ability to call a routine but leave until run time the determination of exactly which routine that will be: *dynamic binding* and *agents*. C++ function pointers indeed make it possible to emulate both of these facilities:

← *"Dynamic binding", 16.3, page 562; agents are the topic of chapter 17.*

- They are the best way to obtain the effect of agents in C++, although they lose the idea that an agent represents a routine with all its properties (all you can do with a function pointer is to call the function, the equivalent of the routines *call* and *item* on agents), and some of the typing guarantees.
- C++ function pointers would also enable you to obtain the effect of dynamic binding, by using an object's dynamic type as an index into an *array* of function pointers to find the right version of a function. This technique was described in detail in the discussion of inheritance implementation. You do not need to use it in C++ since the *virtual function* mechanism, studied below, gives you dynamic binding. (The availability of both techniques can be confusing to C++ beginners.) You will need it if you are programming in C, rather than C++, since C has no object-oriented mechanisms and in particular no virtual functions. This is also why Eiffel compilers that use C as their target language also rely on it, through the scheme explained in the earlier discussion.

← *"A peek at the implementation", 16.8, page 575.*

Combining derived type mechanisms

The five mechanisms for building derived types, as just reviewed, can be combined in various ways. Here are some of the most important.

You can combine the const and pointer modifiers to obtain:

- A *constant pointer*, which will always point to the same memory location throughout execution, although the value at that location may itself change.
- A *pointer to a constant*, which can be changed, but may not be used to change the value at any location to which it points. (But such a location may also be the target of a non-constant pointer, in which case the value can be changed through that other pointer.)
- *Constant pointers to constants*, for which both the memory location and the pointed value are immutable.

You may similarly combine const with references. The following examples illustrate some of the possibilities:

```
const int* pointer_to_const;
const int& reference_to_const;
int* const const_pointer;
int& const const_reference;
```

Also important in practice is the typedef notation that enables you to give a name to any type. This makes it possible to refer to complex derived types through simple names as in

```
typedef const Person* Cp ;
typedef void (* Pf ) (Person*);
```

which make it possible to use Cp rather than const Person *, as in a declaration Cp p; similarly, Pf denotes a function pointer type.

User-defined types

In addition to classes studied next, user-defined types include *enumeration* types which enable you to represent a fixed, usually small set of values, as in:

```
enum Week_day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
enum Error {Division_by_zero, Null_pointer_dereference, File_error, Memory_error};
```

Internally, the values of an enumeration are integer constants; enumeration types conform to integer types, but not vice versa, to help ensure that the value of an enumeration variable remains within bounds.

Another kind of user-defined type is the *structure* type, usually called just "struct", a form of class. Classes are studied below; structures have essentially the same properties but a different export policy.

Finally, a union type describes objects which can be of several kinds all occupying the same storage, defined using the keyword union instead of class or struct. This mechanism is a carry-over from C where it was intended for storage optimization; it is not type-safe, since if you use p.a you cannot be sure that p is of the right variant having the attribute a. For this reason C++ programmers rarely use unions, relying instead on inheritance, as in other object-oriented languages, to support variants of a common type.

Classes

You may define a C++ class through the following syntax:

```
class A{
    … // list of members
};
```

This may occur at any place in a translation unit where definitions are allowed, including inside other classes and even inside function bodies, although such nesting is not common. Similarly, it is preferable (but not required) to include only one class in a file, as you would do in Eiffel.

You can use a class as a type for variables and to construct other derived and user-defined types:

```
A var;              // variable of type A
const A const_var;  // constant of type A
A∗ p;               // pointer to objects of type A
```

For class members (features), C++ distinguishes between *member variables*, also called *data members* (corresponding to Eiffel attributes), and *member functions*, also called *methods* (correspond to Eiffel routines).

A class definition can contain both definitions and non-defining declarations of members. In the latter case members should be defined outside of the class:

```
class A{
        int n;                 // member variable
        void f ()              // member function defined inside the class
             {…}
        void g ();             // member function defined outside the class
};

void A::g () // definition of g
{…}
```

C++ does not apply the Uniform Access principle, instead treating qualified access to a member variable and invocation of a member function as different operations. For a function you must always use parentheses to enclose the argument list even if that list is empty:

← *"Touch of Methodology: The Uniform Access Principle", page 246.*

```
A var;
int i;
i = var.n;
var.f ();
```

Another important difference from Eiffel (and departure from another principle, information hiding) is that you may directly assign to object fields:

← *"Information hiding: modifying fields", page 240.*

```
var.n = 5;
```

Needless to say, this is not recommended — use setter commands instead.

← *"Setters and getters", page 248.*

Special member functions called *constructors* serve to create objects. Here is an example of a constructor:

```
class A{
     int n;
                    // The next two lines define a constructor
    A (int i)
         {n = i;}
…
};
```

A constructor plays the same role as an Eiffel creation procedure but there are important differences. As this example illustrates, constructors do not have their own names but use the name of the class, here A; this still makes it possible to have several constructors as long as they have different type signatures (a case of *overloading* as studied below). To initialize a variable of a class type you must call a constructor, as in:

```
A var = A(2);
A another_var (2);
```

where the second form is an abbreviation for the first. A class may have a *default constructor* with no arguments (think of *default_create* from class *ANY* in Eiffel), enabling you to omit the explicit initialization:

← *“Creation procedures”, 6.5, page 122.*

```
A var; // synonym for A var = A ();
```

In the absence of a constructor declaration, the class is considered to have a default constructor that sets all the data members to default values. If, however, any of the class members is of a type that does not have a default constructor, the class must include at least one explicit constructor.

A constructor assumes that prior to its execution all member variables have been set to default values. These values are language-defined for built-in types, and for classes are given by the default constructor if present. In its absence you must provide a default value through a *member initializer list*, which may also include values for any other data members, as in:

```
class B {
    int value;
    Person a_friend;
    B (int v, string name) : value (v), a_friend (name) {}
            // Can also be written: B (int v, string name) : a_friend (name)
            // {value = v;}
};
```

The process for constructing an object involves: constructing the fields, using the initializer list if supplied and default constructors otherwise, in the order in which the member variables are defined inside the class; then executing the constructor’s body.

To denote the current object (Current in Eiffel) you may use the keyword this, defined as a constant pointer.

It is common in C++ practice to define functions that both change the state and return a value, departing from the Command-Query Separation principle, although nothing prevents C++ designers from following this principle. Indeed you can guarantee that a function does not change the state by declaring it const:

```
class A {
    int n;
    int n_squared () const
    {return n∗n; }
};
```

Such a function may not modify the current object by assigning to member variables or calling non-const functions.

Information hiding

In Eiffel you define clients' access privileges through export specifications, which can be selective. In C++ you may define, for every member of a class, one of three *access modifiers*: public, protected and private. Public access means full rights for clients and descendants (as with exporting to *ANY* in Eiffel). Protected access means exporting to the class itself and its descendants. Private access prohibits access in class descendants, unlike Eiffel where descendants always have full access for unqualified calls. These rules indeed do not differentiate between unqualified and qualified calls (so that there is no equivalent to making a feature completely unavailable through qualified calls, by exporting it *NONE*). The following examples use some of these access modifiers

← Selective exports were seen in "Overall inheritancestructure", 16.10, page 586.

```
class A {
private :
    int n;
    float secret_function () { … }
protected :
    float variable_for_descendants;
    int n_squared () { … }
public :
    string variable_for_everyone;
    do_everything () { … }
};
```

In practical usage, member variables are rarely declared public since this makes them available for assignment as well as access. The common practice is to define a getter; we saw why this is not needed in Eiffel.

← "Setters and getters", page 248.

The default for class members in the absence of an access modifier is private. For *structures*, however, the default is public.

These techniques do not exactly support the notion of "selective export" (as in feature {*A*, *B*, *C*} in Eiffel, which exports the features that follow to the specified classes and their descendants), but there is a closely related mechanism allowing a class to designate a function or another class as its *friend*:

← "Overall inheritance structure", 16.10, page 586.

```
class Linkable {
    friend class Linked_list;
    friend bool is_equal (const Linkable& other);
    …
};
```

A friend can access all class members, including private and protected ones. (This means that the level of granularity of the friend mechanism is coarser than with selective export, which specifies access to individual features). Unlike a member function, a friend function is not called on the current object; hence it may not use this, and may only access private members in a qualified way.

Scoping

Any variable, constant, function or type has a scope: local to a block (a program part enclosed in braces {…}) if this is where its declaration appears, otherwise global (extending to the entire translation unit).

A consequence is that there is no special syntax for local variables, which are simply defined anywhere in the block making up a function body. It is actually considered good practice in C++ to define local variables as close as possible to the point of their first use.

Blocks can be nested. A declaration in an inner block *hides* an element defined with the same name in an outer block or globally:

```
int x; // global x
void f ()
        {x = 1;          // Assignment to global x
        int x;           // Defines a local x
        x = 2;           // Assignment to local x
        }
int y = x;           // Uses the global x
```

A hidden element can still be accessed in an inner block through the :: notation, the *scope resolution* operator:

```
int x;                   // Global x
class A {
    int x;               // Member x
    void f ()
        {int x;          // Local x
        int y = A::x;    // Uses member x
        int z = ::x;     // Uses global x
        }
};
```

These facilities are error-prone and it is preferable to choose different names in nested scopes.

Operators

As in Eiffel, you can define functions which will be called in operator syntax:

```
class Complex {
    …
    Complex operator+ (const Complex& other) const {…}
    Complex operator- (const Complex& other) const {…}
    Complex operator∗ (const Complex& other) const {…}
    Complex operator/ (const Complex& other) const {…}
};
```

Unlike in Eiffel, operator functions do not have an equivalent identifier name, and you are restricted to a fixed set of predefined operators (such as + and such) whose syntax properties — number of arguments, infix or prefix use, left- or right-associativity — cannot be changed.

Overloading

It is possible in C++ to declare several functions with the same name in the same scope, if they have different signatures as in

```
void print (int n) {…}
void print (string s) {…}
…
print (5);            // Uses the integer print function
print ("hello");      // Uses the string print function
```

This mechanism is called *function overloading*. It is often used in particular for operators ("*operator overloading*"). It is also what permits all constructors of a class to use the same name — the class name.

Static declarations

In a pure object-oriented framework, all elements of the program are relative to the current object (we called this "general relativity"). C++ adds a mechanism to describe member variables and functions, said to be *static*, that belong to a class but can be applied independently of any of its instances. (In Eiffel the equivalent effect is obtained through the notion of once routine, not used in this book but described in the standard documentation.)

← *"The current object and general relativity", page 132.*

See e.g. *docs.eiffel.com/book/method/10-other-mechanisms.*

Unlike usual member variables, which represent fields of every instance of a class, a *static member variable* represents data at the level of the class itself. For example, a static data member can be used to count how many times a certain member function of the class was called:

```
class Rocket_launcher {
    …
    static int rocket_count;
    static const int max_count = 100;

    void launch ()                  // Launch a rocket
    {
        …
        rocket_count++;             // Increase by 1
    }
};
```

A *static member function* operates only over static member variables and constants, as in:

```
class Rocket_launcher {
    … Rest of class as above …
    static bool is_in_bounds ()
    {return rocket_count <= max_count;}
};
```

Calls to static member variables and functions do not need a target object; instead of dot notation you may use the scope resolution operator:

```
r = Rocket_launcher::rocket_count;
m = Rocket_launcher::max_count;
if (Rocket_launcher::is_in_bounds ()) …
```

Another use of the static keyword is to declare a *static function local*, which will preserve its value between function invocations, as in

```
void f ()
    { static int invocation_count = 0;     // static local
    …
    invocation_count++;                    // increasing by one
    }
```

Object lifetime

C++ differentiates between objects whose storage is automatically managed by the runtime system and those under direct programmer control.

Programmer-controlled objects are also called *dynamic objects* or *heap objects*. Programmers bring dynamic object into existence by calling the new operator with a constructor:

```
Complex∗ c;
…
c = new Complex (1.0, 2.0);
```

Evaluation of the new expression yields a pointer to an object of type Complex, causing the assignment to have the same effect as an Eiffel creation instruction **create** *c*.*make* (*1.0, 2.0*) with *c* of type *COMPLEX*.

Unlike Eiffel, C++ is not designed for automatic garbage collection. It is the programmer’s responsibility to determine when a dynamic object is going to become unreachable and remove it from the heap using the delete operator:

← *“Memory management and garbage collection”, 6.7, page 128.*

```
delete c;
```

(For the same purpose C provides a library function called free.) If after executing this instruction some pointers or references are still referencing the object that was attached to c before, they will become *dangling pointers*: dereferencing such a pointer means trying to access a non-existent object and will cause a runtime error or yield an arbitrary value. On the other hand, failing to delete objects that have become unreachable will cause memory leaks.

While dynamic objects are always accessed through pointers — with the possibility of several pointers attached to the same object — a non-dynamic object (which can still have “secondary” pointers and references attached to it) is associated with a single variable or constant, whose scope determines the object’s lifetime. A non-dynamic object can be of two kinds:

- Variables local to a function or other block yield *automatic* (or *stack*) objects; they are managed on the call stack, allocated at the point of definition and forgotten on block exit.
- Global variables and variables declared static (static data members, static function locals) yield *static* objects: they are created before the invocation of the main function and exist throughout execution.

The lifetime of object fields (which correspond to non-static member variables) is determined by the lifetime of the object to which they belong. The destruction process will be applied first to an object and then to its fields. In the absence of automatic garbage collection it is often necessary to specify certain operations to perform when an object is deleted (either through an explicit delete or, for automatic objects, on block exit). Any class C can define for this purpose a special *destructor* member function, of name ~T, which will be invoked on object destruction. Here is a typical destructor:

```
class Person1 {

    Passport* pp;
    Person1 (string n, date d)          // Constructor, creating a dynamic
                                        // Passport object
        {pp = new Passport (n, d); }
    ~Person1 ()          // Destructor
        {delete pp;}     // Delete passport
};
```

This example illustrates a common C++ pattern, *Resource Acquisition Is Initialization (RAII)*: use the constructor to acquire resources that an object needs and the destructor to free them. This removes some sources of errors by ensuring that delete operations happen in the right order. Systematic application of RAII confines use of dynamic memory to specific classes, often in libraries; the rest of the software uses instances of these classes non-dynamically, alleviating some of the consequences of the absence of garbage collection.

The RAII pattern extends to resources other than memory, such as files, sockets (network connections) and locks (for multithreading and other forms of concurrent programming), for which the releasing of resources had to be handled manually even in the presence of a garbage collector. In addition, RAII ensures that in the case of abnormal termination (through exceptions, discussed below) destructors will be called properly. These benefits have led some proponents to state that RAII is superior to garbage collection; it remains, however, a manual approach limited to specific patterns of memory usage.

Initialization

Unlike in Eiffel, automatic initialization only applies to static objects (preset values for built-in types, default constructor for class types). So declaring

```
int n;
class Rocket_launcher {
    static int rocket_count;
    … Rest as above (page 819)…

};
int Rocket_launcher::rocket_count;
```

initializes both n and rocket_count to zero. (You need to include the second declaration of rocket_count, since the declaration of a static member variable within a class is non-defining.)

You must initialize references, constants and *automatic* objects manually, using the = *value* syntax. The syntax in all cases is that of assignment, using = (equality is ==); for a constant the assignment is part of the declaration, and no further assignment is permitted.

Omitting the initialization of an automatic object will result in its initial value being undefined, which is almost always an error (and the source of potential security break-ins). So you must manually, and carefully, check that every automatic object has a proper initialization.

"Static analysis" tools can help. See "Verification and validation", 12.4, page 341.

Exception handling

C++ exception handling is available to process abnormal run-time events. Rather than the Eiffel style, based on Design by Contract principles, it uses the "try-catch" style.

← "An introduction to exception handling", 7.10, page 200.

An exception, caused for example by an erroneous arithmetic operation (overflow or underflow), will interrupt the normal flow of control. You can allow a block to *catch* the exception and provide recovery instructions by enclosing it in a try clause, whose catch part describes the recovery, as in this example:

```
try {
     … Code that might trigger exceptions …
} catch (io_error e) {
     … Processing for I/O exception …
} catch (memory_error e) {
     … Processing for memory exception …
}
```

The catch clauses specify an exception type, such as io_error, and a name for the exception, here e, used by the following instructions (in a way similar to a formal argument of a routine) to access specific properties of the exception.

You can also trigger, or *throw*, an exception through the instruction

```
throw exp
```

where *exp* is an expression. Although it can be of any type, the practice is to use special library classes specifically designed to describe exceptions.

Any exception occurring during the execution of a block interrupts the execution of that block (the remaining instructions are not executed). Then:

- If the block is in a try clause and one of the catch specifications matches the exception's type, execution will proceed with the corresponding catch block, then move on to the next construct (unless the catch block re-throws the exception through throw ()).
- If there is no matching handler, or the exception occurs outside of a try block or is re-thrown, the current function terminates and throws an exception to its caller, which follows the same policy.

If no matching catch block appears in the call chain, the exception ends up interrupting main and hence terminating the program in an erroneous state.

In this process of terminating functions and passing the exception up the call chain, known as *stack unwinding*, all automatic objects are destroyed, using destructors if available. This is one of the places where RAII helps.

For safer exception handling you can specify, as part of a function's signature, a *throw set*: the list of exceptions that its execution might trigger, as in

```
void read_and_store (string a_file_name) throw (Io_error, Memory_error)
{ … }
```

The absence of a throw set means that the function can throw any exception (to specify that it can throw *no* exceptions use throw ()). Systematically including throw sets, with the rule that each function's throw set should be a superset of those of the functions it calls, is recommended discipline since it helps avoid missing exceptions; this is difficult to enforce, however, because libraries and existing code used by new systems may not follow the same discipline.

Templates

Templates provide the C++ version of genericity. Here is a simple example:

```
template <typename G> class Stack {
…
public:
    G item () {…}
    void push (G an_item) {…}
    void pop () {…}
};
```

This defines Stack as a class parameterized by the type name G, as with an Eiffel class *STACK* [*G*]. To define a particular stack, you may use

```
Stack<int> s;
```

The main difference between C++ templates and generic classes found in other O-O languages is that every generic derivation such as the above is considered to produce a new class, a process called template *instantiation*.

This is a form of *preprocessing* — transformation of the program text prior to compilation — which actually has the full power of a compile-time programming language and has found some exotic applications in advanced C++ programming.

There is no counterpart to the notion of constrained genericity: if you apply an operation to a variable of a formal generic parameter, such as G above, type checking will be applied to every instantiation to ensure that the operation is always valid.

← *“Constrained genericity”, page 596.*

From a class template it is possible to define full or partial *specializations*. A full specialization freezes the actual parameters, as in this example using the above Stack:

```
template<>
class Stack<bool> {
    … Operations specific to stacks of booleans …
};
```

A partial specialization would retain a formal parameter but impose some limitations on it:

```
template<typename G>
class Stack<G*> {
    … Operations specific to stacks of pointers …
}
```

As these examples suggest, a specialization can have its own member definitions; this does not cause conflict since compilation always chooses the most specialized concrete version.

Beyond class templates, C++ supports *function* templates, as in:

```
template <typename G>
G max (G a, G b) { … Computation of maximum …}
```

In instantiating function templates you may omit actual parameters if it is possible to infer them automatically from actual argument types, as in:

```
int a, b, c;
…
c = max (a, b);        // Calls max<int>
```

which automatically instantiates max with int for G.

The template mechanism goes beyond genericity by allowing template parameters to include boolean and integer values, not just types. The corresponding actual generic parameters must be compile-time constants. The following example uses this possibility to define multiplication of fixed-size matrices with a guarantee that the sizes match:

```
template <int n, int m> class Matrix { … };
template <int n, int m, int k> Matrix<n, k> operator*
          (const Matrix<n, m>& m1, const Matrix<m, k>& m2)
{… Matrix multiplication algorithm … }
```

Trying to multiply matrices of incompatible (constant) dimensions would result in a static (compilation-time) error.

C.4 INHERITANCE

You can define a class B to be a *derived class* (heir) of a class A (a *base class* for B) in the following way:

```
class B : A {
     …
};
```

Overriding

There is no equivalent to Eiffel's renaming and undefinition. To redefine (or *override*) an inherited member function, simply include a new declaration; you must be particularly careful to use an identical signature, since otherwise the new declaration would be understood as simply overloading and there is no simple way to detect such mistakes.

Export status and inheritance

By default inherited members are private. To change their export status, you can specify an access modifier — private, public or protected — for an inheritance relationship, as in

```
class B : public A {…};
```

The effect of such a specification is to give each inherited member the minimum of the access rights of the original member and the access right specified for inheritance; so in this example all inherited members keep their A status in B.

Precursor access

To access the original version of a redefined function — the equivalent of Precursor in Eiffel — you may use the scope resolution operator as long as the that version is not private:

← *On* Precursor *see page 573.*

```
class B : public A {
     void b_function ()            // Need not be the redefinition of r
     {
          A::r ();                 // Will call A's version
          …
     }};
```

Static and dynamic binding

We saw that dynamic binding was a central contribution of object technology to software architecture. A major difference between C++ and other O-O languages such as Eiffel is that binding is static by default: the criterion for determining which version of a function to call is the declared type of the target expression, not the dynamic type of the object attached to it at run time. You may subject a particular function to dynamic binding by declaring it virtual:

← *"Dynamic binding", 16.3, page 562.*

```
class Rectangle {
…
     virtual void draw() {…} // should be declared as virtual
     virtual void rotate() {…}
};
class Rounded_rectangle : public Rectangle {
…
     virtual void draw() {…}    // Specifying "virtual" again in redefinition
     void rotate() {…}          // … but this is optional (same effect:
                                // dynamic binding still applies!)
};
```

Dynamic binding only applies to objects accessed through pointers or references, as illustrated by this example:

```
Rectangle r (1.2, 0.5);
Rounded_rectangle rr (5.0, 3.2, 0.2);
r = rr;                    // r still a plain rectangle, with fields of rr partly copied
r.draw ();                 // Static binding: rectangle::draw()
Rectangle* p = &rr;
p –> draw ();              // Dynamic binding: rounded_rectangle::draw()
Rectangle& ref = rr;
ref.draw ();               // Dynamic binding: rounded_rectangle::draw()
```

Constructors, which are not applied to an existing target object, cannot be virtual. Destructors can and often should be virtual, to make sure upon object reclamation the appropriate resources will be released.

Pure virtual functions

The closest equivalent to deferred features is the notion of *pure virtual* function, which have a definition but no implementation:

← *“Deferred classes and features”, 16.5, page 565.*

```
class Figure {
      virtual void draw() = 0;
      … Other members …
};
```

A class with at least one pure virtual function is called an *abstract class*, similar to deferred classes.

Multiple inheritance

C++ supports multiple inheritance:

```
class Arrayed_stack : public Stack, private Array {…}
```

The mechanism is less flexible than what we have seen in this book. In particular the language, as noted, does not support renaming. You may inherit two identically named functions, and will use the scope resolution operator to disambiguate them:

← *“Multiple inheritance”, 16.11, page 588.*

```
class A {void f () {…}};
class B {void f () {…}};
class C : public A, public B {…};
void test()
{
      C∗ p = new C();
      // p–>f();          This call would be ambiguous and hence invalid
      p–>A::f() ;
      p–>B::f() ;
}
```

Repeated inheritance does not allow you to choose between sharing and replication for each member, only globally for the inherited class; replication is the default. Disambiguation may require complicated uses of scope resolution:

← *“From multiple to repeated inheritance”, page 592.*

```
class D {int n;};
class E : public D{};
class F : public D {};
class G : public E, public F {};
void f()
{
    G∗ p = new G();
    // p–>n = 0;             // This would be invalid: which n?
    p–>E::D::n = 0;         // Valid, assigns to version replicated in E
    p–>F::D::n = 0;         // Valid, assigns to version replicated in F
}
```

If you want the common ancestor's fields to be joined instead of replicated, you should define this ancestor as a *virtual* base class:

```
class T {int n;};
class U : public virtual T {};
class V : public virtual T {};
class W : public U, public V {};
void f()
{
    W∗ p = new W();
    p–>n = 0;                // Now valid
}
```

The disadvantage is that the choice is made not at the point of using repeated inheritance, here in class W, but earlier, in classes U and V, which might not know about W's plans to inherit from both of them.

Inheritance and object creation

C++ has specific rules regarding the creation of instances of derived classes. Constructors are not inherited, but the creation of a derived class instance causes a call to the parents’ constructors (a recursive process) before the class’s own constructor. It is possible to supply arguments to the parent’s constructor, as in the following example

```
class Rounded_rectangle : public Rectangle {
public:
    Rounded_rectangle (float w, float h, float r) : Rectangle (w, h), radius (r)
        {…}
… Rest of class as before …
};
```

C.5 FURTHER PROGRAM STRUCTURING MECHANISMS

In large software systems it can be difficult to avoid name conflicts; a system may in particular use libraries that include classes with the same names. C++ provides *namespaces*: named blocks whose only purpose is to restrict the scope of names declared in each block:

```
        // In "some_library.h":
namespace some_library {
    class Parser {…};
    class Lexer {…}
    …
}
        // In "your_program.cpp":
#include "some_library.h"
namespace your_program {
    class Parser {…};           // No ambiguity: different scope
}
```

To resolve any ambiguities it suffices to use the scope resolution operator. If this becomes too tedious when you are frequently using a name from another namespace, you may introduce a local name through the using notation:

```
using some_library::Lexer;
Lexer lexer;            // Shortcut for some_library::Lexer lexer
```

Or you can do this globally for all the names from a namespace:

```
using namespace some_library;
```

C.6 ABSENT ELEMENTS

The following is a brief review of mechanisms to which you have become used in learning programming through this book, and which have no direct equivalent in C++. It includes some suggestions about how to emulate their effect.

Contracts

C++ does not support the Design by Contract mechanisms (preconditions, postconditions, class and loop invariants, loop variants) which play an essential role in modern programming methodology as developed in the present book.

You may use the instruction

```
assert b;
```

(where b is a boolean expression) to express that b should hold at this program point during execution. With the suitable compilation option, this will trigger a run-time check that will produce a message and terminate execution — a surprisingly result, as one would rather expect an exception! — if the condition is not satisfied.

This mechanism makes it possible to use assertions for debugging, but of course falls short of all the other applications of contracts — for class and routine specification, documentation, design etc.

Many people have proposed C++ extensions or macro packages (a macro is a preprocessor instruction) to emulate Design by Contract. A Web search for “Design by Contract in C++” will yield references to many of these tools, whose use has remained limited since they are not integrated with the language.

Agents

As noted, C++ does not have an agent mechanism. The simple effect of calling a variable function can be achieved (as we have also seen) through function pointers. A more sophisticated solution uses the notion of “functor”, implemented by overloading operator (), the function call operator, making it possible to call a functor object like any other function. The disadvantage is that you must define a separate functor class for every possible number of formal arguments.

Constrained genericity

We saw that C++ has no direct counterpart to constrained genericity and that each template instantiation is type-checked on its own. This means that it is impossible to inform client authors formally that a generic parameter is supposed to represent descendants of a particular type; they will only find out if they violate this intent, as the template instantiation (using non-applicable features) will not compile.

As a methodological rule, you should at least express the intent informally by specifying the constraining type as a comment in the template definition:

```
template <typename G> /* G should be a descendant of Comparable */
G max (G a, G b) { … }
```

Overall inheritance structure

C++ does not have the equivalent of a top class in the inheritance hierarchy, such as Eiffel's *ANY*, or of a bottom class such as *NONE*.

C.7 SPECIFIC LANGUAGE FEATURES

We have encountered a number of C++ features not available in Eiffel. Here we review two other C++ peculiarities: argument defaults and nested classes.

Argument defaults

You may define a default value for a formal argument, enabling calls to omit the corresponding actuals. If not all formals have defaults, those with defaults should follow the others:

```
void f (float x, float y, int n = 1 , char c = '!' ) { … }
f (1.2, 5.0, 2, 'a');                      // You may still include all actuals
f (1.2, 5.0, 2);                           // c has default value '!'
f (1.2, 5.0);                              // n has default value 1, and c default value '!'
```

You may also define a function with a variable number and types of arguments, using an ellipsis (…) instead of formal arguments list. This mechanism is not type-safe and is more appropriate for system-level C programming (see next) than for application programs written in C++.

Nested classes

C++, as noted, allows many forms of nesting. In particular you may declare a class inside another class or even a function. A class nested within a class is also called a *member class*; like any other member it may be private, protected or public.

Private member classes represent data abstractions of interest only to the enclosing class. An alternative solution would be to declare the class on its own and use the friend mechanism to restrict member access to the intended class, although as we saw this makes *all* members accessible to the friend, whereas a member class can have its own private members, not accessible to the enclosing class.

C.8 LIBRARIES

Often used in C++ applications, the Standard Template Library, or STL, covers fundamental data structures, in particular containers (which require genericity, so that most of the classes are templates, hence the library names), exceptions and input-output.

For input and output STL uses *streams*, which can represent media such as the console (standard streams cin and cout), files and strings. Writing and reading use the overloaded bit shift operators >> and <<, so that a typical console interaction looks like this:

```
Person p (…);
int my_age;
cout << “Name: “ << p.name << endl << “Age: “ << p.age () << endl;
cout << “Enter your age: “ << endl;
cin >> my_age;
```

where endl stands for the end of the line.

Many third-party libraries are also available.

For a starting point see for example the peer-reviewed libraries at www.boost.org..

C++ retains the C standard library, which it is preferable to avoid in C++ applications, as many of its facilities are low-level and not type-safe.

C.9 SYNTACTIC AND LEXICAL ASPECTS

The C++ grammar for instructions and expressions is complex; only the basic elements will be reviewed here.

Instructions as expressions

A key concept is the *expression statement*: an expression followed by a semicolon. This notion seems paradoxical since we have maintained in this book a clear distinction between instructions and expressions, in line with the strict distinction between commands and queries. C++, however, does not insist on this separation, and as a consequence an expression statement is both an expression, which returns a value unless its type is void, and an instruction.

> Correspondingly, even if a function returns a value it can have side effects; it is common in C++ to use a call to such a function as an instruction. In this case the return value is simply lost.

One of the consequences of the fusion of the concepts of expression and instruction is that an assignment (using =) is also an expression, whose value is whatever was assigned to the target. This makes it possible to write such combinations as:

```
a = b = 5;
```

which assigns the value 5 to b, then the result of this assignment — the resulting value of b, again 5 — to a. Such schemes are confusing and should be avoided.

Control structures

Blocks correspond to Eiffel compound and consist of a list of statements in braces {…}. Blocks, as we saw, can be arbitrarily nested.

A conditional statement (instruction) has the form:

```
if (expression) statement else statement
```

(note the required parentheses around the *expression*). The *expression* does not have to be of boolean type; it can also be of any numeric or pointer type, with 0 (also written NULL for pointers) representing false and any other value true. Here and with other control structures, you can use a compound for each *statement* by enclosing any number of instructions in braces; it is preferable to use braces in all cases, even for a one-instruction compound, to facilitate adding other instructions later.

There is no equivalent to elseif; you have to use nesting, but the clauses do not have to look nested due to the absence of an end keyword and, visually, the use of comb-like indentation:

← *“Comb-like structure, figure on page 179.*

```
if (expression) statement
else if (expression) statement
else if (expression) statement
…
else statement
```

The multi-branch instruction has the form

← *“Multi-branch”, page 195.*

```
switch (expression) {
case value: statement; break;
case value: statement; break;
…
default: statement
}
```

where *expression* is a boolean or integer expression and each *value* is a compile-time constant. If the value of the *expression* does not match any of these constants, the instruction executes its default branch if present, nothing otherwise. (In Eiffel, in the absence of an **else** clause in an **inspect**, this case produces a run-time error.) The switch instruction by itself is not a one-entry, one-exit conditional but a multi-target goto; to obtain the effect of a multi-branch conditional you have to include break; instructions as shown. If you omit them control will flow, when a branch terminates, to the next branch.

As we saw in the discussion of control structures, it is usually best to stay away from such goto-like constructs

← *“The goto puts on a mask”, page 189.*

C++ offers three kinds of loop:

```
while (expression) statement
do statement while (expression);
for (init_statement ; expression ; advance_statement) body_statement
```

The difference between the two while variants is that the second one always executes the body (*statement*) at least once, since it tests the expression before executing the body (as in a **repeat** … **until** …, with the condition reversed); the first form can have zero executions of the body. The for loop is the most general and the most commonly used. The purpose of the *advance_statement* (included in the body in the Eiffel syntax) is to advance the iteration. So the equivalent of

← *"Other forms of loop", page 192.*

```
from i := 1 until i > n loop
    …
    i := i + 1
end
```

is, in C++:

```
for (int i = 1; i <= n; i++)
    {…}
```

C++ includes goto-like instructions: the goto itself whose use, as you know, is not recommended; break, whose use we saw in connection with switch; and return. The way a function returns its value (to be contrasted with the Eiffel convention of using the value of the special variable Result) is through

← *"Goto elimination and structured programming", 7.8, page 185.*

```
return expression;
```

which terminates function execution and returns the given value. (For a procedure — in C++, a function returning void — omit the *expression*.)

As a result of these constructs, C++ blocks are not constrained to the one-entry, one-exit structure that this book has systematically used in line with recommendations of programming methodology.

← *See the figure "Three kinds of one-entry, one-exit structure", page 188.*

Assignment and assignment-like instructions

The target of an assignment does not have to be a variable, but must denote a storage location (called a "left-value" or "l-value" since it appears on the left of the assignment symbol). Here are some examples:

```
int a; Person p;
a = 5;              // Valid
// a + 2 = 5;          Would be invalid: a + 2 does not denote a location
∗(&a + 1) = 5;      // Valid: assigns to memory location next to location of a
p.name = "Izzy";    // Valid: assigns to name field of p
```

Several C++ operators perform assignment together with some other operation. In particular:

- a += b is a shortcut for a = a + b, and similarly with operators other than +.

- Instead of a = a + 1 you may use not just a += 1 but an even shorter form: a++ or ++a. As usual, these are expressions as well as instructions; the difference is that the second expression yields the already incremented value of a (quiz: what are the effects of a = a++ and a = ++a?).

The use of = for assignment (the equality operator is ==), a departure from centuries of mathematical tradition, causes particular risks of confusion in C++ because of another property, just reviewed in connection with conditionals: the weak typing of boolean expressions, where almost any type is acceptable with the convention that 0 is false and anything else is true. A common mistake, known to have caused bugs in many programs, is to write

```
if (x = y) {Some_instructions}
```

WARNING: almost certainly a mistake — the intention must have been to use the equality operator ==.

almost certainly intended to execute Some_instructions if and only if x and y have an equal value, but producing a quite different effect: assign to x the value of y; execute Some_instructions if and only if that value was not zero. This is something that even experienced C++ programmers must guard against.

Expressions and operators

A C++ expression is a literal, an identifier, this or an operator expression. The following tables include all C++ operators. Unary operators are:

Operator	Role	Example	Operator	Role	Example
+	unary plus	+a	delete	memory deallocation	delete p
–	unary minus	–a	sizeof	size of expression's type	sizeof (a + b)
*	dereferencing	*p	++	prefix increment	++a
~	bitwise not	~a	++	postfix increment	a++
!	logical not	!b	--	prefix decrement	--a
new	memory allocation	new int (5)	--	postfix decrement	a--

and binary operators:

Operator	Role	Example	Operator	Role	Example
+	binary plus	a + b	=	receives	a = 5
-	binary minus	a - b	+=	receives plus	a += 5
*	multiplication	a * b	-=	receives minus	a -= 5
/	division	a / b	*=	receives multiplied	a *= 5
%	modulo	a % b	/=	receives divided	a /= 5
^	bitwise xor	a ^ b	%=	receives modulo	a %= 5
&	biwise and	a & b	^=	receives bitwise xor	a ^= b
|	bitwise or	a | b	&=	receives bitwise and	a &= b
&&	logical and	b1 && b2	|=	receives bitwise or	a |= b
||	logical or	b1 || b2	<<=	receives bit shift left	a <<= 1
==	equal	a == 5	>>=	receives bit shift right	a >>= 1
!=	not equal	a != 5	[…]	subscripting	a [i]
<	less than	a < 5	,	sequence	a, b = 2
<=	less than or equal	a <= 5	.	member access	x.f
>	greater than	a > 5	.*	indirect member access	x.*pf
>=	greater than or equal	a >= 5	–>	member access through pointer	px–>f
<<	bit shift left	a << 1	–>*	indirect member access through pointer	px–>*pf
>>	bit shift right	a >> 1	::	scope resolution	Person::name

The semantics of the division operator / adapts to the types of its operands: on integers it is an integer division, with at least one floating-point operand the result is floating-point.

> The *sequence* operator (the comma) is in the spirit of the language's merge of instructions and expressions; a list of expressions separated by commas is evaluated in order and yields the value of the last one. An example use is
>
> ```
> b = (temp = a, a = b, temp)
> ```
>
> which swaps the values of a and b and yields the resulting value of b. Needless to say, it is clearer to use a slightly longer form with explicit instructions.

C++ also supports a notion of *conditional expression*, of the form

```
x ? a : b
```

which yields the value of a if x, interpreted as a boolean, has value true (non-zero), and of b otherwise. (This means that ? … : can be considered a ternary operator, the only one.) A typical usage pattern is

```
template <typename G>
G max (G a, G b) { return a > b ? a : b; }
```

The parenthesis pair, (…), used in function calls such as f (a, b), is also considered an operator, the only one with a variable number of operands.

Programmers may overload all operators — to reuse them for their own functions — with the exception of the following four:

```
.    .*    ::    :?    sizeof
```

Identifiers

A C++ identifier is an arbitrarily long sequence of letters (including underscores) and digits, from which the first character should be a letter. This allows (unlike in Eiffel) the first character to be an underscore, but by convention this should be reserved for special compiler-managed variables.

Unlike Eiffel, C++ is case-sensitive.

There are no standard naming conventions in C++, so you can use the conventions of this book or others. Note, however, that STL class names are lower-case.

Literals

Literals (manifest constants) can represent integer, character, floating point or string constants.

Integer constants can be: decimal; octal if preceded with 0; hexadecimal if preceded with 0x. In the last two cases the starting character is the *digit* 0. So you can represent decimal 12 as any of

```
12          // Decimal
014         // Octal
0xC         // Hexadecimal
```

> Be careful not to start a decimal constant with 0: the constant written 012 will be interpreted as octal — its value is 10!

A character constant is enclosed in quotes, as in 'A'. A floating point constant consists of an integer part, a decimal point, a fraction part, and an optional integer exponent consisting of the e symbol followed by an optionally signed integer. By default a floating point constant has type double, unless it has suffix f (for float) or l (for long double). A string literal is a sequence of charaters, enclosed in double quotes.

Keywords

The following names are reserved in C++ for use as keywords:

asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, goto, if, inline, int, long, new, operator, private, protected, public, register, return, short, signed, sizeof, static, struct, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, volatile, while.

C.10 FURTHER READING

Reference manual by the language designer (anyone with a serious interest in C++ should read it):

Bjarne Stroustrup: *The C++ programming language*, 3rd edition, Addison-Wesley, 2000.

Introductory texts:

Herbert Schildt: *C++: A Beginner's Guide*, McGraw-Hill, 2003

Bruce Eckel: *Thinking in C++: Introduction to Standard C++*, Prentice Hall, 2000.

For advanced features, especially template-based programming:

Andrei Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001

David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*, Addison-Wesley, 2002.

David Abrahams and Aleksey Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*, Pearson, 2004.

D

From C++ to C

C++, studied in the previous chapter, is an extension of C, a rare example of a language that is almost fully *backward-compatible* with a predecessor, meaning that a valid C program is generally valid as a C++ program and produces the same results.

C++ has not replaced C. The earlier language retains its value for a well-defined application niche: programming directly at the level of the operating system or hardware. This is in part because hardly any processor exists without a C compiler. Or maybe the causal relationship is the other way around: no one dares to release a processor without a C compiler, because the market expects it. In any case C is the *de facto* standard for low-level programming.

Most C programmers also learn C++. For that reason, the present appendix does not describe C from the ground up: it assumes that you have read the previous one, and simply lists C++ constructs that are *not* available in C. This is quickly said, so the appendix is very short.

This mode of description (by difference with another language) also explains why, unlike in previous language appendices, the discussion of C's background and style only comes in the latter part of the discussion.

D.1 ABSENT ELEMENTS

From the description of C++, going down from C++ to C means dropping:

- All object-oriented mechanisms: classes and objects; function members (for structures); inheritance; virtual functions; constructors; destructors; references (but pointers remain).
- Templates (no genericity in C).
- Argument defaults. (Functions with variable arguments are available through a library mechanism, known as varargs.)
- Access control mechanisms and the friend mechanism; the scope resolution operator.
- Namespaces.
- Exceptions.

Most of the rest remains, including:

- Static functions.
- The instruction-expression merge.
- Pointers (not references) and the ability to manipulate them through arithmetic operations.
- The control structures, other than exceptions.
- Operators with side effects such as ++.
- The syntax conventions: braces rather than keywords, = for assignment, semicolons as terminators.
- The availability of a preprocessor, in particular compile-time variables allowing conditional compilation (#ifdef compile_time_variable, where the value of compile_time_variable is set outside of the program, for example through a compilation option).

D.2 LANGUAGE BACKGROUND AND STYLE

C came out of research at AT&T's Bell Laboratories in the late sixties, as a way to benefit from ideas of structured programming yet retain direct access to machine-level mechanisms. This latter requirement is not just a matter of performance; another reason is that the original application of C was to write an operating system, the first version of Unix. That project was a success, and all subsequent versions of Unix (and several other operating systems) have been written in C.

Crucial to this ability of C to handle low-level aspects of applications — that is to say, to work directly at the level of the operating system and the hardware — is the direct manipulation of addresses that we saw in the previous appendix, in particular the use of pointers (* and & operators), pointer arithmetic, and the idea that an assignment can have as its target any expression (l-value) that may denote an address.

The tradeoff here is that by gaining fine control of low-level resources you lose some of the benefits of *abstraction* as provided by more modern languages, in particular the full extent of type checking. Pointer arithmetic is a typical example of this situation: you can compute any address dynamically, but cannot be sure that what will appear at that address will, in every execution, be a meaningful value. This is not just a matter of program reliability, but also a security concern: *buffer overflow*, one of the favorite attack techniques for Internet intruders, fundamentally relies on C's mechanisms for accessing arbitrary memory addresses computed dynamically.

The tradeoff also sets the limit of reasonable C usage. Although C continues to be used for large systems, this is not its best use. Two valuable applications remain for C: short routines for direct resource access, and target language for portable compilers.

In the first role, C remains a tool for direct use by programmers. The observation is that the bulk of any application does not need the low-level aspects of C, and would suffer from them, for example by risking memory access errors at run time. Some specialized part of the application, however, may need direct interaction with the platform (hardware plus operating system). Programmers should provide these mechanisms in the form of clearly specified and carefully written functions, which typically will be in C, and just as typically should remain very short. An example is a routine to send information through a *socket* (abstract network connection). Such routines — usually no more than a few lines or a few dozen lines — should be grouped into a library and made available to the rest of the software through a well-designed API.

> The EiffelBase library relies on such an approach for the implementation of such abstractions as arrays and files. To the rest of the world, the corresponding classes are just normal Eiffel classes with contracts; their implementations simply call out to short external C functions.

In the second role, C serves as the target language produced by compilers for some programming language *PL* offering a higher level of abstraction than C. This technique presents significant advantages:

- The almost universal availability of C compilers facilitates the construction of *portable* compilers (where portability means the ability to support many platforms). The compiler for *PL* can concentrate on the platform-independent aspects of compilation of *PL* programs, relying on the C compiler to turn its output into machine code for specific platforms.

← *"Long-term product quality", page 708.*

- The generated C code can still include platform-dependent elements by relying on conditional compilation.
- C compilation technology is well understood; considerable *optimization* work, in particular, has gone into C compilers. Authors of *PL* compilers can concentrate on *PL*-specific optimizations, and (generally) rely on the C compiler to perform standard optimizations of low-level constructs such as arithmetic expressions which do not need to be re-implemented for every particular language.

This approach has been successfully used by a number of compilers, including Eiffel compilers.

Applications of C other than the two described here seem hard to justify given the limitations of C and the availability of many solutions devised in the forty years following the first introduction of this highly successful language.

D.3 FURTHER READING

Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, second edition, Prentice Hall, 1988. (First edition with same title, Prentice Hall, 1978, still available.)

> This is the bible of C programming, known as "K&R" from the authors' initials, notable for the clarity and conciseness of the presentation. There are many other books about C, but it seems hardly necessary to read anything other than K&R to master C. For C++, see the references at the end of the preceding appendix.

E

Using the EiffelStudio environment

Throughout this book you have been invited to write examples and run them using the EiffelStudio environment. The present appendix helps you prepare the examples and set up the execution. More precisely, you will find only some of the basic information here; for details, see the expanded version of this appendix online at touch.ethz.ch/eiffelstudio.

E.1 EIFFELSTUDIO BASICS

EiffelStudio is a general-purpose Integrated Development Environment (IDE); it is the result of many years of development and is routinely applied to produce software systems, small and large, in many application areas.

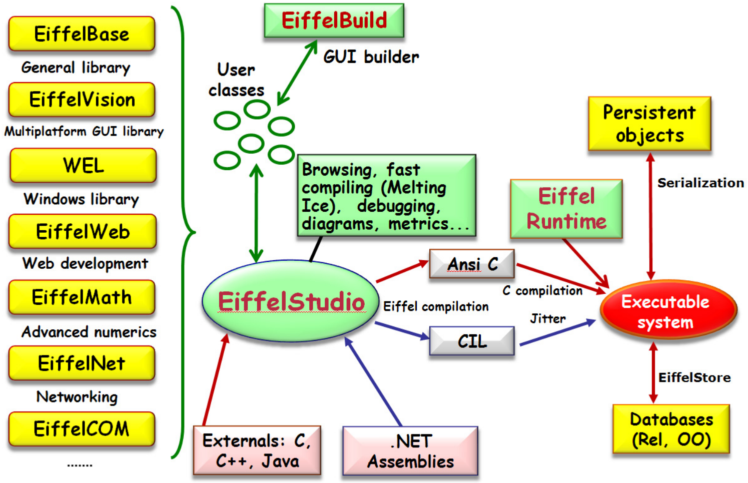
One of these systems, by the way, is EiffelStudio itself, written — just in case you are wondering — entirely in Eiffel with the exception of some C code making up the "run-time system". At the time of writing EiffelStudio includes about 2.2 million lines of code, written in accordance with the principles of this book. If you are curious to see that code, do not be shy: the whole environment is available as open source.

EiffelStudio has versions for numerous operating systems, including Windows, Linux and other variants of Unix. The details of installing and starting EiffelStudio will vary with each platform, and the "look-and-feel" (the graphical appearance) conforms to the standard conventions of every supported platform; but the environment otherwise works identically.

The explanations in this appendix are platform-independent, with the only exception that they use the term "directory" for what you may know as a "folder" if you are a Windows user.

The figure overleaf shows the essential components of EiffelStudio. It is reproduced from the discussion of IDEs and EiffelStudio in the chapter on tools. You do not need to have read those sections to start using EiffelStudio, but when you do get to that part of the book you will be able to form a better idea of why things are the way they are.

← *"An IDE: EiffelStudio", 12.11, page 353.*



EiffelStudio components

(*Figure from page 354.*)

E.2 SETTING UP A PROJECT

With EiffelStudio you build *systems*. A system (also called a “project” in EiffelStudio) is a collection of classes, grouped for convenience into *clusters*, with one of the classes serving as root; the root defines where execution starts.

← *“System execution”, 6.8, page 130.*

There are two ways to launch EiffelStudio to work on a system:

- To build a completely new system, launch EiffelStudio (typically by clicking the corresponding icon or menu entry among your installed programs, but this can also be done from a command line) and choose File → New project. A wizard will guide you through the (simple) process.
- If you start from an existing system, it will include an “ECF” (Eiffel Control File), automatically generated by the EiffelStudio session. You will see it as a file with the **ecf** extension; more precisely, the file name is *s*.**ecf** where *s* is the name of the system. Double-clicking this file (or using any other convention provided by your operating system) will start EiffelStudio on the corresponding project.

If you want to work on one of the examples of this book, you can use the second technique since the ECFs already exist. The conventions were given with our very first example: each example occupies a subdirectory of the directory *example* in the Traffic delivery. The name of each subdirectory is made up from the chapter number and the example name, for example *02_object* for the “*object*” system serving as the basis for chapter 2.

← *“A class text”, 2.1, page 15.*

To start working on one of these systems, double click the ECF in the appropriate directory. EiffelStudio starts, loading the system. The first thing you should then do is to click the "Compile" button to compile the system in its initial version, before you start modifying it.

E.3 BRINGING UP CLASSES AND VIEWS

To bring up a particular class in the top-left subwindow, you may either:

- Start typing the class name. If the class exists, you do not need to type the full name; EiffelStudio will complete it. Type Enter when the name is final. If no class exists with the given name, EiffelStudio understands that you want to create one, and brings up a simple wizard to help you do this.

- Use pick-and-drop. Right-clicking a name anywhere in the interface enables you to select the corresponding class (or feature, or other software element). You only need a click; do not maintain the button pressed. You can then move the cursor to a suitable "drop" place, and right-click again. If the drop place is, for example, the top-left window, it will retarget itself to show the text of the chosen element.

← "Browsing and documentation", , page 355.

A row of buttons enables you to change the display format by selecting an appropriate "view" such as Contract or Flat.

← Contract View: "What characterizes a metro line", page 53 ; Flat View: "The flat view", page 556.

You can explore many more views, as well as graphical representations, metrics, compiler information and other facilities by using the bottom-left subwindow and its various tabs.

Note that "bottom-left" and other positional descriptions only refer to the default layout. You may organize the various subwindows of your EiffelStudio workspace any way you like by dragging them around; the layout will be retained from one session to the next, although if you mess up you can always use View → Tools Layout → Restore Default Layout. You can also save your preferred layout into a file; this is useful in particular if you use EiffelStudio on different machines and do not want to repeat the setup process.

E.4 SPECIFYING A ROOT CLASS AND CREATION PROCEDURE

To specify or change the root class and root creation procedure of a system, choose File → Project Settings, then click **Target**:*s* where *s* is the name of your system. The Root entry lists the root in the format *root_class*.*root_procedure*; you can edit it, then recompile.

E.5 CONTRACT MONITORING

You can control the level of run-time monitoring of contract elements: preconditions, postconditions, class invariants, loop invariants, loop variants, and the **check** instructions. (The last construct, which we have not used, is a simple instruction stating that a property must hold at a certain point in a routine). The settings are separately applicable to each of these categories. To change them from the default, choose File → Project Settings, then **Target**:*s* where *s* is the name of your system, then **Assertions**.

E.6 CONTROLLING EXECUTION AND INSPECTING OBJECTS

To monitor the execution of your system, explore the run-time object structure, execute code step by step, set breakpoints, and even step backward, you may use the EiffelStudio debugger. You can find detailed information on how to use it at docs.eiffel.com/book/eiffelstudio/debugger.

E.7 PANIC MODE (NOT!)

Things should go smoothly, of course, but if at some point nothing suddenly seems to work any more, do not panic; remember the following.

If the problem is just that you do not find the tools to which you are used in the interface, restore the default layout or one you previously saved, as explained above; use View → Tools Layout.

If things really go bad and you find your system in a strange state, it may be that the project files have been corrupted. Then you may:

- Exit EiffelStudio.
- Restart it, but directly (not by clicking an ECF).
- Choose File → Open and select the ECF of your system.
- Check the box labeled **Clean**, then click **Open**. This will bring up the system but remove all the compilation files. You will have to recompile the system from scratch.

An equivalent technique is, outside of EiffelStudio and while it is not running, to remove the directory called EIFGENs (for Eiffel-Generated files) in your project directory. It only contains compiler-generated files, which will be re-created in the next compilation.

E.8 TO KNOW MORE

EiffelStudio is a rich environment with many facilities; this appendix only scratches the surface. You may find extensive information, including tutorials, manuals, videos and reference documents at docs.eiffel.com.

Picture credits

Punched cards, page 12: thanks to Immo Noack and Luca Previtali for helping to locate a surviving deck of punched cards.

Tunnel sign illustrations, page 27: Till Bay.

Buitoni Minestrone bag, page 142: with permission from Société des Produits Nestlé S.A., Vevey, Switzerland.

Tintin extract, page 87: *Explorers on the Moon* (American edition of *On a marché sur la Lune*), Litttle, Brown and co. 1976, © Hergé/Moulinsart 2008, with permission.

Portrait of Peter Naur (page 296): Painted by Duo Duo Zhuang (www.dodoz.com), with permission.

Laughing Cow, page 435: with permission from Fromageries Bel, www.bel-group.com. The laughing cow trademark is the property of Fromageries Bel.

Zurich map, page 460: Google maps, reprinted per permission policy at www.google.com/permissions/geoguidelines.html.

Teletype brochure cover, page 664: reprinted with permission of Alcatel-Lucent USA Inc.

Debugging cartoon, page 730: Tom Van Vleck.

Photographs of computer scientists:

Page 187 (Dijkstra): Hamilton Richards.

Page 318 (Chomsky): Prof. Chomsky's MIT office.

Page 295 (Backus): IBM Switzerland.

Page 325 (McCarthy): John McCarthy.

Page 328 (Dahl and Nygaard): Personal archive of Kristen Nygaard, courtesy of Prof. Marius Nygaard.

Page 329 (Goldberg): Adele Goldberg.

Page 494 (Naur & Horning): Jim Horning.

Page 695 (Reenskaug): Trygve Reenskaug.

Others photographs are by the author of the present book. (For more, see the "Gallery of Computer Scientists" at se.ethz.ch/~meyer/gallery.)

Miscellaneous illustrations: chip (page 9), by Michael J. Connors, www.mconnors.com; desktop computer system (page 9), by Quentin Houyoux; laptop (page 9), by Elvin Santana; GPS system (page 9), by Julien Gron; scroll (page 380), by Davide Guglielmo, www.broken-arts.com; memory chip (page 284), from Matrix Warehouse Computers, www.matrixonlineshop.co.za; diamond (page 704), by Tomislav Zivkovic.

Index

Page numbers in **boldface** indicate a page where the concept is defined.

In the electronic version, clicking a page number will take you to the corresponding occurrence in the text.