A DISCIPLINE OF JAVA PROGRAMMING

SIMONE SANTINI

Cuadernos de Apoyo 31

A DISCIPLINE OF JAVA PROGRAMMING

Servicio de Publicaciones de la Universidad Autónoma de Madrid

A DISCIPLINE OF JAVA PROGRAMMING

Simone Santini

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Servicio de Publicaciones de la Universidad Autónoma de Madrid

Todos los derechos reservados. De conformidad con lo dispuesto en la legislación vigente, podrán ser castigados con penas de multa y privación de libertad quienes reproduzcan o plagien, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte, sin la preceptiva autorización.

© Ediciones UAM, 2011

Diseño y maquetación: Miguel A. Tejedor López
Ediciones Universidad Autónoma de Madrid
Campus de Cantoblanco
C/ Einstein, 1
28049 Madrid
Tel. 914974233 (Fax 914975169)
http://www.uam.es/publicaciones
servicio.publicaciones@uam.es

ISBN: 978-84-8344-190-9
Depósito legal:
Printed in Spain - Impreso en España

e-ISBN: 978-84-8344-221-0

ÍNDICE

PREFACE 7

I. WHAT'S WRONG WITH THIS BOOK? 11

II. JAVA, SOUTH-EAST OF THE SILICON VALLEY 21

III. NOMINA NUDA TENEMUS 33

IV. OBJECT (ALMOST) ORIENTED 55

V. OBJECT-(NOT-QUITE)-ORIENTED DESIGN 69

VI. DESIGNING LIBRARIES 97

VII. THE CASE OF CONNECTIVITY 107

VIII. THE INTERNATIONAL IMPERATIVE 121

IX. A COUNTER-CULTURE MANIFESTO 145

NOTES 159

BIBLIOGRAPHY 175

PREFACE

In his delicious book *Diario Minimo*, Umberto Eco tells us that, in 1951, in the midst of the cold war and under the imminent threat of compete annihilation of civilization, Bertrand Russell collected all the volumes of the *Encyclopædia Britannica* in a zinc box enclosed it in a concrete block and submerged it in a lake (which I believe was Lockness) with the caption "Bertrandus Russel submersit anno homini MCMLI" for the benefit of a future civilization emerging from the ashes of the seemingly inevitable atomic war. A similar collection in Germany was introduced by the caption "tenebra appropinquante".

It would be an exageration to say that this book is my message to a future programming civilization that will emerge from the deep crisis in which programming is plunging happily into; it would also be an exageration to say that, to me, many programming "paradigms" that are proposed around, and a lot of the enthusiasm that surrounds current software development, look suspiciously like uncorking champagne on the Titanic. There is a little bit of truth in it, though. This book is, in a sense, a plea for formalization, for education, and for the intellectual and cultural excellence of programmers as the only possible way out of the programming crisis, a crisis that was declared for the first time about 40 years ago, and from which we haven't still been able to emerge. And, if you think that I am exagerating, a look at the data on the failure of software project that I report at the beginning of chapter I should be enough to make you reconsider.

This book is born out a personal insatisfaction. Years ago, I was leading a reasonably large Java development effort as part of a large, multi-university project in the US. Although the overall project was academic, the group I was leading was in charge of a strictly industrial type of development. All the programmers were hired away from companies, and the kind of product we were asked to deliver had to have industrial characteristics. I, myself, had been hired away from the indutry, having

previously worked for three years as a designer/analyst in a small internet start-up. Development went on slowly, painfully, with all the usual delays and frustrations that we are by now accustomed to. In the end, like the majority of development groups, we delivered our product late, with some bugs (that were discovered during its use, much to the chagrin of our customers) and without some of the features that we had promised. I left the group to take on a fully academic position shortly after the release of the first version of the product, but I stayed in contact with the group, and the news that arrive every now and then tell me that the situation is pretty much the same.

The guys I was working with were brilliant, and we were following all the collective wisdom available to us. That is when I started thinking that maybe the collective wisdon was wrong. I noticed, for instance, a lack of abstraction in my programmers that often surprised me, and that often prevented them from finding simple and elegant solutions, but I also found that this lack of abstraction was not only tolerated, but actually encouraged by the methodologies we were using. That, I was worried about. I came to realize that the excessive reliance on superficial metaphors at the expense of abstraction, the insufficient "instinct" for formalization, the excessive reliance on simple (and often irrelevant) "to do" lists, the excessive faith in the use of libraries (often of the same poor quality we were plagued with) were all important components of the problem. (And yet, in spite of everything, this book does contain a lot of "to do" lists.)

This book is my contribution to the idea that the problems that plague software design are not technical but, in large part, human. We are simply training programmers the wrong way, not giving them a mathematical enough mind-set, not giving them enough linguistic education (the importance of language for programming is one of the main threads that go thorough this book). We train programmer the way we would train technician, forgeting that a programmer is not a technician, but a creative mathematician.

This book is the result of many personal experiences and much reading. Too much to recognize all the influences one by one. Among them, however, I would like to single two out. The first is Strunk and White's book *The elements of style*, possibly the best book on the subject ever written. Fittingly enough, I like it especially because of its style. It is simple, direct, written in a no-nonsense way. And it is short: maybe a fifth of the length of the majority of style books. Programming is a disciplne plagued by enormous books that say nothing. I have in mind, for in-

stance, a book that I shall not mention on the *Java data base connectivity interface* which totals a whopping 700 pages. Well, you know, there simply *isn't* that much to say about the connectivity interface. In this scenario, the short, essential nature of a book like Shrunk and White's looked extremely appealing to me, and I tried (unsuccessfully, I am afraid) to imitate their style. If nothing else, I did manage to keep my book short.

The second influence, acknowledged by the paraphrasis of its title, is E.W. Dijkstra's book *A discipline of programming*, one of the most lucid statements on programming produced to-date. I am not sure Prof. Dijkstra would approve this book for its unacceptable concessions to lack of formality and to the "tenebra appropinquante", and I am afraid that, if he could express his criticism, I would have to agree with him. I do hope, however, that he would in some measure approve the direction in which I am trying to steer a programming area plagued by the same problems that he was trying to address, 36 years ago, with his book.

Simone Santini
Madrid, June 4, 2010

I. WHAT'S WRONG WITH THIS BOOK?

Even the most casual and distracted reader (the one who couldn't remember the book she read last week even if it were "Moby Dick" and she were reading it for the fifth time) will have noticed a couple of unusual things about this book, just by looking at it in the bookstore, without even taking the trouble of picking it up from the shelf: firstly, it has less (considerably less, in fact) than the 1,000 pages that seem to have become the *de facto* standard for any book related in any way to programming; secondly, on the cover of this book I haven't called my reader an idiot, a dummy, or any other similarly appreciative epithet with which many authors of programming books address their audience. I haven't even, come think of it, tried to appeal to whatever sense of religious awe you might have by calling this book a *Bible*.

There are two very good reasons for this lack of insulting epithets. The first, quite pragmatic, is that all the good ones are taken, the publishers of the series that use them would probably sue me if I used one of theirs, and I haven't got the money to resist a lawsuit from a powerful publishing house. The second, more substantial, is that I do not consider my readers either idiots or dummies. (If you think you are, please return the book immediately: you might still qualify for a refund.) My reader is a very intelligent person, curious about programming and its principles and, quite likely, a programmer herself, a student of programming, or a person involved in some capacity in the production of software.

Programmers and software designers, like everybody else, have habits and, like everybody else, they are sometimes "married" to their habits or, at least, are quite comfortable with them and have a strong and understandable propensity not to change them. When you are writing a book in which you are trying to give

guidelines to people, this propensity places you in a bit of a conundrum. On the one hand, if the guidelines that you give are in complete agreement with what your readers are already doing, then the book is quite useless: reading it might reinforce the ego of your reader, but it will be essentially a waste of time, as nothing new will be acquired from it. On the other hand, if you propose guidelines completely alien to your reader's programming habits you risk rejection, and in this case too the book would be useless, not having managed to obtain any durable effect. The narrow line between these two undesirable alternatives can be negotiated trying to be rational. You and I, I mean. As my contribution to the mutual rationality, I will make every effort to justify the guidelines that I give on purely rational ground, trying to give good arguments on why I consider the programming practices that I am proposing better than the alternative, and this especially when I will give guidelines that go a bit off the beaten path, and are likely to be received with a healthy dose of skepticism. In exchange, I expect the reader to examine the discipline of style proposed here open-mindedly and rationally, especially those guidelines that might go against his programming habits. Should a guideline be rejected, I expect that my reader will have valid rational reasons for doing so.

There is a rejection argument that I wouldn't accept as rational: the "this is the industry standard" argument. I know that the industry likes to have things done in a certain way, and that it puts a lot of pressure on programmers to convince them that the industry way is "the" way of getting the job done and that everything else is futile academic exercise. But standards are not laws of nature: they are human creations, and people can, and in certain cases should, change them. More than 20% of the software projects that the industry starts are abandoned before completion, and 45% are released late, over budget, and without all the requested features[1]. If you think that this is not disastrous or that it is a natural consequence of the complexity of the software artifacts, compare the software industry with another industry that builds things even more complicated than programs: the airline industry. Whenever Boeing or Airbus start the design of a new airplane, you can be quite sure that the project will be completed, and the airplane built. Oh, sure, the odd project here and there might fail, but a 20% failure rate would drive the company out of business in a very short time. When a project is seriously delayed, the consequences are dire: Airbus lost its advantage over Boeing (temporarily, at least) and went into serious financial trouble when deliveries of the A380 were delayed. (And we are talking here of the largest commercial jet in the world, a project on a scale never attempted before, hardly comparable to the relatively humdrum nature of most software projects.) If the software industry were to suffer the same consequences for a delay,

most of the software producers would have long been out of business. By all industrial standards, the performance of the software industry is unacceptable. Given this situation, one would think that the software industry has very little ground for defending the validity of its way of doing things, let alone to propose as the only correct way of doing things. All in all, they should welcome any proposal for a change as a potential life saver, knowing that change can't possibly make things worse than they already are.

* * *

This book is about *style*, a problematic concept when applied to mathematics, even to that part of it known as programming. We all might agree, more or less, on what we mean by literary style but what on God's green earth is programming style?

With a certain degree of approximation one could say that style is whatever distinguishes two programs that work correctly. That the program works correctly, that it satisfies the syntactic constraints of the programming language and the semantic, model-theoretic, requirements of the formal specification is, of course, a pre-requisite: leaving out the semicolon at the end of a statement or using a variable before declaring it is not a matter of style but simply a programming error. This leads to a natural objection: if two programs work correctly, why should we care about the difference between them? This is only a polite way to ask: who cares about style? We should try answering this question before we spend the time and energy to write a book about style (I) or to read it (you).

Who cares about style?

Indeed: once a program works, shouldn't this be the end of it? That one wrote it this way or that way doesn't really matter: if it works it works, and that's the end of it. This might *maybe* be true if one wrote a program in a relatively trivial domain and if the program was, furthermore, intended to be used only once (or, at most, just a few times in rapid sequence) and then destroyed. There are some programs with this profile, and for them one might have half of a point in defending the use of a

"quick and dirty" programming style but, clearly, the vast majority of the programs that are being written are of a very different nature. If the program has a relatively long life, it ceases to be primarily the implementation of a formal algorithm on a given computer and becomes primarily an instrument of communication between people, more specifically between programmers. The distinction is important, and we should spend some time on it.

The idea that the purpose of a program is primarily to be executed on a computer is to a certain degree natural, but it is also terribly misleading. Once the program has been prepared and proved correct, the cost associated to its translation into executable code and its actual execution is negligible. As the industry knows very well, the main costs associated with the life of a program are its correctness and its *maintenance*, that is, everything that is done to the program after its initial release. These activities hinge on two kinds of formal/linguistic actions: the translation of the description of a problem from the informal spoken language of a programmer to the mathematical one of the programming formalism, and the communication of the formal solution to other programmers (or to the same programmer after some time that he is not looking at the problem).

* * *

The first point will be the subject of a more extensive discussion in the following chapters. At the beginning of every programming activity there is a problem to be solved, described and understood using natural language. This informal description generates two different kinds of artifacts. The first, classically, is the formal requirements document, that is, a formal specification of the external behavior that the program must exhibit in order to satisfy the customer (which may be the same organization where the programmer works, or even the programmer himself). The formal requirement is, of course, fundamental because it acts like a binding contract between the programming team and the customer: the programmers can use compliance with the formal requirements as a proof that they have fulfilled their contractual obligations, and the customer can use lack of compliance to show that they haven't. There is no excuse for the incompleteness or the insufficient formality of requirements, which should be taken with the utmost seriousness by any programmer: without them it might be impossible to show that you have fulfilled your obligations, and with some clients you might have to prove it in a court of law

in order to be paid. Look with suspicion at people who tell you that formal requirements are a waste of time, that they are an abstraction, that they are too static and that "in this fast-paced world" requirements change. Your lawyer might soon be talking to their lawyer.

The second artifact that originates from the informal description, via a separate formalization, is the structure of the program. The requirements tell the programmer how the program should behave *vis à vis* its environment (that is, it defines the interface of the program) but tell nothing of what the internal organization and structure of the program should be. This structural formalization must be derived, through an independent formalization process, from the description of the problem. While the requirements are a contractual obligation, the translation of the description into a structure is a matter of choice: no contract is forcing the programmer to devise a *good* structure for the program; the requirements document imposes merely that the program perform certain functions. Winnowing out the proper structure is a matter of style. As we shall see, in the context of object oriented design the proper structure is one that is isomorphic and functionally homomorphic to the problem that one is trying to solve and to the domain where the problem appears. In object orientation, the particularities and idiosyncrasies of the solution found by the programmer should be buried, so to speak, deep within the class structure, in the algorithms that compose its methods. Two benefits are claimed for this way of structuring: on the one hand, the program is likely to be simpler and, therefore, smaller, faster, and easier to prove correct; on the other hand, the isomorphism with the problem domain will facilitate the second activity we considered above: the communication between programmers.

Perhaps the most important function of a program is to be read by programmers. A program is essentially a mathematical theorem: it is the proof that a solution to the given requirements exists. With respect to the theorems of other branches of mathematics the program has one useful quirk: once executed on a (virtual) machine, it actually finds solutions to instances of the problem. In this sense, a program is a proof of existence with a vengeance: it is a proof of the existence of a solution that can be used to find the solution. This charges the programmer with certain burdens with respect to other mathematicians. For one thing, the proofs found by other mathematicians often have informal passages, or parts described in natural language; a programmer's proof must be completely formal, since it has to be executed by a machine. Programmers also don't have access to the whole array of techniques used by other mathematicians. For example, in general they can't use

the *reductio ad absurdum*, since this proof technique doesn't lead to the construction of the solution whose existence it proves. On the other hand, programmers make extensive use of mathematical induction, in the form of recursion. The sociology of mathematics imposes that, in order to be considered valid, a proof must be understood (potentially, if not in actuality) by other mathematicians, at least by those with sufficient preparation in the area of the proof. A proof that is not understood by anybody will be considered gibberish, not published and, to all effects, will never be part of mathematics. Programs are subject to a similar constraint in the sociology of programming: they don't live in a one-person environment: groups of programmers develop them and, in this process, they communicate the developing program to each other. Once a program has been completed it is likely that afterwards it will be modified, that is, that some of its functions will be used to solve similar problems, very much the way a lemma can be used in a theorem different from the one for which it was originally proven. Even if the program is developed by a single person, the communication problem is essentially the same: people are seldom able to keep all the details of a complex program in their mind at the same time; they need to read and understand the things they have already done. The industry estimates that most of the cost of development of a program is related to the communication between programmers and to *maintenance*, that is, to the operations that are performed on a program after its original release[2]. These activities have nothing to do with the program as an executable artifact (viz. with its correctness); their feasibility and their cost depend on the clarity of the program as an instrument of communication between programmers: they have to do with *style*.

Those of us who have some experience in academic environments have observed that a program written by a student is generally longer and harder to understand than a semantically equivalent program written by a good, experienced programmer. The difference is style. We have already observed that the stylistic basis of object orientation is an isomorphism between the structure of the problem and that of the program that solves it, and we have mentioned how this isomorphism may help translating the informal description of a problem into a formal structure. The same isomorphism can help communication. While the specific program is the creation of a programmer at a given moment, the problem is common to all programmers that work in the area, and doesn't change with time. So, a structure isomorphic to the problem constitutes a of shared code, common among programmers and resilient, which lays the basis for communication. As a matter of fact, many student programs are difficult to read precisely because of the use of structures and data that, while at the time made sense in the head of the student, do not match very well the structure

of the problem. The reader of the program knows the structure of the problem but, alas, he doesn't know what was inside the head of the student. Trying to read these programs is like trying to read text in a foreign language that we don't know and that uses concepts alien to us: we have to get through the language barrier before we can even start to understand the text; it is like reading a secret message encrypted using a cipher for which we don't have the key.

The crucial importance of style becomes evident once we recognize the true nature of programs as instruments of formal communication among programmers, and we pull away from the apparently prominent but ultimately misleading aspect of programs as executable artifacts.

Where style comes from

Can one teach style? Well, yes and no and yes and no. It depends on what kind of teaching one has in mind and at what conceptual level one is considering style.

Let us start with the different aspects of style. At the simplest level, we have what we could call the *syntactic/local* issues: finding good names for variables and functions, writing the code in a way that is neither too compact (code in which many instructions are written one after another on the same line is harder to understand) nor too spread out (functions that don't fit in a single screen/page are harder to grasp), and so on. These aspects are important, and it is thanks to them that the average, reasonably written program is always easier to read than the year's winner of the obfuscated C code award[3], but creating a truly well written program, one that constitutes a good communication instrument between programmers, requires more than just that. It requires, as we have briefly seen in the previous section, a proper structural design. This is a much more complex and, in a sense, ill-defined activity: while it is relatively easy to give *a priori* advice on how to avoid common syntactical/local pitfalls, an adequate structure only reveals itself *post facto*: everybody will be able to tell you whether a given structure is good or bad, but spelling out what makes a good structure is quite a different matter. Much like St. Augustine said about the nature of time: "if you don't ask me, I know, if you ask me, I don't".

This leads to the second part of the answer to the question that was posed at the beginning of the section: whether or not style can be taught depends a lot on what form of teaching one is willing to employ. A list of guidelines, put into effect

with good judgement, can be of considerable help to improve one's syntactical/ local style, but no manual (not even this one, much to my dismay) can *teach* you to derive a proper structure of a problem: the most I can hope is to give you valuable instruments to recognize good structure, and therefore to correct your designs towards better structuring. But to devise a good structure in the first place is a matter (in part) of intuition and (mainly) of mathematical and linguistic culture. Ennio Morricone said, about his famous film soundtracks, that they were "1% inspiration, 99% perspiration". He forgot to mention that he was in possession of an enormous culture, both musical and general, and that this culture helped him immensely when it was time to find the right harmony and the right arrangement for a given situation. The same is true for a programmer with respect to programming culture and to general culture. The last chapter of this book will discuss more in depth the importance of programming culture (which is a particularly rigorous form of mathematical culture) for the acquisition of a good programming style, but it is worth stating out front that there are no short-cuts, no list of "do do the do's and don't do the dont's" that can quickly instill a correct programming style in anybody. A programmer that goes about designing the structure of a program is like a mathematician that goes about proving a theorem: there are many proof techniques that can be chosen, many intermediate lemmas that can be built as stepping-stones, many theorems in the mathematical literature that can be used. Knowledge of mathematics makes all these instruments available to the mathematician; mathematical *culture* allows him to pick the right ones and to organize them properly to create a beautiful and elegant proof. George Gamow wrote that no physical theory can be true that is not beautiful. The same is true for mathematical proofs, and for that particular type of mathematical proofs known as programs.

Knowledge of proof techniques and mathematical literature is a *conditio sine qua non* for the creation of beautiful proofs, but it is not enough: it takes mathematical culture, a correct mathematical *modus operandi* to do it. Knowledge of programming languages, techniques, and libraries is a *conditio sine qua non* for the creation of beautiful programs, but it is not enough: it takes programming culture to do it. Culture, in the sense in which I am using the term here only tenuously related to the accumulation of little tidbits of know-how. It is the kind of knowledge that permeates through contact with an intellectually stimulating environment; it is, as the Italian refrain goes whatever is left when you have forgotten everything you had learnt.

We can make a parallel with writing: in order to write you have, of course, to know grammar, but you don't learn to write by reading grammar books or writ-

ing manuals; you learn to write by reading good literature and by being immersed in a literary environment. A complete approach to style, in other words, requires much more than a list of guidelines; it requires a new approach to the education of programmers and to the development of an intellectually broad and stimulating programming milieu. I will return to this important point in the last chapter.

II. JAVA, SOUTH-EAST OF THE SILICON VALLEY

When, in May 1995, SUN® Microsystems released the Java programming language, many people (including, admittedly, myself) were amused but not overly impressed by the introduction: Java was a nice language but, at first, its main advertised use was the relatively easy creation of little applications (yclept, with a dubious etymological choice, harbinger of things to come, *applets*) that could be executed "inside" a web browser. The idea seemed nice and the language appeared reasonably clean and useful, but nothing to write home about. When, in late 1995, I used Java to create a rather complex interface for an image database that I was developing at the time—including a graph editor for the creation of a query workflow, and a nice image browser[1], for a total of maybe 5,000 lines of code—I was convinced that it would be one of the largest Java programs that I would ever write. I was, of course, wrong. By more than two orders of magnitude.

Universities started using Java in programming language courses with surprising rapidity: if memory serves me right, already in the fall 1996 semester the University of California, San Diego started using Java in lieu of C and C++ (or in addition to, soon to be in lieu of) for its introductory programming classes. I believe that many teachers at many universities were attracted to Java by its relative simplicity and lack of trickery with respect to C++ (which is, of course, a good thing to be attracted to), by the ease with which they could include graphical user interfaces and other flashy displays into programming assignments (which is, of course, a bad thing to be attracted to), and by the clout generated by the appearance of the language in industrial environments (which is, of course, just about the worst possible thing to be attracted to). It is true that the principles of programming languages would have been better introduced with the use of a functional programming language or, even better, with a formal, semantically unambiguous notation independent of any pro-

gramming language but, by 1996, the industrial establishment with its "real world" anti-intellectualist banner and resume-oriented curricula was already well on its way to take over academia. Education as an avenue for a rich intellectual life and knowledge as a goal rather than a tool were already looking like lost causes and, in this situation, Java is not the worst thing that could have happened to computing education.

* * *

In just a few years after its introduction, the Java programming language became dominant in internet based applications. Most commercial internet services today have a significant portion of their software written in Java. The language also extended to other industries that, although not directly connected to the internet, want a share of the lucrative internet market. An example is the data base Oracle: although, to the best of my knowledge, the data base is not written in Java, Oracle uses Java as one of the recommended languages for the development of user defined data types, preferring it to its native C and C++; Oracle even contains a specially designed version of the interpreter of the Java object code (the *Java virtual machine*). In some cases, I suspect, the expansion of Java into areas other than the internet has been facilitated by the academic endorsement of the language: some companies might find it easier to convert their development to Java rather than to find newly graduated engineers proficient in other languages (*rara avis*, these days) or (god forbid!) to teach a programming language to their newly hired engineers.

By now, the first generation of programmers educated using Java is employed (recession permitting) and productive[2]. Many of us have had the opportunity to work with or supervise Java programmers, and this might be a good time to start assessing the effects of the programming habits and design practices—in a word, of the Java *culture*—on the programming profession. While a lot of what I call the Java culture is only contingently related to the characteristics of the language, being more a consequence of the environment in which the language is used and in which the Java programmer works, it is not a bad idea to start any discussion about the Java culture with a brief assessment of the language itself, which is what I intend to do in this chapter.

As programming languages go, you could do a lot worse than Java. It is, all in all, a reasonably clean, simple, solidly traditional programming language, without frills, without too many aspirations, and without many of those short-cuts and tricks that for some reason programmers like very much but that often result in code that is hard to understand and even harder to modify. Even in 1995, when it was introduced, Java was by no means a revolutionary language: as a matter of fact, one would have been hard pressed to call it *modern* at all. Virtually all its major features had been implemented before. Rather, Java was a simple, judicious assembly of well known and solid characteristics. This simplicity, as well as the introduction, together with the language, of libraries to make Java program work in a peculiar and, by that time, interesting environment (the internet and the web browsers) was a major factor in its success.

When I started using Java I thought I would miss the flamboyant possibilities offered, in C and C++, by the conjunction of pointer arithmetic and type casting but, with surprisingly few exceptions, I didn't. Sure, there were cases in which I had to write six or seven lines of code to do something that in C I could have done with a single well placed pointer trick, but that trick would have been so obscure that its replacement with a longer (but clearer) version wouldn't be such a bad idea in any case[3]. Java inherited from Pascal the idea of an abstract intermediate language in which the source code would be compiled but that would itself be interpreted by a "virtual machine" (my Alma *mater*, UCSD had a very good implementation of the idea in what was known as the UCSD Pascal), the idea of interfaces from languages such as Modula[4]. In some cases the hallmark of Java is simplification: using interfaces and single inheritance simplified considerably the semantic problems caused by multiple inheritance[5].

One of the first characteristics that the typical C programmer will notice about Java—sometimes with infuriating results or even with a complete rejection of the language—is the lack of pointer data types. Of course, Java does use pointers: all the objects that passed as parameters and that are returned as results of functions are in reality pointers to objects. The question is, however, that in the semantics of Java, these pointers are not data types distinguishable from objects, and behave exactly like objects. Consequently, Java doesn't define the concept of a generic pointer to a location of memory, within which one can do whatever he pleases. Needless to say, Java doesn't allow pointer arithmetic, and makes no identification between arrays and pointers. One might say, with some claim to truth, that Java has gone a bit too far in its idea of hiding the presence of pointers. In order to avoid at

the same time the explicit introduction of pointer and the penalty of copying large data structures when they are passed as parameters to, and returned as results from, a function, Java had to muddle its semantics by stipulating a mixed form of parameter passing: scalar data types, such as integers or float, are passed by value, while arrays and objects are passed by reference (that is, in practice, using a pointer). The consequences are sometimes curious. Consider a function[6] such as

```
int f(int i) {
    i = i*2;
    return i;
}
```

an object

```
class exclass {
    public int i;
};
```

and a function

```
exclass g(exclass q) {
    q.i = q.i*2;
    return q;
}
```

then if one executes the following piece of code

```
exclass c1, c2, d1, d2;
c1 = new exclass();
d1 = new exclass();
d2 = new exclass();
c1.i = 1; d1.i = 1;
d2.i = f(d1.i);
c2 = g(c1);

println(d1.i, d2.i);
println(c1.i, c2.i);
```

one obtains the result:

```
1    2
2    2
```

that is, in the case of the object, the parameter sent to the function has been changed, even if the object consisted of a simple integer. If the operation is done by passing the only element of the object to the function `f`, the parameter passed to the function is not changed. Because of the desire to avoid pointers at all costs, the parameter passing semantics of Java is not very clean. In hindsight, it might have been better to adopt a solution in the style of Pascal, in which one could have explicit pointers to data, although their manipulation was not permitted, something akin to the *reference type* of C++[7].

It is fair to say that, at the time of the introduction of Java, when C and C++ had a preponderant position in the industrial programming practice, this is the feature that caused the greater number of eyebrows to raise, that caused the greatest debate and, quite likely, the greatest number of early rejections of the language. The idea that a block of memory could be allocated *qua* a block of memory, and then transformed into any other data type just by casting it, was so ingrained in the *modus operandi* of the typical C programmer (myself included) that, at first, it was hard to believe that one could live without it. FORTRAN programmers must have felt something similar when they were told that they would have to live without the *goto* statement and, in both cases, it turned out that we could live very well, and with better clarity of code, without either thing. The very liberal attitude of C towards memory allocation made program verification almost impossible, and it is fair to say that the vast majority of "mysterious bugs"—those over which one spends a week because the point at which the error occurs is 2,000 lines of code lexically, and 300,000 instruction temporally from the source of the problem—were due to pointers. Eliminating pointers meant giving up the possibility of playing certain "tricks" with the language (and also, in some cases, giving up a certain elegance) in exchange for a code clearer and easier to verify. Not a bad bargain, all in all. Much like television, pointers might at first seem impossible to live without but, like television, one finds out that, once you give them up, life becomes suddenly better.

* * *

I like the decision of the Java designers not to allow operator overloading, another thing that C++ allows: the '+' operator, for instance, means arithmetic sum if applied to integers or other numbers, but one can re-define it for other classes defined in a program. The prototypical example is that of re-defining '+' to be the composition operator of any collection type endowed with a monoid structure. So, if A and B are sets, $A+B$ is their union, if C and D are bags, then $C+D$ is their bag union, and if L and G are lists, $L+G$ is their concatenation. Outside of these textbook examples, I still have to see an application in which overloading operators leads to clearer and more legible code than defining a function to do the same thing. The only advantages of infix operators I can think is that they make certain equational properties (e.g. the associative law) typographically clearer than a functional notation. That is, if A and B are sets, the equation

$$A + (B + C) = (A + B) + C$$

is clearer than its functional counterpart:

$$union(A, union(B, C)) = union(union(A, B), C)$$

but, unless this kind of things is asserted in a program (which in Java you can't do anyway), the advantage doesn't amount to much.

Quite on the contrary, the use of operators can be deceiving because different operations represented by the same symbol are not quite the same. Consider again the textbook example in which A and B are sets, C and D bags, and L and G lists. Then $A+B$ is idempotent and commutative (viz. $A+A = A$ and $A+B = B+A$); $B+C$ is commutative but not idempotent, and $L+G$ is neither commutative nor idempotent. Operator overloading led us to a single, syntactically indistinguishable form to represent three operations with very different semantics. A very naughty thing to do. This is, of course, a considerable source of confusion, not worth the small advantage that one can hope to derive from overloading operators.

* * *

Java also dispenses with one of the features of C that created the most violently ambivalent feelings among programmers: its very sophisticated pre-compiler

that processes commands such as `#define, #ifdef`, and the like. The possibilities offered by the C pre-compiler are remarkable, but they come at a considerable cost in terms of program legibility and verifiability. Macros can be nested, and their text replacement capability allows almost endless possibilities of substitution. Moreover, the pre-compiler is a general purpose text-replacement system, completely independent of the C syntax: it takes a source file and replaces text as directed by the macros, without any limit or syntactic constraint. It is possible, for instance, to pass a keyword of the language as a parameter to a macro and have the pre-compiler introduce it into the text of the program. Only the final product of the pre-compiler, viz. the file that will be sent to the compiler, is required to comply with the C syntax: the source file and the intermediate results can be pretty much anything. All this makes the goal of understanding what the program does just by looking at the text almost impossible: each macro introduced in the body of the program can be expanded into dozens of other macros defined in dozens of other files. The code can be fragmented every which way, and put together only at the end of pre-compilation. A lexical nightmare. It is fair to say that nothing resembling the *obfuscated C code*[8] competition would exist had C not contained a pre-compiler, or had it contained a less sophisticated one. To this we should add that many symbolic debuggers do not expand macros, so understanding what happens when one of them is encountered in the code is a very complicated matter.

The designers of Java, quite wisely, decided not to have such a pre-compiler for the expansion of macros. Their decision was supported, I believe, by the consideration that the most common and useful uses of macros in C are the adaptation of the code to work on different operating systems and the definition of numerical constants[9]. In the case of Java, adaptation to an operating system is not necessary because, due to its interpreted intermediate code, all libraries can be used on any operating system that implements the Java virtual machine. This eliminates a problem common in C and C++ that occurs when a set of functions necessary to a program (say those found in the *sockets* library) have different implementations and different interfaces on different operating systems.

Constants, in Java, can't be defined outside of classes. This decision might be debatable but, once one decides to go with it, the static variables mechanism provided by Java is more than adequate for most things. All in all, therefore, Java did not need a pre-compiler of the sophistication of that of C, and giving it up increased the legibility and verifiability of the language.

There are a few cases in which the designers of Java could have made a more careful choice. An example is the decision to use the same cryptic form to express the *for* loop that C uses. A *for* loop in C is quite different from the numeric iterator that it was originally supposed to be and whose *classical* form, so to speak, includes a counting variable, a start value and a final value[10]:

```
for i=0 to n do
    X;
od
```

(or some other syntax expressing the same construct). The importance of this loop construction for program verification is that it is always guaranteed to terminate: using only *for* loops gives a program the expressive power of bounded recursion, which is not as expressive as general recursion (equivalent to Turing computability) but for which termination is decidable. General recursion is implemented using the *while* statement:

```
while cond do
    X;
od
```

which is more expressive, but for which termination is undecidable. Of course, the *for* loop is subsumed by this more general construct, but the advantage of having a distinct syntactic form is that it keeps the two kinds of recursion separated, making program verification easier since, for instance, any portion of program not containing *while* statements (or the equivalent *do...until*) termination is *ipso facto* proven. The *for* statement of C, on the other hand, is equivalent to general recursion, since the statement

```
for(A; B; C) {
    X;
}
```

is equivalent to

```
A;
while B do
    X;
    C;
od
```

Only in special cases such as

```
for(i=0; i<n; i++) {
    X;
}
```

can termination be proved. In other words, some very important properties of the statement depend not on its general semantics, but on the specific form that it assumes. Given that there is an equivalent way of expressing the *for* statement of C using *while*, it would have been better to return to the simpler form. Not to mention, of course, that the C *for* is extremely cryptic and, in some cases, very hard to read.

In this respect, however, the Java designers had at least the good idea of not identifying the boolean data type with the integer, and rejected the stipulation that a (integer) 0 value means *false* while a non-zero value means *true*. In Java, conditions have to be expressions that evaluate to *boolean*. This avoids the possibility of creating statements such as the typical C example

```
for(; *a++ = *b++; )
    ;
```

that, although quite economic, is very hard to understand: I wonder how many programmers without a specific knowledge of C would understand that this statement copies the string b into the string a. In Java, such a statement would be impossible because the assignment of the value 0 (and its return by the assign statement) can't be identified with the value *false*. This greatest clarity mitigates somewhat the obscurity of the *for* statement inherited from C. Still, it would have been much

better to keep finite recursion separated from general recursion by defining a limited form of the *for* statement.

* * *

I don't very much like the decision to allow only one public class per file (and, much less, the constraint that the name of the class has to be the same as the name of the file). The organization of a program into files is often an issue of development management, and not of program design. The file constraint mixes the two aspects in an unreasonable way. I approve the decision of getting rid of "header" files that contain the class definition but not its implementation. From the point of view of programming, the lack of header makes a certain sense and has no negative consequence since the *import* statement makes the textual inclusion of the header (as it is done in C++) superfluous. When I started using Java I craved the possibility of writing header files but, quite honestly, right now I am pretty happy without them. The limitation to one public class per file can be at times infuriating but, all in all, I became quite accustomed to that too, and never had problems adapting to it. A more seriously annoying limitation of Java, maybe a consequence of the identification of classes with files, is the impossibility of defining free functions: routines that are not part of any object or class. This results in the creation of many useless classes with a lot of static methods (*vide* the *Math* class in the basic Java library) that muddle the program structure needlessly. Additional lack of flexibility comes from the impossibility of defining *friend* classes and functions: in Java, the only way in which a class *B* can access the private methods and attributes of a class *A* is if *B* inherits from *A*; quite a hefty price to pay to have a look at *A*'s privates.

* * *

It should be clear that most of these flaws are rather venial ones and, compared to the languages of its class, Java is a fairly acceptable one. Java is not truly an object oriented language (the way Smalltalk is, for example) because it implements only part of the object oriented model, the most conservative and least interesting part, in a sense (for example, in Java you can't create an object that doesn't belong

to any previously defined class, and classes can't be defined dynamically during execution). Much like C++ before it—of which it is essentially a simplified version—Java is a structured language with some object oriented ideas sprinkled on top. It is relatively strongly data typed, and this fact, by itself, makes it *not* truly object oriented; you may like strong data typing and you may like object orientation but, in the end, you have to choose or compromise between the two: you can't have both. True object orientation is the antithesis of strong data typing. Java represents a rather typical attempt to incorporate some object oriented ideas in a traditional structured language without giving away too much in the way of strong data typing; in this sense, it keeps closer to its structured origins than it ventures into uncharted object oriented territory. If you try to add methods and inheritance to Pascal structures, you will end up with something reasonably similar o Java and, all in all, Pascal or Modula would have been the best possible starting points for the creation of Java.

Java didn't introduce anything new in programming languages and, even in 1995, one couldn't quite call it a *state of the art* language: Java doesn't have the functional purity of Haskell[11], the outstanding type system of Opal 2α[12], the object orientation of Smalltalk[13], or even the solid mix of modern ideas of ML[14]. Its success is due mainly to the environment in which it was proposed, and to the promises of facilitating internet programming that it made. It is probably not too far from the truth to say that any reasonable language that, in 1995, promised the possibility of developing small applications that were executed inside a browser, and offered decent libraries to connect to other computers on the internet would have had the same success. The final judgement of Java is a cautious approval. One the one hand, it can be regarded as a missed opportunity: it would have been nice if such an extraordinary success had blessed a truly modern language. Programming languages in 1995 were already much more advanced than Java, and Java incorporated none of the new ideas that had emerged in the previous decade; as a programming language, it could have been designed in 1985[15]. On the other hand, things could have been worse. Suffice it to say that at the time Microsoft[16] was proposing BASIC, of all things, as a viable programming language.

Many of the problems of Java derive, ironically, from its bizarrely quick and unreasonably broad success. Many aspects of the language betray an original design oriented towards "applets" or, in any case, towards small applications. I doubt that its designers originally thought of Java as a language for large server applications. The virtual machine, its inefficiency and flexibility, for example, make per-

fect sense for small, quick applications that must run in the unknown environment of an alien browser; they make much less sense for large applications that must run efficiently on a server. Java has never been a good language for large projects and large projects have been, in a sense, forced upon it. The subsequent versions of the compiler have added new libraries to Java's environment, but they haven't changed the language, which continues to make it somewhat difficult to design and manage large application programs.

Even the object oriented model might not be the best for server applications: web server applications are essentially batch programs that receive http requests and produce the corresponding outputs. They typically maintain little state, since most of the relevant state is kept at the client and passed along with each request. As Ian Sommerville stated:

> [...] where systems retain minimal state information, a functional rather than object oriented design may be used[16]

Still, as I have said, things could have been worse. Java doesn't allow the phantasmagoric flights of fancy that C++ does but, given the problems that have emerged in the years after its introduction, given the way in which the Java culture has managed to create internet software plagued by so many problem despite the use of such a clean language its lack of sophistication is more a merit than a problem.

III. NOMINA NUDA TENEMUS

One amusingly frustrating aspect of the Java culture (but in this the Java culture is by no means alone) is the abundance of coding conventions, some of which it would be an understatement to call anything but droll. There are coding conventions for (almost) all tastes, not seldom in contradiction with each other, taking the ones directions completely different from the others, although the different directions are often justified based on the same criteria. Some of these conventions are, in their limited scope, quite sensible, others not so much.

The vast majority of existing coding conventions are utterly irrelevant, dealing with such flimsy details as the capitalization of function names or the positioning of the brackets with respect to a *for* statement. A little pondering will reveal to every programmer that the real obstacle to reading a piece of code is almost invariably a poor or insufficient design: an object oriented program is hard to understand when the classes that compose it make little sense in the problem domain and correspond, to the elements of some convoluted solution generated in the mind of the programmer, obfuscated by a diet of coffee, sodas, and junk food. If the structure of the code is clear, the methods correspond to important operations in the problem domain (which, of course, should have been properly formalized and abstractly described during design), whether one capitalizes the names of the variables or not makes precisely no difference. The code written by a good programmer with a thorough understanding of the problem and the scientific culture to put the solution in a mathematically correct form, is clear regardless of whether she places the opening bracket in the same line of the "for" statement or not. The code generated from an insufficiently formal design and by programmers with insufficient scientific culture will be obscure no matter what coding conventions are being used.

Nevertheless, coding conventions are the apple of the eye of the software industry (academia seems, so far, relatively immune from the practice, but things are degrading rapidly), and we can expect more of them to be produced, due in large part to the evolution of industrial management practices. In the last fifty years, almost all branches of industry have fallen under the rule of professional managers, and managers have decided that the best way to preserve their organizations is to make them as independent as possible of the special abilities of the people who work there, so that everybody can be replaceable (whether they are right or not I couldn't say)[1]. As part of this effort, it helps if anything that can be regulated, is, and everything is homogenized beyond recognition. In some cases, and software is one of them, regulators are faced with an uncomfortable dilemma, since the things that really matter are practically impossible to regulate, while the things that are easy to regulate hardly matter at all. Almost invariably, in these cases, the industrial choice is to regulate the irrelevant things, and try to bring the others to the lowest common denominator.

Since we are not going to get rid of coding conventions anytime soon, we might as well try to make them sensible and minimalist: coding conventions should constrain the programmer only when—and to the extent which—this can result in provably better code, while they should otherwise let her free to experiment with her own creativity. The normalcy should be to have the programmers write code as they see fit, introducing conventions only when some practice may cause confusion, and always remembering that few conventions are better than many. That is, "no holds barred" should be the normal development situation, norms being introduced only when they are *provably* necessary, the burden of proof being on the party that wishes to introduce the norm. Of course, in order to make this situation work, one needs a special type of programmers, a topic that we shall consider in the last chapter. Given the current state of the Java culture, one purpose of good coding conventions should be to neutralize the poor coding conventions that are being enforced. Consequently, some of the "conventions" that appear in the following pages are of the form "do as you please (as long as you are consistent)". This might seem like a void advice but, with all conventions telling you exactly how you should do utterly irrelevant things, this is really a polite way of telling you to ignore them. The essence of a readable program—I will repeat it *ad nauseam*—is good design and, in the case of Java program, a correct object-oriented implementation of the design. We will consider these issues in the next chapter. The scope of coding conventions is much more limited: they should merely ensure that, whenever alternatives exists

that are seemingly equivalent but result in different quality of code, the clearer one is used. So, let us have a look at some of these conventions.

Capitalize as you see fit (but be consistent).

Many Java conventions deal with capitalization. There is an almost frantic haste to have all names capitalized according to their function: a capitalization for everything and everything with its capitalization. For instance, class names should be in mixed case (a capital at the beginning of every word that makes up the name) starting with a capital. Variable and function names should be similarly organized, but they should begin with a lower-case letter. This, to put it bluntly, is a dumb rule.

There is no reason why calling something `thisAndThat` should be better than calling it `this_and_that`. Many style standards in areas other than the internet advise to separate the words with an underscore rather than with capitalization. Not only is `this_and_that` often more readable than `thisAndThat`, the CTRL+← and CTRL+→ commands of most editors will allow you to place the cursor at the beginning of each word of `this_and_that` but not of `thisAndThat`. It is not a ground breaking difference, but it makes editing handier[2].

I start the private variables and the private methods of a class with an underscore. Since private variables and methods are often used in conjunction with the keyword *this*, I find that the extra space represented by the underscore makes for a better reading. That is, I consider that it is easier to see at first sight the variable name in

```
this._var;
```

than in

```
this.var;
```

Capitalization should be used to convey the meaning of a symbol. One reasonable rule is to start class and type names with a capital letter, writing variable and function names all in lower-case but, as the heading of this guideline states, any reasonable convention will do, as long as you apply it consistently.

Avoid using symbols that can be easily mistaken.

This recommendation is in direct opposition to what I have found in many a Java style manual. I quote from one of them:

> generic variables should have the same name as their type [with a different capitalization][3].

It is not quite clear what a “generic variable” is, but the idea is that if a program contains a data type `nudnik`, then the data type should be capitalized as `Nudnik`, and the definition of a variable of this type should be

```
Nudnik nudnik;
```

This kind of declaration must be avoided. For one thing, it is quite confusing since, while reading a program, one doesn’t always notices changes in the capital at the beginning. This solution will also make it hard to detect typos due to a slip of a shift key. A fragment such as

```
Nudnik nudnik, q;
    :
q = Nudnik;
```

will generate a compilation error that might not point to the correct cause of the problem or that might be difficult for the programmer to spot. If the name of the variable is distinct from the name of the type, as in

```
Nudnik nuisance, q;
    :
q = Nudnik;
```

the error will immediately be identified for the typo that it is. Capitalization is not the only culprit in town. In general, one should avoid using symbol names (variable names, class names, etc.) that differ only because of:

i) capitalization (as we have seen);

ii) the substitution of the digit 0 with the letters O or D;

iii) the presence or absence of an underscore;

iv) the substitution of the digit 1 with the letters I or l;

v) the substitution of the letter S with the digit 5;

vi) the substitution of the letter Z with the digit 2;

vii) the substitution of the letter n with the letter h.

Some of these rules might seem a little excessive because, after all, on your computer it might be very easy to tell an S from a 5. Consider, however, that on the computer of whoever will have to maintain your code it might not be so easy. In any case, if one has to err, it is better to err for excess of caution than for lack thereof.

* * *

Use verbs to name procedures; name functions after the thing they return.

For the purpose of this rule, a procedure is a routine that does not return any result except, possibly, an error code. Because of this, in order to be useful, the procedure must have some *side effect*: it can output something, change the state of the object to which it belongs, or change some of its parameters. A function is a routine that returns a meaningful value (that is, obtaining the value is the reason why one called the function), and has no side effects: it doesn't print anything[4], it doesn't change the state of the object to which it belongs or of its parameters. There are good reasons to require that any routine belong to one of these two categories, that is, that one avoid as much as possible the creation of hybrids: routines that return values and have side effects. I will consider these reasons later in this chapter.

A procedure is a command to an object to do something, and the most appropriate name for it is a verb in the second person imperative tense. In English this corresponds to the short form of the verb, but not in other languages; if the language

in which you write the code has no imperative tense, or has more than one, use the same verbal tense in which you would command your young son to shut up (there is no reason to use respect forms with a machine: the computer will not be offended but if your language has a respect form and you prefer to use it, go ahead). So one should write

```
Ukase r;
    :
r.wait(secs);
r.wahrt(secs);
r.espera();
r.attends(a, b, c, d);
```

A function is better called with the name of the thing it returns. Not only does this abide to common sense, it is also the standard way of naming mathematical functions, and the way in which functions are written in most languages other than Java. So you would write

```
Jumble r;
    :
x = r.log();
p = r.book("Unamuno");
q = r.arrondissement(12);
```

obtaining, from an object *r*, its logarithm, its books written by Unamuno, and its 12th arrondissment. The convention is quite sensible on the ground that in an assignment statement such as

```
x = log (y)
```

one says that "x becomes equal to the logarithm of y". Syntactically, the main verbal fragment of the statement ("becomes equal") is expressed by the symbol "="; the name of the function must tell us what x will become equal to, in this case the logarithm of y. Note that in the case of methods of an object the interpretation can be extended by reading the dot as a genitive marker. The expression `x = r.arrondissement(12)` corresponds to the sentence "x becomes equal to r's 12th arrondissment". All this holds when the value returned by the method is not

an object. The case of methods returning an object is more complex, and we shall consider it later.

Java coding guidelines have a candid love for standards but, in this case, they generally choose to ignore them (the *Not Invented Here* syndrome?) and suggest to call all methods with verbs. According to some conventions[5], one should write something like

```
x = getLog (y)
```

which is much less sensible since the fragment would correspond,linguistically, to "x becomes equal to the getLog of y".

Use method names that do not tell how the result is obtained.

This guideline is meant to help enforcing one of the most important aspects of object oriented programming: abstract data typing, which is quite closely related to the idea of information hiding, that it, to the principle that the interface to an object should be independent of the object's implementation. I will return on this concept more profusely in the following chapters. For the moment, this is a fairly general guideline, which applies to any kind of name of procedures, from Java objects to Haskell functions.

One has to be careful in this respect because sometimes the standard guidelines go in the opposite direction. One guideline for Java programming[6], for instance, suggests to use in the name of a method the prefix *get* whenever an attribute (of the implementation) is accessed directly, the prefix *compute* whenever a quantity is computed, and the prefix *find* whenever a quantity is looked up in a table. But the idea of information hiding is precisely that one should not know, just by looking at the signature of a method, how the result is derived, for knowing this would reveal details of the implementation. Calling a method `get_minimum` rather than `compute_minimum` reveals that there is a quantity, called *minimum*, which in the first case is stored as an attribute of an object, while in the second case is calculated: this is an implementation detail that the theory of abstract data typing compels us to hide.

What would happen for instance if, at some point during the development cycle, it were decided that the quantity *minimum*, which used to be computed, is better stored as an attribute of the object? Would one, because of this, change the interface

renaming the method `compute_minimum` as `get_minimum`? This would go flatly against any sound principle of programming.

The interface should always be more abstract than the implementation, and it should depend only on the functions that the object performs and on the behavior that it is required to exhibit. The interface should never depend on the way these functions are implemented. This is a design issue that I will consider in the following chapter.

Write the names in the language that is more convenient for you.

(but don't use accents or other regional characters.)

This suggestion comes from the unfortunate legacy of the ASCII code. In spite of the existence of more extended codes meant to include characters from a significant sample of the world's languages, still many applications, especially applications destined to a specialized technical public such as programmers, get confused by accents and other non-ASCII characters.

Several Java guidelines suggest that the programmer should always use American English to name things, but in some development environments, this is not the most convenient thing to do if we want the program to be an effective communication instrument. Variable names, for instance, are supposed to help programmer understand the function of those variables in a method or as part of an object's state. If a development team is entirely based in, say, France, and the programmers are unfamiliar with English, there is no reason why they should not call the indicator of the status of a file *fermé* instead of *closed*, as long as they are willing to use the spelling *ferme*, which will give them less problems with editors and compilers. (Of course, there is the slight complication that, in French, while *fermé* means *closed*, *ferme* means *farm*. This is a small price for the use of ASCII that the French programmers will just have to pay.) The purpose of variable names is to make the communication between programmers easier and communication, we have seen, is the main function of a program. One should use whatever language makes things easier in the environment in which the program is developed and maintained; a program, like any other instrument of communication, must adapt to the specific needs of the community that it serves. Program development, like any other human activity, is historically and culturally situated. All its components, and all the decisions that one takes must reflect the specificity of any singular development.

There are also other, more subtle, points that one should consider in this respect. One's mother tongue is one of the most important work instruments for a programmer. The world is made sense of, analyzed, and constructed through language. Breaking things down into pieces and putting them together is an important linguistic activity, and the essence of the work of a programmer. No two languages divide up the world in the same way, and it is quite possible that the use of Mandarin will suggest a Chinese programmer a way of dividing up a problem that English would not suggest to an American programmer. By restricting ourselves to one language we are limiting our structuring possibilities[7].

Use *while* or *do* for non-counting loops.

The *for* statement in Java is derived from C and has a much greater expressive power than the counting loop statement in older programming languages such as FORTRAN or Algol. These programming languages presented a clear distinction: the *for* statement can only be used for enumerable loops in which the number of iteration is known before the loop is executed. In many languages the condition is actually more restrictive: the *for* statement can only be used in counting loops in which a constant is added to the loop variable between iterations. That is, the *for* statement is limited to loops such as

```
for i=a to b step c do
     (body of the loop)
od
```

All other kinds of loops (e.g. loops that continue running until a certain condition has been verified) are implemented using the *while* or *repeat...until* operations:

```
q := file.read();
while not (q = Nil) do
    (do something with q)
    q ← file.read()
od;
```

The same division should be adopted in Java. Clearly, since the Java (and C) version of the *for* loop is as expressive as a *while*, in this case it is not a matter of the program's syntactic correctness, but of style. Nevertheless, the same principle should apply: the *for* instruction should be used only for enumerable loops, the *while* and the *do* should take care of the others. This doesn't mean that the *for* loop should be subject to the same limitations that it has in other languages. Loops like this one:

```
for (i=1; i<M; i *= 2) {
    (loop body)
}
```

or this one

```
for (double x=0; x<1.0; x+=0.1) {
    (loop body)
}
```

are not allowed by many programming languages, but there is no inconvenient in using them. The key property of a *for* loop should be that the number of times that its body is executed can be deduced *a priori* before beginning its execution, that is, that it does not depend on the body itself. In particular, this guarantees that a *for* loop always terminates, isolating into *while* and *do* loops potentially infinite repetitions and, therefore, making it easier to identify possible bugs that lead to the non-termination of a program.

* * *

Always place blocks in curly brackets.

A block of instructions should always be enclosed in curly brackets, even if it consists of a single instruction or it is empty. So one should write

```
while (a[i++] != NULL) {
}
```

rather than[8]

```
while (a[i++] != NULL)
    ;
```

This rule will eliminate ambiguous fragments such as:

```
if (cond_1)
if (cond_2)
action_1();
else
action_2();
```

which can be interpreted in two different ways:

```
if (cond_1) {
    if (cond_2) {
        action_1();
    } else {
        action_2();
    }
}
```

or

```
if (cond_1) {
    if (cond_2) {
        action_1();
    }
}
else {
    action_2();
}
```

In the first case, the function *action_2* is executed only when

```
(!cond_1 && !cond_2),
```

while in the second case the function is executed whenever `!cond_1`. It is true that many modern compilers detect this ambiguity and warn the programmer but this, of course, should not be taken as a reason to be sloppy!

In these examples, as in the fragments of code that I have written so far, I have placed the curly brackets in the same line as the *if* and the *else*. Some standards prescribe that the brackets should go on separated lines, aligned with the *if...else*, as in

```
if (cond_1)
{
    action_1();
}
else
{
    action_2();
}
```

Both forms are quite clear, and the programmer (apart from following company standards), can do it any way he pleases, as long as he does so consistently. Personally, I prefer the first form because it generates a more compact code. The legibility of the two versions is about the same, and writing more compact code has the advantage that a larger portion of it will fit in a screen of the editor or a printed page, giving the programmer a better view of the working context. In general, given two solutions with more or less the same legibility, I prefer the solution that generates the most compact code, both in order to give a better global view, and in order to take better advantage of blank lines, a topic that we shall consider shortly.

* * *

Manage sequences of decisions properly.

Each sequence of *if* .... *else if* ... statements should have a final *else* to capture the default case in which none of the conditions are true. If this is not done due to the logic of the program, the reason for the absence of the *else* should be documented *in situ*. In the case of an *if* with a single option (viz. without an ensuing *if*), no such precaution is necessary, as such statements are generally interpreted correctly, as in the following example:

```
if (a < 0) {
    a = -a;
}
```

But the comment is necessary in the following case:

```
if (x < 0) {
    action_1();
}
else if (x == 0) {
    action_2();
}
else if (x > 0) {
    action_3();
}   /* No else needed: all numbers have been
     covered. */
```

Of course, in these cases, the last condition is often redundant, but the programmer might decide to put it there anyway, either for the sake of clarity, or to make it easier to shuffle the alternatives around with the editor, should it be necessary.

Each block of a *switch* statement should be terminated with a *break* statement, as the *fall through* behavior of the switch statement is often confusing and can easily give rise to interpretation errors. The final *default* statement should always be present, unless the value tested in the switch is of an enumeration type, and all the values have been tested in the switch.

```
switch(value) {
    case 0:
    case 2:
    case 4:
        process_even();
       break;          /* prevent fall-through */
    case 1:
    case 2:
    case 5:
        process_odd();
        break;
    default:
        out_of_range_error();
}
```

A switch statement should never test a boolean variable, and it should always have at least two cases and a default. If these conditions are not met, it is better to use an *if ... else* statement:

incorrect	correct

`switch(val) {`
`    case 3:`
`        action_1();`
`        break;`
`    default:`
`        action_2();`
`}`
	`if (val == 3) {`
	`    action_1();`
	`}`
	`else {`
	`    action_2();`
	`}`

Do not test floating point numbers for equality.

Truncation and rounding errors are always present in numerical computation involving floating point numbers, and two numbers that are supposed to be equal often differ in the last decimals, enough to give *false* as the result of a comparison. Floating point equality (or, equivalently, test for zero with floating points) should always be done with respect to a tolerance, set depending on the precision that the program needs to attain, and on the error propagation properties of the algorithm that is being used.

incorrect	correct

`double v, w;`
`     :`
`if (v == w) {`
`    (action)`
`}`
	`double v, w;`
	`      :`
	`if (Math.abs(v-w)<this._tol) {`
	`    (action)`
	`}`

Writing programs for numerical analysis avoiding gross inaccuracies because of rounding errors is a science in itself, with a large set of sophisticated techniques and

solutions[9]. It is clearly not within the scope of a general-purpose style book to go into this very specialized domain. Moreover, in the case of numerical accuracy, the issue is not quite one of style-as-communication, but of sheer program correctness and adequacy to its algorithmic purposes.

Use operation with side effects in a safe manner.

Operations with side effects can, if used without constraints, severely impair the readability of a program. They must be only used in context in which their effects are clear, that is, in the following contexts. (In the example, assume that *f*, *g*, and *h* have side effects.)

i) By themselves; for example:

```
++k;          /* correct */
i++ - ++k;    /* incorrect: not by itself */
```

ii) alone or with a constant in the right-hand side of an assignment; for example

```
y = f(x);         /* correct */;
y = k++;          /* correct */;
y = f(x) + 1;     /* correct */
y = f(x) + h(x); /* incorrect: not alone */
y = f(z) + z;     /* incorrect: f might change z */
y = i++ - ++k;    /* incorrect */
```

iii) alone or with a constant in a condition; for example:

```
if (p++ == NULL)  /* correct */
if (i++ == --k)   /* incorrect */
if (f(z) == z)    /* incorrect: f might change z */
```

iv) as the only expression with side effects in the parameter list of a function call; for example:

```
f(g(z), 3);    /* correct */
f(g(z), z);    /* incorrect; see above */
f(g(z), h(z)); /* incorrect */
```

v) as the condition of a loop;

```
while(f(z))          /* correct */
while(f(x) != g(x)) /* incorrect: not by itself */
```

vi) as the condition of a switch;

```
switch(f(z))     /* correct */
switch(c = f(x)) /* incorrect: not by itself */
```

vii) as part of chained operation.

```
c.f().g().h(); /* correct */
```

Other uses of operations with side effects tend to generate unreadable code, and should be broken into several instructions with the insertion of suitable temporary variables.

* * *

Do not allow results to depend on the order of execution.

The result of an expression should always be the same, independently of the specific execution order that the compiler will implement.

```
k = (h++) + h; /* incorrect: (h++)+h != h+(h++) */
p.func(p++);   /* incorrect */
k = v[k++];    /* incorrect: unspecified behavior */
```

*   *   *

Declare all methods at the class level.

In Java, methods can be declared within other methods, or even within blocks. These practices should be avoided, as they obscure the code considerably. All methods should be declared at the class level, distinguishing clearly in the code organization between the private methods of the class and the public methods that constitute its public interface.

*   *   *

Define inverse methods one based on the other.

Methods that test some conditions and return a boolean value, a distance, or a degree of similarity are often defined with their inverse. That is, every method that tests a condition is usually paired with an "inverse" method, which returns true when the other returns false and vice-versa. In this case, only one of the two methods should be defined from scratch, while the other should be defined in terms of the first. For instance, one can define some notion of equivalence between objects of a same class:

```
class Vibrissa {
      :
    bool equivalent(Vibrissa hair) {
        (code to determine the equivalence here)
    };
    bool different(Vibrissa other) {
        return !this.equivalent(other);
    }
}
```

If the definition of equivalence is changed during the project, or if the implementation of the equivalence function is changed during maintenance, this definition will avoid inconsistencies.

* * *

Do not pass too many arguments.

Functions with more than six or seven arguments should be avoided. If a function receives more than six or seven independent parameters, this is often due to insufficient abstraction: the design of the class should be revised.

If a function has more than three parameters, the common way to write them down is to have the first parameter on the same line as the function name and the rest on different lines, one per line but, as always, any reasonable rule will do, as long as it is followed consistently.

* * *

Write a function as you would write a short story.

The body of a function, if well written, is often the best description available of what the function does. Write the function thinking of other people, not the computer. The compiler is perfectly capable of taking care of the program independently of how clearly it is written, and with a decent optimizing compiler, code optimization should be a concern only for a handful of functions[10], so the programmer should concentrate on writing *well written* code. In addition to the advices that we have seen so far, and to the programmer's sensibility, one should take advantage of all typographic techniques that can help clarify the code.

i) Make good use of empty lines. Empty lines are a powerful instrument for visual grouping, and should be used in a semantically consistent way. Instead of placing them haphazardly throughout the code, use them as paragraph separators in order to visually divide the code into semantically coherent units.

ii) Apart from those that perform the function of point i), the code should contain no empty lines. The idea, here as elsewhere, is to create a clear but compact code, so that a single editor screen, or a single printed page contain as broad a context as possible. In the ideal situation, each function should fit in a page or in a screen except, as we have already observed, for functions with long *switch* statements. In this case, place CTRL+L characters within comments strategically to break the page between semantic units when the program is printed.

iii) Indentation is another important visual indicator of semantics, and its intelligent use can do a lot to improve the readability of a function. Whoever had a chance to read an old FORTRAN IV program, without any indentation, can't fail to agree. Indentation should be deep enough to clearly distinguish the different indentation level, and shallow enough to allow the creation of several indentation level in the horizontal space of the page, and to avoid spreading the code too thin horizontally. An indentation of four to six spaces between levels will be adequate in the majority of cases.

iv) Indentation should always be done with spaces, never with tabs. Different editors format tabs in different ways, and what appears very readable on your editor might show up as a mess in the editor of the person that, at some point in the future, will have to understand your program. Many editors give you the option of automatically translating the TAB key into the appropriate number of spaces. Use it.

v) Indentation should highlight the logical structure of the program, and not merely the syntax of Java: whoever will read your program will in all likelihood already know the syntax of Java without your indentation to remind it to them; what your reader will not know, and what your indentation should communicate, is the structure of the function. For instance, in the case of alternative sequences, an indentation such as:

```
if (n==1) {
  :
}
else {
    if (n==3) {
      :
    }
    else {
        if (n==5) {
          :
        }
        else {
              :
        }
    }
}
```

is formally correct, but tends to obscure the structure of the function. In this case, it is better (and more compact) to write:

```
if (n==1) {
   :
}
else if (n==3) {
   :
}
else if (n==5) {
   :
}
else {
   :
}
```

Note that in this case all the operations that are performed in the different cases are at the same indentation level, highlighting the fact that they are semantically equal alternatives.

In the last example, it should be noted that the solution proposed violates the norm on the explicit placement of curly braces even in alternatives composed of a single instruction (in this case the alternative of all the *else* except the last one was a single *if*). This is an example of the last rule of this chapter, possibly the most important of all.

*   *   *

Do not use norms as an excuse for poor code.

The only guarantees of a clear code are the culture, sensibility, and sense of mathematical elegance of the programmer who writes it. Rules can help creating more consistent code within a project group, and even to create a certain “corporate culture” that might make the code less alien to future readers. But whenever applying a norm would result in a more obscure fragment of code, drop the norm. Norms are an instrument towards clearer code, not an end in themselves, and whenever an

instrument works against its own ends, it is better left unused. The choice between a norm and a readable code is always clear: go for the readable code.

Of course, this norm too must be applied with good engineering judgement: the norm, after all, applies recursively to itself as well.

IV. OBJECT (ALMOST) ORIENTED

The guidelines given in the previous chapter were of a very general nature, and they were not especially related to the object oriented nature of the Java language. One could apply them when programming in pretty much any programming language, be it C, FORTRAN, or whatnot[1]. In this section, we will take a more specific approach and look at guidelines that apply to Java *qua* an object oriented programming language. It is worth remembering that here we are concerned with object oriented *programming*, which is a different thing from object oriented *design*. The former entails the use of certain kinds of programming languages and constructs, while the second is a discipline and a set of techniques to determine the structure of a program and the modes of interaction of its components. Nothing prevents a designer to put together an object oriented design and then decide to implement it using a programming language that object oriented is not. There are many cases in which this is the best solution.

The opposite situation is much less common: the most compelling reason for using an object oriented programming language is to implement an object oriented design because if the design is, for instance, functional, one would not be able to use the most important properties of the language, and the whole object oriented paraphernalia would come out to be just an useless superstructure. Nevertheless, we know that the reasons for choosing a programming language rather than another are rarely purely technical: a company must consider the languages that its programmer can maintain, the investment (in equipment and training) necessary to switch to another language, compatibility issues, standards, corporate culture, the idiosyncrasies of the vice-president of engineering, and so on. So, as uncommon an occurrence as it may be, it might happen to use an object oriented language to implement a design that object oriented is not. In this case, the guidelines of this

chapter still apply, albeit less compellingly so, because the corresponding features will be used only sparingly.

It is however important to keep the distinction in mind: these guidelines refer to programming using an object oriented *language*, and not to object oriented *design*, a topic that I will take up in the next chapter. The distinction is important especially because in many programming books it is underplayed to the point of disappearance: many authors seem to imply that, just because Java is an object oriented language, this will be sufficient to give you all the object orientation that your little heart desires. This is, at best, an illusion. Ed Post wrote (only half jokingly): *a* real programmer can program in FORTRAN using any programming language[2]; whether you program in an object oriented way or not is mostly a matter of your own programming discipline. An object oriented programming language will help you somewhat, but I have seen some very good object oriented programs written in C and Java programs that were all but object oriented.

* * *

The most important reasons to use a clear programming style is to make other programmer understand what a program *means*, that is, to convey the program's *semantics*. To this end, the program statements that have a special and fixed semantics play an important rôle. These statement have a unique interpretation, set once and for all in the language specification and enforced by the compiler, and greatly reduce the uncertainty of the people who read the program, providing some kind of *semantic anchors*: points of fixed meaning that help making sense of what one is reading. (Remember, once again: the primary purpose of a program is to communicate the solution of a problem to other programmers.) For example, a *constructor* is a function that has the same name as the class in which it is defined, that does not return any value, and that is not invoked directly, but through the keyword *new*. Its semantics (enforced by the compiler) is that it creates a new object, different from all the objects created hither or thither during the execution of the program. An object whose status is initialized inside the body of the constructor. So, given a call like

```
Shibboleth q = new Shibboleth();
```

we immediately know that *q* is a new object, distinct from all the objects afore defined in the program. It may well be that the status of *q* is not properly initialized (a constructor is subject to programming errors just like any other method), but we can guarantee that the object is unique and that operating on it will not interfere with any other object of the same class. This kind of information, hard-coded into the compiler and therefore known instinctively to any experienced programmer, is of valuable help to understand what a program does, and the writer of any program should make the best and most evident use of it.

Unfortunately, this kind of valuable fixed-semantics statements are not as widely available in Java as they are in other programming languages. Whenever a statement with a fixed semantics is not available, the best thing one can do is not try to replace it with a standard statement that enforces *de facto* the same semantics, because this can lead to undue inferences by the person reading the program. An example is given by the following guideline.

* * *

Do not assume the existence of a destructor.

(In particular, do not use the *finalize* method.) The Java programming language does not define destructor methods. Rather, it has a garbage collection mechanism that will destroy an object when all the variables that point to it go out of scope. The garbage collector comes handy in many occasions, especially when, in the middle of a function in which there are dozens of objects that may or may not have been created, one has to leave the function because an error has been detected. Thanks to the garbage collector, it will not be necessary to keep track of which objects have actually been created in order to delete them. A simple instruction

```
if (error_condition) {
    return NULL;
}
```

will suffice. On the other hand, Java does not have a function that is guaranteed to be called when, *and only when*, an object is eliminated. In C++, for instance, one would declare such a function as `~<class_name>`:

```
class y {
    public y(....)
        (constructor)
    }

    public ~y() {
        (destructor)
    }
}
```

The semantic peculiarity of `~y` is that, just like the constructor y, it can't be called directly, but only by using the compiler keyword *delete*:

```
y foo;
foo = new y(....);
    :
delete foo;
```

Java has no function with a similarly restricted semantics. (In Java, in any case, the function `~y` would never be called directly through the use of a specialized keyword: only the garbage collector would be allowed to call it.)

In order to "simulate" it somehow, the Java documentation suggests to place all the operations that must be executed when an object is destroyed in the method `finalize`, which is a method of the class `Object`, from which all classes inherit. The documentation says

> The general contract of finalize is that it is invoked if and when the Java virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not died, except as a result of an action taken by the finalization of some other object or class which is ready to be finalized [....]
>
> The finalize method is never invoked more than once by a Java virtual machine for any given object3

This statement is ambiguous. The conjunction *if*, used to introduce the subordinate, seems to imply that there is no guarantee that the method will be called by the Java virtual machine. In other words, what one can evince from this statement is that the virtual machine will call the method *at most once* and that, if the method is called, no other method of the object will be called afterwards. So, the "clean-up" operations that it implements are not quite guaranteed to be executed.

The most problematic aspect of this method, however, is that, although *finalize* receives a specific semantic (a semantic that derives from the compiler, and not from its contents), there is nothing special about its status: it is just a method of the class `Object` that is inherited by any class in a Java language. That is, `finalize` is not part of the *language* specification: finalize is not a reserved keyword of the language like `for` or `while`: it is just like any other method that you can define in one of your classes. This regular method, whose definition is not part of the language has, however, a very peculiar semantics attached to it (it is guaranteed--maybe--to be called, and no more than once), which is enforced within the Java virtual machine, but *not* within the compiler. This means that, while the virtual machine guarantees that it will never call the method more than one for a given object, there is nothing that will impede to a program to call it as many time as one pleases. Consider the following code fragment:

```
public class Nimrod {
    public void oops() {
        this.finalize();
    }
}

Nimrod u = new Nimrod();
    for (int i=0; i<100; i++) {
        u.oops();
    }
```

What happens here to the guarantee "the finalize method is never invoked more than once"?

* * *

Use copy constructors.

A copy constructor is a constructor that takes as argument an object of the same class as that of the constructor, and returns an object with the same invariant as that which is passed as parameter. The fragment:

```
Willy_nilly x, y;
y = new Willy_nilly(par1, par2);
x = new Willy_nilly(y);
```

produces in x an object that is (should be) a copy of the object y, in the sense that the atomic attributes of x have the same values as those of y, and the objects contained in x are either same as those contained in y or a copy thereof. This assumes, of course, that a copy constructor is defined for the class Willy_nilly:

```
class Willy_nilly{
      :
    public Willy_nilly (int p1, int p2) {
          :
    }
      :
    public Willy_nilly (Willy_nilly x) {
          :
    }
}
```

Copy constructors are important because with them the copy semantics is enforced by the compiler: in the first fragment of code above, one is guaranteed that the object x is *distinct* from the object y. The programmer is in charge of building the copy constructor, of course, so that the constructor itself may contain errors, and the "copy" might turn out not to be equal to the original after all. Nevertheless, one can be sure that the copy semantics is enforced: x is a new object, distinguished from y. This fact alone can save a lot of debugging headaches, since the individuation of an object built with the wrong parameters is considerably easier than tracing the potentially convoluted problems caused by a *faux* copy.

A copy constructor should make a copy of all the members of the state of the original object that affect the invariant. State variables that do not affect the invari-

ant (e.g. a cache) should not be copied, and should be allocated anew. If an object contains a pointer to a data element or to another object, the most sensible behavior for a copy constructor should be to copy the data element or to create a new member object using the copy constructor of the corresponding class. There are special cases in which this behavior is not appropriate; these cases should be treated accordingly and well documented. One never knows when objects of a class will be part of the state of a class that defines a copy constructor. For this reason, a copy constructor should be defined for *every* class.

Quite surprisingly, Java manuals often takes a negative view of copy constructors, discouraging their use and advising the programmers to use the method `clone` instead. We find here the same flaw as in the previous example: putting the semantics into the libraries, rather than in the formal definition of the compiler, where it belongs. While the copy constructor has a semantics enforced by the compiler, `clone` is just a method, and it doesn't take an excessive leap of the imagination to figure out a programmer (pressed by time, an unreasonable project manager, or by even more unreasonable deadlines) override the default `clone` method of the `Object` class with the following "provisional" implementation:

```
public Willy_nilly clone() {
    return this; /* to be changed*/
}
```

This implementation, duly forgotten amidst the dozens of things that poor management piles up on the programmer's head, is bound to generate "inexplicable" problems later in the development.

It would seem obvious that, whenever there is a language construct that enforces the desired semantics, that construct should be used but, apparently, in the Java world, such statement is not quite obvious, at least considering the advice that is being dispensed by the language pundits:

> If you are a C++ programmer and feel the urge to write a copy constructor in a Java class, STOP! Close your eyes, Take a few deep breaths. Then—when you feel your ready (sic!)—open your eyes, implement *Cloneable* and write *clone*. It will be OK[4].

My impression is that the solution will be just as “OK” as the spelling of the expression “feel your ready”. I would, of course, give exactly the opposite advice to Java programmers.

*   *   *

Get new objects only through the appropriate class constructor.

This situation is similar to the previous one, but it involves two objects, so the constructor involved is not quite a copy constructor. Suppose we have an object `x` of class `Pundit`, which contains an object of class `Dispute`, and we want a copy of the latter. Consider the following code:

```
class Dispute {
    private int _x;

    public int X()  { return _x; }
}


class Pundit {
    private Dispute y;
}
```

If we want a *copy* of the object *y* in the class `Pundit` (we might want a *reference* to it, which, of course, would entail a wholly different story) the correct way to do it in an object oriented language is through a constructor:

```
Pundit x = new Pundit();
Dispute z = new Dispute(x);
```

or, if one doesn’t want to make the constructor of `Dispute` to depend on the class `Pundit`, by a method that returns a reference to *y* followed by a copy constructor[5]:

```
Pundit x = new Pundit();
Dispute z = new Dispute(x.dispute());
```

This is not the way in which the Java conventions recommend getting a copy: in a Java program you are likely to find a method of the class `Pundit` called *`get_dispute`*, or something to that extent that returns (supposedly) a copy to the `Dispute` class inside `Pundit`. This way of getting a copy introduces two problems in the program.

Firstly, it introduces a bizarre functional situation in what should be an object oriented framework. I will repeat myself here: object oriented languages have one canonical way of building new objects, and this is the use of a constructor. Unless there is a very compelling reason for doing otherwise, one should embrace coherence and use constructors. I will give here my own personal version of Occam's razor: *methodi non sunt multiplicandi sine necessitate.*

Secondly, we have the same problem that we encountered in the case of `clone`: the program semantics of `get_dispute` is uncertain because there is no syntactic (viz. enforced by the compiler) distinction at the level of the caller between a function `get_dispute` that returns a new object and a function `get_dispute` that returns a reference to the object *y*[6]. That is, from the outside, the two functions

```
Pundit get_dispute() {
    return new Dispute(y);
}
```

and

```
Pundit get_dispute() {
    return y;
}
```

(where `y` is a private attribute of the class `Pundit` as defined above) are indistinguishable. The semantics of `get_dispute` (which has an impact on the external behavior of the class `Pundit`) depends on its implementation in a way that it neither recorded nor enforced in its signature. It does, in other words, violate the principle of information hiding.

* * *

Do not define unnecessary default constructors.

A default constructor is a constructor that takes no parameters and initializes the variables of the class to some default values. These constructors should be avoided for two reasons.

i) Their presence encourages programmers to create objects before all the necessary data for their proper initialization is available; this, in turn,

ii) results in objects being created in a status far removed from that needed in the program, and creates the need for massive change of status after object creation, requiring the definition of many status-change functions that break the abstraction and problematizes the preservation of the class invariant.

* * *

Do not use public attributes in a class with an invariant.

Telling people not to use public attributes these days is a bit like telling them not to smoke: you would think that, by now, everybody should know better. Nevertheless, a lot of people still smoke, and a lot of people still makes a rather indiscriminate use of public attributes. If you remember the general guideline about function names (they should not reveal how a value is obtained), you will see why public attributes should be avoided at all costs: they reveal the implementation of a class and are, *ipso facto*, not abstract enough to be part of the interface. Very seldom is there an excuse for using public attributes. The only case in which they might be useful is the definition of abstract symbols with no logical connection to the value that represents them, that is, symbols whose only requirement is to exist and to be different from one another. That is, one might find it useful to define

```
public class direction {
    public static final int NORTH = 1;
    public static final int SOUTH = 2;
    public static final int EAST  = 3;
    public static final int WEST  = 4;
              :
}
```

The four constants defined here are pure symbols: all they need in order to function is to be defined and distinguishable; the fact that they have values 1, 2, 3, and 4 is contingent. As pure symbols, these constants are quite isolated from the vicissitudes of the implementation. In C, in these cases, one would probably declare an *enum* data type.

This is not the case in many occasions in which the value is the meaning of the symbol. Consider a class that allocates a certain amount of memory to do certain things. One might be tempted to declare publicly the number of items that the class can contain:

```
public class stuff {
    public static final int max_items = 100;
             :
};
```

This solution would get us in trouble if we changed the implementation so that the maximum number of elements is computed on the spot, based maybe on the amount of memory available to the process and on the state of the program. Much better would be to define a function that, in the static implementation, just returns a constant:

```
public class stuff {
             :
    public int max_items() {
        return 100;
    }
}
```

Any modern compiler would spot the fact that the function returns a constant and replace it with the constant in the compiled code, so that the performance cost is zero (besides, it is hard to imagine a function like this in the middle of a tight loop).

Many values necessary for the functioning of a class can be either computed once and then stored or computed on demand. Which solution is adopted is a matter of class implementation, an internal decision to which the interface should be indifferent.

There is a reasonable use for public variables: a class that doesn't maintain an invariant, that is, the data type that in Pascal is called a *record* and in C a *structure:* conceptual entities that just aggregate variables without maintaining an invariant and without providing any abstraction. These entities should be modeled as *structures*: classes without methods and in which all variables are public[7]. Any entity that provides an abstraction and that maintains an invariant should be modeled as a *class* in which all the variables are private and the only access allowed is through the public methods of the interface.

* * *

Classes will maintain their invariants.

The class invariant must be part of the post-condition of every class constructor (including, if it exists, of the default constructor), as well as part of the precondition and postcondition of every public method. Private methods may invalidate the class invariant, but it must always be restored before the public method that called them returns, even if it throws an exception.

In the case of public inheritance[8], the methods of the subtype (the class that inherits) should conform to the following rules:

i) the preconditions of the derived methods must be at least as weak as those of the methods they override;

ii) the postconditions of the derived methods must be at least as strong as those of the methods they override.

That is, the methods of the subclass should expect less from the data they receive than the corresponding methods of the base class, and should deliver results subject to stronger constraints.

* * *

Avoid the use of anonymous blocks.

Java allows the definition of *anonymous blocks*, that is, groups of statements that are not part of any method and that are executed in the order in which they appear in the body of the class. The syntactic status of anonymous blocks is therefore different from that of methods, since the latter can be moved around without changing the semantic of the program. That is, the following two pieces of code:

```
class C {
    public int f1() { A };
    public int f2() { B };
}
```

and

```
class C {
    public int f2() { B };
    public int f1() { A };
}
```

are semantically equivalent, while the following two

```
class C {
    { A };
    { B };
}
```

and

```
class C {
    { B };
    { A };
}
```

are not. The structural status of anonymous blocks is therefore quite uncertain. If we consider them equivalent to pieces of the same function (as they are presumably meant to be), then they break the relation between spatial contiguity and temporal contiguity in the code of a function. The blocks A and B in the following example

```
class C {
    { A };
    public int f1() {...};
        .
        .
    public int fn() {...};
    { B };
```

are always contiguous in the execution sequence (just like the blocks that constitute a function) but very distant lexically, a circumstance that can decrease considerably the legibility of the code.

The only thing anonymous blocks can do that methods can't is the implementation of the so-called *static constructor*, a piece of code that initializes the static portion of a class. This static constructor is semantically unclear (to me, at least). The standard Java documentation claims that the static constructor is executed only once: the first time that an object of the given class is created. This entails that, if no object of the class are ever created—for instance if the class is entirely composed of static methods—the static constructor is never executed. In this case, it is better to make the semantic status of the static constructor clear by putting it into a static method called once—and only once—by the regular constructor:

```
public class stuff {
    private static boolean first_time = true;

    private static void _start() {
        (static constructor here)
        stuff.first_time = false;
    }
      :
      :
    public stuff() {
        if (stuff.first_time)
            stuff._start();
        (regular constructor here)
    }
}
```

With this solution, any useful application of anonymous blocks is eliminated, and their use can be happily avoided.

V. OBJECT-(NOT-QUITE)-ORIENTED DESIGN

In an ideal world, perfection is everywhere, truths are absolute, and software design is done independently of program implementation. A designer will be largely unconcerned with the programming language in which the design will be implemented (in an even more ideal world, the designer will be oblivious even of the fact that the design he is creating will eventually result into an executable program), but will create a formal system based on his understanding of the problem and his mathematical acumen, express it using a suitably formal notation, as unambiguous as a programming language but not bound to any specific execution model, pass it along to the development team, and happily take a long vacation on the floating islands of lake Titicaca.

We do not, as most readers are undoubtedly aware, live in such a world. We are almost perfectly imperfect, the only absolute truth that I know of is that there are no absolute truths, and software designers create their designs very much with a specific implementation in mind. At least, this is the impression that one gets by looking at the majority of software that appears on the internet[1].

Looking at many an internet product, one has the impression that often the designers are or have been Java programmers themselves, and that the design is influenced more by the characteristics of the language in which they know that the program is going to be implemented than by those of the problem that they are trying to solve. I must say outright that, although I do believe that this is a censurable habit, and that it is responsible for many serious design flaws, I do not blame the designers for it. Not exclusively, at least. I must confess, for instance, that in more than one occasion I have been guilty of exactly the same sin. It is hard not to commit it in the current industrial environment since most of the people to whom the designer of a software system is reporting don't have the culture necessary to read a formal

specification and often force the designer to go directly from the phase of fuzzy and largely meaningless architectural diagrams to that of creating the program. Since I still have to see a block diagram worth the paper it is printed on, this means that the design begins, *de facto*, with the creation of the class structure, a structure that is often cast directly in a Java (or pseudo-Java) form, thereby resulting very vulnerable to contamination by the quirks and peculiarities of the language[2].

A trivial observation, heard many times but worth repeating *(repetita juvant)* is that the programming model used in the design should not necessarily be the same as the programming model used in the implementation. Object oriented design (an abstract design model and a collection of design criteria) and object oriented programming (writing code using an object oriented language) are two quite different and independent things. It is very possible and, in some cases, useful, to create an object oriented design and then decide to implemented it in a language that object oriented is not: for instance using a general purpose language such as C, a functional a functional language such as Haskell or ML, or even a logic language such as Prolog. The converse is also true: it is perfectly possible to use Java to implement a design that was not intended to be object oriented. (The first decision is more common: implementing a functional design using an object oriented language can sometimes be awkward.) The decision to implement a system using a language rather than another depends on a number of factors, some of which have little to do with the technical and formal aspects of a system: existence of suitable software libraries, possibility to execute the program on certain computers and operating systems, existence of technical expertise within the company, corporate culture, and so on. The design, on the other hand, should be done looking solely at the formal aspects of the computational problem for which a software system is designed.

Of course, this ideal situation belongs, in many cases, to the far far away realm of wishful thinking: often the designer is severely constrained in the choices of the model in which the design can be carried out, and even the formalism in which the design will take form. I have seen, for instance, projects in which management required that the design be specified using UML or that the messages that go through the system use XML even before the requirements and the problem were clearly formalized. Such constraints are usually recipes for disaster but, short of quitting with heroic disdain, there is very little a programmer can do about them.

Let us assume, then, that an object oriented design is, *de facto* if not *de jure*, a requirement of the project. We are still faced with the puzzling observation that many

object oriented designs that appear in the internet milieu are of a tremendously poor quality, in many cases resulting in a bizarrely ill-formed mish-mash between object orientation, functional programming, and plain old-fashioned spaghetti code. In order to understand why this is the case, I shall start, a little pedantically, by going briefly over the basic ideas and assumptions of object oriented design.

Object Oriented Interlude

Formally, object orientation is a design (and programming) style synthesizes three ideas: abstract data typing, inheritance (viz. sub-typing of abstract data types), and overloading (the possibility of having different signature for the same functor). Of the three, the only one unambiguously and formally defined, and the one whose benefits are more readily apparent, is abstract data typing, a technique that consists of defining a data type (*class*, in the object oriented parlance) only through the operations and functions that apply to it *(methods)*[3]. It is worth noticing that in the classical definition of objects there are no methods: objects receive and send *messages* to each other, and a class is defined by the syntax of the messages that its object can receive and send, and by their behavior upon receiving a message. The first simplification to this model was to define a *message type* and place it as the first element of the message, so that a message sent to object *u* would look like "here is a message of type *f* with parameters *x*, *y*, and *z*". The following stipulation was that each message type was to be sent always with the same number of parameters, and that each parameter had to have always the same type. With these simplifications it was quite obvious to transform the statement "object *u*, execute message *f* with parameters *x* and *y*" into the now equivalent functional form $u.f(x, y)$. With these changes, an abstract data type is now seen as a collection of operations, that is, as an algebra. This algebraic point of view is by now so dominant that the original, message based, formulation is practically forgotten and, while the message formulation is more general, I will stick to the algebraic one.

In this book, however, more than in formal definitions we are interested in analyzing what object orientation entails for the structural design of programs. What are the design characteristics of object orientation? How does it differ from, say, functional programming or structured programming?

The basic idea of object oriented design is that the solution of a computational problem should be structurally isomorphic to the problem. That is, the elements and

behaviors of the program should be the same (structurally) as the elements and behaviors that one finds in the problem domain. The recommendations that one finds in manuals of object oriented design tend to enforce this isomorphism, and to ensure that the designer identifies the structure of the solution *in the problem domain*. In the course of doing a correct object oriented design, one ends up thinking very little about computers and computation, and a lot about the nature and the structure of the problem that one is solving. This shift in the focus of attention of the designer is one of the major characteristics of object oriented design. While I am not as enthusiastic a supporter of object oriented design as some people might be, I recognize it a certain internal coherence and, more importantly, a conscious effort to direct the attention of the designer where it belongs: away from the computer on which the program will be executed—or even away from the fact that the program will eventually be executed—and towards the structure of the problem that is being modeled. This effort gives object orientation a validity *vis à vis* the nature of programs as communication artifacts that we have remarked in the first chapter: it is much easier to communicate with somebody who is trying to solve the same problem as we are if we talk about the problem than if we talk about the possibly idiosyncratic solution that we are cooking up. The fact that there are object oriented languages in which the design can (relatively) easily translated, and therefore the fact that object oriented design is especially relevant for problem models that will, eventually, result in executable program is, in this sense, quite incidental and, while making the method more useful, it has little bearing on its internal logic and coherence.

* * *

In this chapter I will discuss some norms of conduct that can help the designer build a better design. With respect to the previous chapter, where we were dealing with relatively concrete problems of implementation, we are now in much more abstract, uncharted, and maybe unknowable waters. Consequently, the norms will also be of a different nature: there will be fewer of them, they will be more abstract and, in a way, hazier. More discursive and less prescriptive. The first two guidelines are a compendium of what has been said in the previous chapters and in the brief introduction to object orientation that opened this chapter. I regards them as the most important rule of all: if you keep only two things in mind during your design activity, let them be these guidelines.

Structure the program based on the problem, not the solution.

This guideline is a logical consequence of the considerations about the nature of object oriented design that I was proposing in the opening of this chapter. Object orientation postulates that the structure of a program should be isomorphic to the structure of the problem that one is trying to solve, that is: a program, before being the solution of a problem, is a *structural description* of it. A program that fails to describe its problem domain properly is not a good object oriented program.

Proper design methods are an application of this principle of isomorphism. More precisely, object oriented design is a formalization (with all the caveats of the case) of a linguistic description of the problem one needs to solve. An approach to design, for instance[4] suggests that we start with a linguistic description of the problem domain, and then identify the relevant *nouns*, which will become the objects of the design, and the relevant *verbs*, which will become methods of the objects corresponding to their (grammatical) subjects. One is assuming, of course, that the problem domain is consolidated enough to have created a lexicon in which the relevant concepts are expressed as nouns and the relevant behaviors as verbs, without the use of periphrases. If this is not the case, then one should start with the clarification and the formalization of the language of the problem domain, even before starting to think about a program to express that domain. Poorly understood or insufficiently formalized domains do not lead to good programs and, in many cases, the inability to formally describe a domain is a sign that formal machines such as computers should stay out of it. Software design methods that are not based on the "primacy of the noun", so to speak, are deemed unsuitable to form the basis of an object oriented design. The Jackson method, for example, is not considered a good object oriented design method because it makes *events* its primary entities of interest, and considers objects in a somewhat subordinate rôle, only as a support for events[5]. Again, it should be remarked that object oriented design is not a panacea:

> It is unwise to be dogmatic about the design process and always adopt an object oriented approach irrespective of the system being developed. An object oriented view of system design is not always the most natural. At some levels of abstraction, a function-

> al view is easier to derive from system requirements than an object oriented view. In particular, where systems retain only minimal state information, a functional rather than an object oriented design may be used[6].

Nevertheless, if one has decided that object oriented design is appropriate for a certain system (and, yes, this might be a big "if"), there is no excuse for not doing it right: the proper method must be followed.

* * *

Design interfaces based on function, not implementation.

This guideline is a compendium, at a higher level of abstraction, of several observations and guidelines introduced in the previous two chapters and is, at the same time, the complement of the previous guideline. The previous guideline was structural, and considered the *nouns* in the problem description, this one is dynamic, and deals with the *verbs*. As we have seen, the important names in the problem domain become classes. The objects that belong to these classes *do* things, as expressed by the verbs of the description; these verbs are the nucleus of the design of the methods that will define the behavior of the classes. The functions that one defines in the interface of a class (its *public* methods, in Java) should reflect the *actions* of the objects of that class in the problem domain, and not the way in which the computational solution of the problem works. This general principle was the source of many syntactic guidelines of the previous chapter, from the elimination of the implementation dependent distinctions in function names to the avoidance of public attributes. The present guideline, however, places the previous ones in a broader frame: the interface of a class should be completely independent of the way in which the class is implemented, and should reflect only the rôle of that class in the formal description of the problem.

This guideline is violated in many a design and this is one of the major causes of code obscurity and maintenance costs. There are many ways in which the implementation can creep up in an interface. The previous chapter should have warned you about the most obvious ones but there are, naturally, subtler ones against which there are no simple syntactic safeguards; the only way to avoid them are a thorough understanding of the problem and a programming culture based on mathematical abstraction[7]. The consequences of violating this guideline are often serious, affecting the maintainability of the program and its structure. Maintainability is affected because an exposed implementation is an implementation that can't be changed. Implementation decisions that affect the interfaces will have to be made very early in the life of a project (viz. at the time the class structure and the class interfaces are designed)—too early, in fact, leading to poor structuring—and they will be made permanent by their reflection on the interface, creating the conditions for maintenance problems later on. As serious as the maintenance problems may be, the most troublesome effect of the presence of the implementation in the class interface lies in the way in which it affects the structure of the program. We have seen how, in object oriented design, the high level structure of a program should reflect the structure of the problem domain. This structure is the most general, abstract, and invariant that one can create for a program that will work on that particular problem (a more abstract structure would rip the program away from the problem). On the other hand, the particular solutions chosen for the implementation are contingent: for each problem there are many possible design solutions and, for each solution, many possible implementations. Making the interface depend on the implementation brings the contingency of the solution at the level of the structure, making it less general, less abstract, and less invariant than it ought to be. The resulting structure will inevitably lead to an ersatz design, often one with an excessive number of classes, with messy interaction. I suspect that one of the reason (and not a secondary one) why a lot of internet software is bloated (in memory occupancy) and inefficient (in execution time) is a poor structure due to the frequent disregard of this guideline. This point is so important, in fact, that the whole chapter VII will be dedicated to work out an example of its application.

*   *   *

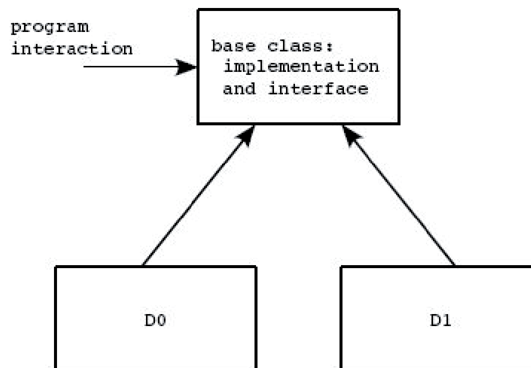
Derive implementation by private inheritance, interfaces by Interface implementation, subtypes by public inheritance.

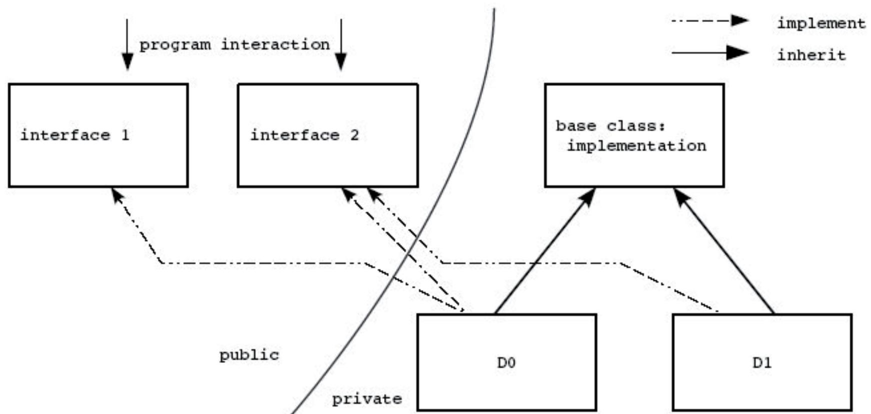
This rule also helps keeping implementations separated from interfaces, that is, this is yet another rule meant to help enforcing the most important principle of design: abstraction. A class *A* may inherit from a class *B* in order to use the methods of *B* for the implementation of certain functions, but the design of *B* was not done with *A*'s problem in mind so, except for special cases, the interface of *B* is not the best way to interface to *A*.

Classes that provide implementations should be abstract, with no public methods and no public constructors. This is the only effective way to guarantee that the public interfaces of a hierarchy will not depend on the implementation.

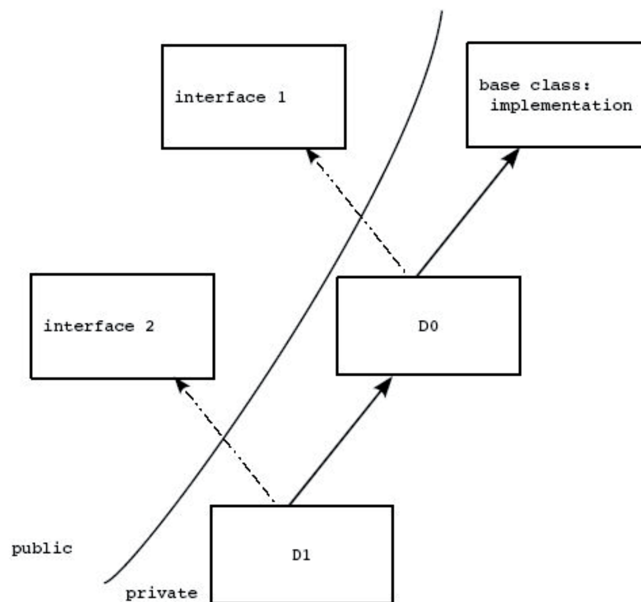
As we have just seen, interfaces should be based on the problem, not on the solution, so you should begin by designing an interface for the problem (creating an *Interface* Java entity) and then have the concrete class that solves that problem implement the interface. If you just inherit the interface of the base class, you will be in a situation like this:

In this schema, the user interacts directly with the interface of the base class, which was not designed for the tasks of *D1* and *D2* (The two might not even be the same: *D1* might require a different interface from *D2*, or a more extensive one.) A good programming style should separate the implementation from the interface, creating a structure like:





This way of operating separates the concerns of the user of a class from those of the developer: the user will not be concerned with the implementation class, but only with the abstract interfaces that the developer provides. In the case of a class hierarchy, the structure would be the following:



Inheriting public methods from a class is allowed only for is-a hierarchies, that is, for sub-typing, subject to the restriction on the precondition and postcondition of the derived methods that we have already seen in the previous chapter. Other dependences, such as hierarchies of type *is-implemented-as*, or *has-a*, must be implemented according to the schemas presented here.

* * *

The interfaces must be complete and minimal.

A complete interface allows the user of the class to perform all legal operations on the data, as defined by the semantics of the problem.

The interface should also contain as few functions as possible or, equivalently, there should be no overlap between the operations performed by different methods. Having different methods perform overlapping operations represents a risk of inconsistency when the class is changed for any reasons, and makes maintenance harder.

One exception to this rule is the implementation of partially redundant functions for the sake of efficiency in critical applications. This kind of redundancy should be used only when absolutely necessary. It should not be done based on guesswork on the desirability of efficiency, but only based on measurements done with a profiler, only if efficiency is a primary concern, and only if the efficiency requirements can't be met by a minimal interface. In any case, the situation should be well documented to help whoever will have to deal with the class in the future.

* * *

Write short methods.

All (or nearly all, see below) the methods that compose a program should be *short*. Now, the word *short* is one of those terms that can be stretched to fit anybody's aims, from those of the manager that insists that no printed material produced by the company shall contain sentences longer than 8 words to those of the pastor who, 40 minutes in his sermon, pronounces the ominous words "*to make a long story short*," hinting at 40 more minutes of the same. Nor, alas, is it possible to

give fast rules like the one that the manager above seemed to appreciate too much and the pastor seemed determined to ignore, whatever they might be.

If one were to make a parallel with natural language, one could say that the sentences of the previous paragraph were just a bit too long and convoluted (see the next point). But this would be just an example of violation of an unspecified rule. Stipulating clearly the rule that the example violated is a completely different problem.

In general, any function longer than 100 lines deserves a serious scrutiny to see if something odd is going on. A program that contains many functions of 50 lines or more is very likely to present some design flaws, and no function should be longer than 150 lines of code. As a practical gauge, most functions should fit in a single screen of the editor and/or in a single printed page. This is, of course, not always possible, but it should always be the case for methods with a significant structural complexity (see below), for which a general view of the flow of control is important in order to understand what the method does. All long methods (say, more than 50-70 lines) should have a simple and repetitive structure. The typical case of an unavoidably long method is one that contains a *switch* statement with many cases. In this case, however, it is important to isolate length from complexity: the function should be structured only around the long switch, and there should be no decision structures in any of the cases of the switch. That is, one should avoid a long *case* statement in which every condition has a complex code such as:

```
switch(c) {
       :
    case 3:
        if (a==0) {
            while (!n) {
                n = f(n);
            }
        }
        else {
           n = 0;
        }
        break;
    case 4:
       :
}
```

Here, all the complex decision structure should be placed in a well documented private method, and the body of the *case* statement should contain only a call to that method.

In a well structured program, large functions should be the exception, and short functions the norm.

* * *

Write simple methods.

Length is not the only concern when writing methods. Methods should break the structural complexity of a program into manageable chunks. That is, no method should have too complex a structure. Unlike length, it is not immediately obvious what the *structural complexity* of a function means. A number of measures of structural complexity have appeared in the literature and, as is always the case, all of them are acceptable and none of them is perfect, in the sense that it is always possible to find examples in which the measure gives a result on a collision course with the most vanilla of intuitions. Of course, the best filter to determine when a function is simple enough is the programmer's judgment, aided by a massive dose of experience. However, using standard quantitative methods has the nice side effects that the results can be replicated and that the responsibility for failure (it may happen) is diverted away from the programmer. Here I shall briefly describe one fairly reasonable and simple measure of complexity, the *cyclomatic* number.

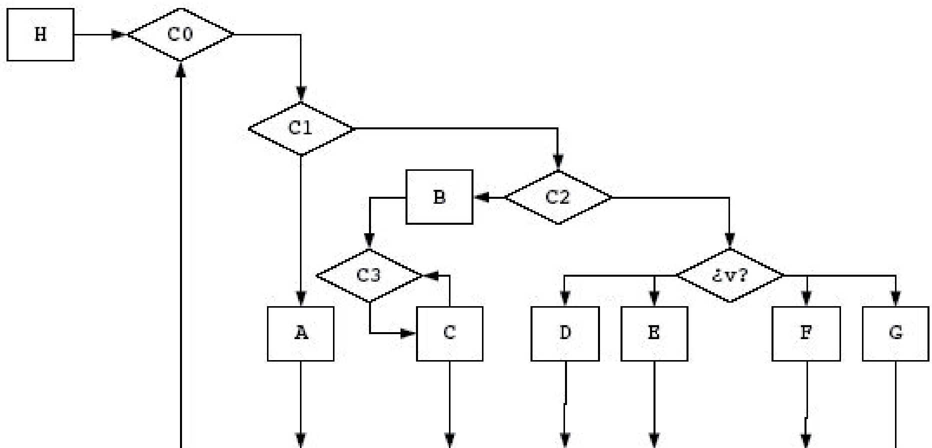
The cyclomatic number of a function depends only on the number and structure of the flow-directing operations (*if ... else*, *while*, *do*) and not on the presence and number of sequential operations. The number is defined as

$$c(G) = e(G) - n(G) + 2$$

where G is a graph representing the flow diagram of the function, $n(G)$ is the number of nodes of G (that is, the number of operational blocks in the diagram), and $e(G)$ the number of edges (that is, the number of flow transfers in the function). As an example, consider the following function skeleton. Since, as we said, the cyclomatic number is independent of the non-flow direction operations that the function contains[8], we will indicate the sequential blocks with letters that represent all the operations in the block:

```
while (cond0) {
   if(cond1) {
     A;
   }
   else {
     if (cond2) {
        B;
        while (cond3) {
           C
        }
     }
     else {
       switch (v) {
         case 1: D;
         case 2: E;
         case 3: F;
         case 4: G;
       }
     }
   }
}
H;
```

The flow chart of this function is the following:



The diagram contains $n=13$ operations and $e=19$ arcs, resulting in a cyclomatic number $c(G)=8$.

As in the case of the length, it is difficult to give clear-cut rules. A prudent strategy would advice to analyze closely every function with cyclomatic number greater than 15. A good way to do this is to ask a colleague participating to the same project, but not directly involved with the design of the function, to interpret it. A relatively large cyclomatic number doesn't necessarily imply a hard-to-understand function: as always, there are peculiar cases. Each instruction *if* placed in a sequence of instructions adds one to the cyclomatic number, so a function structured as

```
if (C1) A1=1;
if (C2) A2=1;
     :
     :
if (Cn) An=n;
```

will have a cyclomatic number equal to $n+2$, even though the function is by no means hard to understand. These special cases do not invalidate the general rule that methods with a cyclomatic number greater than 20 should be avoided.

*   *   *

Make every method either a procedure or a function.

A procedure is a method that changes the state of the object of which it is part, prints something, writes something on the disk, etc. A function is a method that computes a value and returns it without changing the state of the object. The two should never be mixed: if a routine modifies the state of the object, it should never return anything except possibly an error code (technically, there is no need to return an error code, since one can throw an exception, but sometimes it is just simpler to do so). If a routine returns a result it should not change the state of the object or of the process.

This division is a valuable help in writing a program because one is sure that a function will always return the same result independently of the number of times it

is called[9]. Changes in the state of the computation or in the state of an object have long term, non-local consequences, and it is important that these changes be kept under strict control. A function that gives a result that is not only a function of its parameter, but depends on how many times the function has been called, can be an enormous source of headaches.

Consider a typical example: a class contains a file, and a method to read the next record from that file. A poor way to implement this class is to use a method that reads the record from disk and returns it while, at the same time, advancing the file pointer to the next record:

```
Archive a;
String s;
   .
   .
while ( (s = a.current()) != NULL) {
    (do something with s);
}
```

Suppose that the part "do something with s" at some point changes s and, later on, needs it again in its original state. The temptation is great to implement something such as:

```
Archive a;
String s;
   .
   .
while ( (s = a.current()) != NULL) {
    (do something with s);
    s = a.current();
    (do something else with s)
}
```

Alas, the file pointer is no longer in the same position, and we end up reading and processing two records per iteration instead of one. The problem is that "current" is not a function, since it has side effects, but it is not a procedure either, since

we use it to return something. Even if one sees the problem, one still has to save a copy of the record, introducing a new variable only for this purpose: we now have two variables that point to what originally was the same object but later on is not, since it has been modified in the loop. This modified record that we don't need anymore is still hanging around—a situation that can easily lead to errors. The best thing one can do is to separate the procedural part of the operation (reading from the current position in the file) from the functional part (returning the value of the current record):

```
Archive a;
String s;
  .
  .
while ( a.read() != EOF) {
    s = a.current();
    (do something with s);
    s = a.current();
    (do something else with s)
}
```

The separation of procedures and functions has made the code easier to read and less error-prone.

* * *

Do not allow the automatic generation of member functions.

The class interface is one of the crucial elements of the design, and you should have complete control over it. Never delegate this essential function to automatic tools that create methods. This is true especially because such tools typically create precisely those methods that should not be placed in an interface without careful consideration: methods that read and set the value of the class variables. A class in which all variables can be read and set is *de facto*, if not *de jure* a structure and, as we have seen, structures should be used only for agglomerations of variables that provide no abstraction and maintain no invariant.

As a matter of fact, if you find yourself writing a class with methods that set and read all variables, you should seriously question your design. You are probably in the presence of two conceptually separate elements, a structure and a class, that ended up mixed in the same programming construct. In other words, in most cases automatic tools will invalidate your design. Do not use them. They may seem convenient, but in the end they are a source of significant design and maintenance costs.

*   *   *

Do not overuse libraries.

By and large, the idea of collecting useful programs in well proven, general libraries is a good one: programming would be impossible if, every time we had to write a program, we had to re-implement everything, from the file open function up! Software reuse, made possible by subroutines, is one of the early conquests of programming, and makes our life so incomparably easier that to question it, as a principle, would be utterly foolish.

Like many good things, however, software libraries can sometimes be overused beyond their point of usefulness, leading to negative effects that exceed the positive ones. We must keep in mind, once again, that a good program structure should always be our first consideration. In an object oriented design, as we have repeatedly said, the structure of the program must reflect the structure of the problem domain, and the methods defined in the classes should reflect the important actions that the entities in the problem domain perform. Libraries do not come from the problem domain, are not modeled after its structure and this, together with the previous observations, entails a basic library use principle: *the use of a library should in no way affect the structure of the program that uses it*. If it does, do not use the library. This principle, alas, is not always respected, especially in object oriented programming. Things are somewhat easier in structured programming languages such as C or in functional programming languages such as Haskell since they have only one mechanism to connect a program to a library—function call—, and this mechanism is *local*: a function call affects the structure of a program only in the point where the function call is made. One can design the functional structure of a program based on the problem at hand and then do the opportune library calls based on the *implementation* of the functions.

The equivalent object oriented solution would be to implement the library as a series of specific, locally instantiated objects. Objects of the problem domain will include objects of these classes as part of their invariant and use them by executing the appropriate methods. The locality principle entails that library objects should be internal to an object of the program, in particular, that they shouldn't appear in the interface of a class nor should they be passed from one object to another unless this is absolutely necessary or the library object is also a part of the problem domain. If locality can't be guaranteed, it is likely that the library will disrupt the structure of the program, and its use should be seriously reconsidered.

The main problems in the use of object oriented libraries derive from the use of interaction methods other than object instantiation/function call, interactions that are inherently non-local. There are libraries, for instance, that require that some class of the program inherit from a library class in order to access some functionality. Inheritance is not local in the sense that I have just expressed, since it is a relation between two classes and not between two objects. What is worse, inheritance is a structural relation: to force one class to inherit from another means to alter the original structure of the program, destroying the isomorphism with the problem domain. In these cases, unless one really needs to, it is better not to use the library: one should always evaluate whether the advantages of using a library outweigh the considerable loss of quality deriving from the alteration of a program's structure. One should *always* analyze with a healthy dose of skepticism whether to use a library or not. Libraries are often abused so, next to the advantages of their use (which are too obvious to be reminded here), one should consider the possible disadvantages:

i) the library might not implement exactly the functions that we need, forcing us to alter our algorithms in order to adapt them to the library;

ii) the library might do more than we need, forcing us to include in our program a lot of code that we don't need, thus making the program bigger;

iii) the library might be of poor quality, not thoroughly tested, decreasing the quality of our own code; one should only use libraries whose quality criteria are publicly available;

iv) we have no control over the library code; we can't optimize it, nor adapt it to the changing requirements of the future versions of the program (in extreme cases, libraries may hold programs back, preventing them from implementing additional functions because the library does not allow them);

v) sometimes we have no control over the library's distribution and we can't bundle it with our code, resulting in complex installation procedures (in order to install A you first need to install B and C and, for these you need D,... in this case the only reasonable thing to do is not to use the library: there is no excuse for using code that one can't bundle with one's own, no matter how useful it is);

vi) learning to use the library might be more complicated and time consuming than implementing the functionality that we need.

The abuse of libraries (and their poor design, which I shall consider in the next chapter) are significant factors in the inflating size of program and in their decreasing quality. There are a number of causes for this abuse: the illusion that the use of a library will invariably result in a quicker release, the relinquishing of rational thought to a design process driven by empty maxims such as "do not reinvent the wheel", and the education of programmers to be assembly technicians rather than abstract mathematical thinkers. All these habits need to be changed if we want to design reliable programs, a point to which I will return in the last chapter.

* * *

Define classes of high cohesion.

Coupling and cohesions are qualitative properties of any program that is divided into modules. In object oriented design, we find this concept at two levels: the division of the overall program into classes, and the division of each class in methods. The principles are more or less the same.

Cohesion is a measure of how a program module "fits" together. Typically, one class should do one, and only one thing, rather than assembling willy-nilly a number of unrelated things that the programmer needed to be done at a particular design stage. The presence in a class of many methods with low cohesion is also a harbinger of trouble in that it is a fairly reliable sign of design problems. If only a few methods have low cohesion, the best solution is probably to re-design the class interface. If many of the methods have low cohesion, then the program design should be reviewed, as the class itself must probably be eliminated or broken up. Normally, one recognizes seven levels, reported here in increasing order of cohesion, that is, from the worst to the best.

coincidental cohesion	(low cohesion--worst)
logical cohesion	
temporal cohesion	
procedural cohesion	
communication cohesion	
sequential cohesion	
functional cohesion	(high cohesion--best)

One has *coincidental cohesion* when the parts of a module have been assembled willy-nilly, or for reasons other than their function (e.g. a class *util* that contains methods used very often). One has *logical cohesion* when the parts of a module have been assembled because they are logically categorized as doing the same thing, although these things differ in nature (e.g. a class containing I/O functions). *Temporal cohesion* is that of a module assembled because its parts are executed at a particular time during program execution, while *procedural cohesion* is attained when the parts of a module are grouped because they are *always* executed together following the same sequence in the execution of the program (e.g. a function that verifies the existence of a file, access permissions, and opens the file).

One has *communication cohesion* when the part of a module have been grouped because they operate on the same data (this is the cohesion typical of a class, as all its methods operate on the invariant of the class), and *sequential cohesion* when the parts are grouped because the output of a part (or a portion thereof) constitutes the input (or a portion thereof) of another part. Finally, one has *functional cohesion* when the parts of a module have been assembled because they contribute to a well defined task of the whole module.

The scale is not linear. Studies[10] indicate that incidental and logical cohesion leads to poorly designed software and high maintenance cost, and that communication and sequential cohesion are already sufficient for a good software design. In object oriented design, it should be normal for every class to have communicational cohesion but, in a language like Java in which functions can be defined only as part of a class, this might be impossible. We have already seen that the class *Math* of the standard Java library, has what turns out to be logical cohesion. Due to the limited expressivity of Java, the presence of such classes might be unavoidable but, in this case, one should keep in mind the following rules:

i) logical cohesion should be obtained in any case;

ii) a class with logical cohesion should not carry an invariant, and be composed only of static methods;

iii) the presence of these classes should be limited to the absolutely necessary, and well documented.

The second rule will help isolating classes with low cohesion in such a way that they will not interfere with the overall design of the program. A class with an invariant should always have, at least, communication cohesion, that is, all its methods should operate on a class invariant.

* * *

Define classes with low coupling.

Coupling is a measure of the interaction between methods of the same class. In an object oriented design, methods of a same class are expected to be quite strongly coupled through the class invariant, since this is necessary in order to give the class high cohesion. In this discussion, I shall factor out this form of coupling, which is desirable, and, when talking about coupling, I shall only consider forms of coupling other than the class invariant.

In this sense, the coupling between the public methods of an interface should be zero. We have already said that the interface of a class should be minimal, a requirement that rules out all kind of coupling between its methods. A method of the interface should never call another method of the interface. If the two share some functionality, the two should call the same underlying private method.

Coupling between private methods (or between an interface method and a private one) should be minimized by minimizing data sharing. Methods should share only the indispensable data, and access to these data should be minimal. The literature reports eight forms of interaction between functions which we report here in descending order of coupling (viz. from the worst to the best):

Alteration of the other function's code (high coupling--worst)
Entry point other than the normal function's entry point
Access to private data of another module
Access to shared global data
Parameter passing by reference with possibility of change
Function call with a function selection parameter
Parameter passing by value, pure data
Data pipelining (low coupling--best)

The first two interactions are (fortunately) impossible in all high level languages, although they must be considered when programming in assembler.

Access to data within another module is distinguished from the case of shared global variables in that, in the latter, the module that defines the global variables, and all the modules that use them, do so with the knowledge that they might be tampered with unexpectedly, while in the case of private variables, the module that declared them presumably doesn't expect them to be changed from outside. Access to another module's private data must be avoided, and that's it. On the other hand, global variables may sometimes be useful, and there are cases in which their intelligent use produces a better code than their absence. Keep in mind, however, that they must be used with parsimony, and that they should be "cold": they should be read much (much) more often than they are written[11], and they should not be written as part of the main execution flow of the program but wonly hile recording some special situation that must be broadcast. Global variables, in a sense, should be used like television: only one (or few) channels fill in the content of the programs, and they are broadcast to a large number of viewers.

Data passed by reference follow the same recommendation, although here the situation is better, since, in any case, the data are shared only by two modules, they are shared only through one interface point, and with full knowledge that they might be changed.

Data passed by value are the optimal form of interaction. We must remember, in any case, that the coupling between two methods is lower when fewer parameters are passed.

Data pipelining is the weakest form of coupling, especially if the data are sent and received using some standard protocol that makes no hypothesis on the internal

data representation of the modules involved (e.g. if the data are transmitted in XML format written and read a file). In this case the modules can work asynchronously, without having explicitly to transfer control to one another, and without being connected by the constraint of being written using the same programming language and with the same calling conventions. Java offers no native support for doing this kind of transfer without efficiency penalty (e.g. the possibility of writing pipelined code that then, during compilation, is translated into normal function calls) and the only possibility of implementing a pipelined coupling is by manually managing independent threads, a condition that limits pipelining to rather large, high level modules.

* * *

Be wary of classes with many static methods.

Static methods break the message interchange model of object orientation, according to which messages are always exchanged between objects, and never between classes. Classes are only abstract universals that describe the behavior of certain groups of objects. They are like the design plan for a type of object, or the Platonic ideal of objects but, just like the Platonic ideal, they do not exist in the world of experience: at run-time, only objects exist, not classes. Static methods break this separation between description and instantiation, making classes appear in the run-time environment. This is a price Java has to pay for the absence of free functions. While in some circumstances avoiding static methods altogether might be so awkward that a violation of the strict object orientation methodology is better than the alternative, they should be used sparingly and only in well justified circumstances. Whenever one sees a class composed in large part of static methods, one is almost invariably in the presence of a set of functions haphazardly assembled in a class. One is, in other words, in the presence of a class with very low cohesion and scarce "identity" in the problem domain: remember that in object oriented design it is *objects* that have behaviors, not classes: classes only define the structure of the problem, that is, the types of behavior that objects can tokenize. Whenever a class with a lot of static methods appears in a design one can be almost sure that somewhere in the design a proper class structure failed to materialize, with classes that, had they been defined, would have implemented the same functions as regular methods of their objects.

Examples of this situation can be found in many Java libraries, sometimes because of limitations of the language, sometimes because of bad design. The `Math` class in the basic Java library, for instance, is a jumble of functions that defies any sensible association with objects. (`Math` is, needles to say, not a data type in any problem domain, unless one is putting together a schedule management program for a high school.)

The *objects* with which one has to deal when doing mathematics are (among others) *numbers*, while functions can be seen as behaviors of numbers (single argument functions, at least: the general concept of function is difficult to encapsulate in an object oriented framework *ex hypothesi*). The design options, at this point, are two: either one considers functions as objects (which would require to define classes such as sin, cos, log, etc. with methods such as *apply*) or consider function computation as a behavior of numbers. To me, the latter seems the most correct way of proceeding (a number "knows" how to compute its own logarithm), so one should consider numbers as objects and functions as their methods. The only price to pay for this solution is that the computation of multi-value functions would require the definition of the Cartesian product of number sets as a class. The proper way to compute the logarithm of a number would then be:

```
float x = 2.71;
float y = x.log();
```

or

```
float y = (2.71).log();
```

One might object to this solution on efficiency grounds: defining numbers as objects is just too cumbersome and inefficient a way to go. But, of course, floating point numbers are part of the basic typing system of the language, so there is no reason why this way of considering number should make it into the compiled code: numbers and other native data types can be defined syntactically as objects and then implemented functionally. The previous equation can be transformed, during compilation, in

```
float y = log(2.71);
```

Things are more complicated for multi-value functions, for which one should define some suitable Cartesian product class. I am thinking something along the lines of:

```
Product c = new Product(2.71, 3);
float y = c.power();
```

or

```
float y = new Product(2.71,3).power();
```

This way of doing things is obviously too cumbersome, and one can't really propose it as a way of implementing multi-argument functions. The only real solution in this case would be an amendment to the language that allows the definition of free functions: functions that are not the method of an object or a class. Since Java doesn't allow the definition of these functions, sometimes the definition of static methods will be unavoidable.

However, one should always remember that they represent an ersatz solution, and should use them sparingly, cautiously, and their use should be very well documented.

* * *

Be wary of classes for which you can't find a simple name.

A common tell-tale sign of a poor object oriented design is the difficulty of finding, in the problem domain, good *names* for classes and methods. This consideration might appear quite bizarre at first, and you might feel inclined to ascribe it to a personal fixation of mine with language, but the idea will start to make more sense if you think for a moment about the process of emergence of isolated words in human language, and of the way in which the foundations of object orientation rest on the results of such a process. Human languages evolved to organize and direct our experiences[12], and words are elements of the same granularity as the concepts on which this organization is built. Words are not too specific (there is no single word to denote a specific dog as it appeared yesterday at 4 in the afternoon, except

in the vocabulary of Borges's *Funes el memorioso*), not too heterogeneous (there is no single word to indicate all things that are yellow, fast, and less than three feet tall or loud and built on Tuesday), and modeled on the environment from which they arise (reputedly, Inuit contains more than 20 words for snow, while Bengali uses the same word for snow and ice). One aspect of the creation of words that is quite remarkable—and one of the bases of my thesis here—is that, although there are deep differences in the grammar of human languages (the presence or absence of prepositions, the existence of syntactic markers—such as the Japanese *wa* that indicates the subject of a sentence—the structure of verb tenses and so on), the *kind* of things that are designed by words, and the granularity at which the words divide the world are surprisingly uniform. Words denote all sorts of different things and, in different languages they support different taxonomies, but they are never too specific or too heterogeneous.

Object orientation is based on the identification of important objects which, as per the previous method, are the components of the problem domain denoted by nouns and, therefore, are of the same granularity of words. If the problem domain is mature enough to have created a technical language, this language evolved to indicate precisely those things that object orientation should consider as the basis of design. Since the domains in which we work are often very specific, it is often the case that we must bend the rule a bit and, instead of identifying an object with a single noun, one might want to consider entities designated by a single adjective form, paralexeme, or syntagm. This caveat standing, it is still the case that the impossibility of telling what a class is in the problem domain, and of giving it a proper name often indicates that the "thing" that we are trying to model is either too specific, too heterogeneous, or that it doesn't belong in the problem domain at all. As a rule of thumb (to be taken with the usual grain of salt), I would surmise that every class with more than two words in its name, more than one noun, or more than zero verbs, is too specific; every class whose name ends in "-able" or "-ator" (with the exception of classes such as "alligator," of course!) is too heterogeneous; every class whose name ends with "-manager", "-server" or suffixes of this nature is an intrusion of the implementation in the design (again, with the exception of design in which managers and servers are part of the problem domain, such as in the model of a bureaucracy or of a feudal citadel). Of course, everything to which one has to add "-object" to make it into an object name is not, *ipso facto*, an object.

Often, the consequences of poor design are baffling interfaces at the limit of comprehensibility. A prototypical example of this is the *Image* package in the

original Java user interface (*java.awt.Image*), but examples abound. Classes like `AffineTransformOp` or `StringSelection`[13] are clearly derived from a functional design (and a poor one, for that matter). The event model in the *java.awt* package also derives from a rather transparent desire to implement "callback" functions, a technique very useful in functional programming (the C library for X windows does that) but out of place in an object oriented model: interface objects are expected to manage their own behavior, and special behavior should be implemented by inheritance. Many classes of the Java library are of this type and, maybe taking off from these examples, this strange hybrid of object oriented and functional style has been used in almost all Java designs. The results for the logical coherence of the designs have been, needless to say, disastrous.

VI. DESIGNING LIBRARIES

Among the many changes that the internet and the open source software movement have brought to programming is the proliferation of software libraries. As little as twenty years ago, the term "library" meant essentially two things. On the one hand, a collection of support functions that came with the compiler or were closely associated to it, often distributed by the same vendor as the compiler (as in "the C socket library"). On the other hand, one of the large and often expensive packages that one could buy to perform important and delicate functions in a program. The prototypical example of this second sort is the linear algebra library LINPACK. The library has been developed over decades, starting with the early versions in FORTRAN. It has been refined, and well documented; its algorithms have been improved by skilled mathematicians, and its code optimized by skilled programmers. Most programs that depend heavily on linear algebra use the library simply because it would take a dedicated team years to develop something of comparable quality.

This pattern of production and usage was changed by the availability of the internet as a distribution medium, and by the emergence of a large community of developers that release products as open source. Today, the average software library—especially the average Java software library—is much more fine grained than it was twenty years ago, often performing much simpler (bordering occasionally with the banal) functions. The release cycle has sped up considerably, sometimes in order to introduce new features that the developers found interesting, sometimes in order to keep up with emerging standards. A consequence of these phenomena is that, as we have seen in the previous chapter, answering the question *"shall I use a library here?"* has become more complicated today than it was twenty years ago.

But these same technical-social transformations have also brought about changes in the way libraries are developed, and not all these changes have been for the

best. We have seen in the previous chapter the problems that improperly designed libraries may cause to an application program: fragmentation, structural inconsistency, loss of quality. All this doesn't mean, of course, that one should not take advantage of the new design possibilities that the internet and the open source give to the library developer. It *does* mean that the library developer needs a new discipline of design style. This chapter attempts to give a few guidelines in this sense.

* * *

Consider not developing the library.

E. Dijkstra wrote that the first thing to do when one had a research idea was to try to destroy it: only ideas strong enough to resist this attempt were worth pursuing. Something similar happens with a library. Developing a library represents an effort an order of magnitude greater than developing the same functions as part of an application program, and this is true especially in the design phase. (At least, it is if you do it right.) Before embarking on such an effort, the development team should verify that it is warranted, by subjecting the idea of the library—still in the requirements stage—to a set of sieves. Some of these sieves will be non-technical, depending on marketing strategies, corporate culture, etc. Only a library idea that passes all these sieves and that satisfies the criteria on which they are based, is good enough to be implemented. Here, I will only consider the most obvious technical criteria.

i) The library should solve a clearly identified problem. If you can't tell, simply and concisely what the library is about, then you are probably looking at an collection of functions assembled willy-nilly, good for a specific application, but that don't belong together in a library.

ii) There should be either a large community that saves considerable work using your library or a small community that saves a tremendous amount of work. It doesn't make sense to spend time and effort designing a library if the total amount of work that it can save is not very large.

iii) There should be an independent, extended in time, need for the library. Designing a library represents a long time commitment to improvement, documentation, and maintenance. It should not be undertaken for ephemeral problems.

An adversarial approach works fairly well in these cases. In the decision meetings, some members of the group should defend the decision to develop the library, while others should be given the rôle of devil's advocate and find arguments not to develop it. I used to give the winning group a small award to make sure that they took their rôle seriously. If you are alone in the decision, dedicate some time to list the reasons why you should not develop the code in a library. The decision to develop should always rest on positive arguments: if you think you could go either way, your best option is probably not to create a library.

* * *

Implement a compact and coherent set of objects and behaviors.

Expect problems if your library lacks a compact and coherent design. If the library doesn't identify a well defined object (or a cohesive group of objects) whose behavior it implements, the risk of implementing too little or too much is significant: the organization of the library is liable to reflect a contingency that occurred to a programmer and that is not likely to occur to another. In this case, it is quite probable that the library designer will fail to implement some function that the user would need while wasting space to implement something that most library users will never need.

To make an example, a logging library with extensive file management functionality is more than most users will need and will lack some functions that a user interested in, say, logging on a network might need. The same goes if the library has functions to manage error codes, which is something logically distinct from logging, and that, the way it is done in the library, might not fit the needs of many users. It is better to design a library that does only what it promises: log messages to an output channels in an efficient way, possibly buffering the messages, printing them in background, dealing with channel failures, and so on.

* * *

The library must be of better quality than the programs that use it.

Keep in mind that your users, will have no access to your code or, if they do, having to change it will pretty much voids the advantage of using the library. The operations of a library must be absolutely flawless. Your typical user will always assume that the library works well and, if there is a problem, will spend a lot of time looking at his own code before pointing the finger at the library, not to mention that checking whether the library contains a bug might not be an easy matter. It is necessary that a programmer have a great confidence in the libraries she uses; thus, it is necessary that only absolutely correct libraries be released. The quality criteria adopted for the library, the methods used to determine that the library satisfies the criteria, and the result of the application of the methods should be published and included in the library documentation. A library that can't quantify its own quality is not to be trusted.

* * *

The program should control the ancillary operations, not the library.

A library should not impose any operation, file, or file format that is not part of its function. Unless a library deals with communication, it should not impose any communication protocol; unless it deals with files, it should not impose any constraint on the way a program defines or uses files, and so on.

A few years ago, in a development I was supervising, we were trying to *log* certain activities, and one of the programmers proposed the use of a Java logging library that he had used before. I looked into the library and noticed that, at start-up, one had to pass the name of a file from which the library would read its configuration parameters. That was enough for me to veto its use, and we decided to implement our own logging functions: it just so happened that, early in the design, I had decided that the program should have only one configuration file, in which the parameters for the whole program (including those that configured the logging function) were to be specified. Using the logging library would have imposed the presence of a *second* configuration file in a different format (the library used XML, which usually makes no sense for a configuration file; our configuration file was in a simpler and more compact format). Had the designers given me the option to con-

figure the library with a series of function calls with the values of the parameters, I would have considered using it. As it is, we designed our own functions in less than a week, they did just what we needed them to do, and we had control over the code. In the end, the presence of the configuration file resulted in a benefit: it gave me a reason to veto a poorly designed library and to implement something smaller and better.

As a further example, consider a library that implements a parser that takes an input in some formal languages and builds a parse tree. The result (the parse tree) is an object and therefore, as we have already seen, the correct mechanism to invoke the parser and build the tree is through the constructor of the tree class:

```
Parse_tree t = new Parse_tree(String expression); (†)
```

Note that we pass the input in a string and not in a file, for the same reasons highlighted in the previous example. What happens now if the data are not available *in situ* but are received through an input channel in the form of a data stream? The library designer might have the temptation to write a constructor such as:

```
/* wrong! */
Parse_tree t = new Parse_tree(InputStream is);
```

This is incorrect, since it would require the library to deal with something other than its core function, which is parsing, and not receiving data streams. The caller should take care of collecting the data before calling the parser. We can, of course, offer some help, without going out of the core functions of the library, to make life easier for the programmer. Here is just an example. We can begin by creating a tree class with simply a fragment of the input[1]. The constructor will not change, and will still look like in (†), but the tree will not be a valid parse tree unless the string *expression* were a complete sentence in the input language. The class tree, of course, must provide a method to verify whether this is the case:

```
boolean t.valid();
```

If not, we can add fragments of the input sentence as they arrive:

```
void t.add(String fragment);
```

and, at any time, we can force the validation of the tree, by forcing the parser to parse the input string:

```
boolean t.validate();
```

This is just a partial and incomplete example, but it gives the gist of how a library can serve different needs without imposing unnecessary constraints to the calling program.

* * *

Your library should not affect the structure of the program that uses it.

The interface to a library must be a small set of classes that will be instantiated within an object of the calling program and that will be used by calling suitable methods from within the same object; I have already called this the principle of *locality* of the library interface. As we saw, this entails that one should avoid all interaction with a library based on inheritance. Interaction using the implementation of an interface is in general quite complex, and should be limited to cases in which the desired functionality can't be implemented using function call (e.g. an interaction based on interfaces might be necessary in order to implement a *call-back* mechanism). Interaction based on inheritance should always be avoided in a single inheritance language such as Java.

No library object should be transferred between objects of the program. Admittedly, this principle may be impossible to implement fully: sometimes it is just too convenient for objects of the calling program to pass library objects to one another. It is, however, a useful guideline: if there is a way to avoid passing library objects around in the calling program, the library interface should favor it. For instance, instead of having a method that returns an object of class

```
class Person {
    String name;
    int  age;
};
```

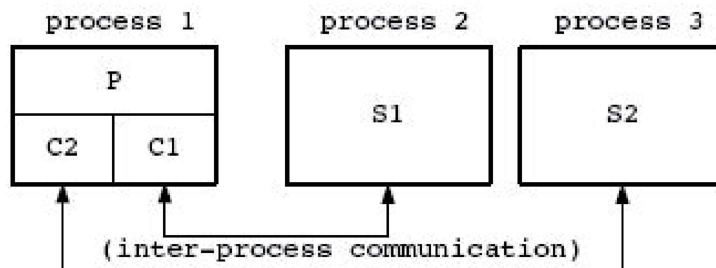
(or in addition to it) the library could have separate methods for obtaining name and age. Then it would be the calling program that, if necessary, could define the class `Person` and fill it with the data. This will give the program a greater flexibility whenever the whole class person is not necessary or if the data are part of a larger class.

* * *

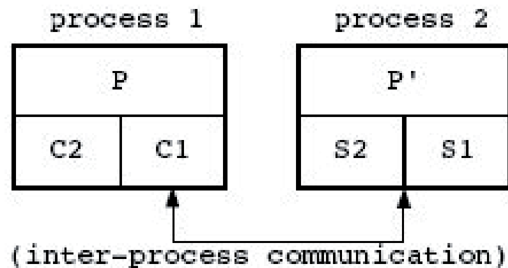
The library should reside in the same address space of the calling program.

Just like the interface of the library should not constrain the static (class) structure of the program, so the run-time model of the library should not constrain its dynamic (process) structure. A library should not require the creation of processes, servers, clients, or impose any division of the process and address spaces of the calling program. Any part of the library should consist of a number of object files (a jar file in the case of Java) that are simply *linked* to the calling program at compile time, that become part of the same executable file (the same jar file in the case of Java), and that are loaded in the same address space at run-time.

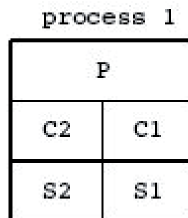
For example, there are libraries composed of two pieces: a client, which is attached to the calling program, and a server, which is executed as an independent process, launched before the program, and running in a separate address space. This is a poor organization for a library, because it limits the freedom of the programmer to design his own process structure. Let us say that the program uses two such libraries: one with client C_1 and server S_1, and one with client C_2 and server S_2. Each one attaches its client to the calling program P and requires its server to run as a separate process. The only possibility for the designer of P is to adopt a three process structure like the following:



But, depending on the requirements of *P*, it might be more convenient to adapt a two-process structure, in which the two servers are part of the same process, controlled by a program *P'*:



or a single process solution, in which everything runs in the same address space, the calls to the server are blocking, and interprocess communication is replaced by function call:



This organization may entail an execution time penalty *vis-à-vis* non-blocking inter-process communication but this might not be a problem in the case of *P*, and the greater simplicity of function calls might more than make up for this limitation. This decision should be taken by the designer of the program, and not imposed by the organization of the library. A library (like the ones in this example) might still be composed of two parts (viz. a server and a client), but the server should not be designed as an independent, executable process. Rather, the server should be a collection of objects that communicate with the calling program through suitable function calls. It will be the designer of *P* who decides whether the server should be implemented as a separate process (in which case he will design a separate program) or whether it should reside in the same address space as the caller.

This solution can also be as a logical consequence of the coherence principle that was discussed a few pages ago: the library C_1/S_1 is designed to accomplish a certain function, and it should stick to that function. Unless its purpose is inter-process communication, the library should not be cluttered with functions it doesn't need.

* * *

Don't mess up installation by retaining too much control.

There is a nasty habit, common especially in the Linux milieu: many times you find a program that you would like to install, program *A*, say, and you discover that, in order to install *A*, you first have to install program *B*, that, in order to install that, you first you have to install program *C*, and so on, until you run out of letters of the alphabet or of patience, whichever comes first. In my case, patience is always the first victim and, as a matter of personal policy, I refuse to use any program that is not completely self-contained. Sometimes, the responsibility for these flaws falls squarely on the shoulders of library producers who made it impossible (either technically or legally) to include their library in the same executable as the program that calls it.

If one follows the previous guideline, there is no technical reasons why a library should complicate the installation of a program: including the library in the same address space as the calling program means that they can be linked at the time of compilation, and that they can become part of the same executable (or of the same jar file) as the calling program: the responsibility of how the executable or the jar files are organized and installed should be solely of the program designer.

The matter is more delicate if the reasons for the separate installation are legal, of "branding", or something along those lines. This is essentially a technical book, so I will not enter into the *desiderata* of the marketing department. Those guys might have their own reasons to want that everybody that uses the library visit the company's web site in order to download it and, from their point of view, their reasons might even be valid. The problem is that their marketing reasons decrease the technical quality of their product, and any company in which marketing is enforced at the expense of technical quality is not a place where a serious programmer would want to work. Better leave the library and start working on your resume. I did it a couple of times, and I never regretted my decision.

VII. THE CASE OF CONNECTIVITY

In this chapter I will take a look at the design of a standard library that the Java culture has popularized for Internet programming. While in the last chapter I took a rather general point of view, covering several broad classes of design problems, in this chapter I will look at a specific example: the so-called *Java* data base connectivity (JDBC)[1].

I will begin with a brief description of the interface, in which I will highlight the points where the design differs from the structure of the problem under consideration (that of sending queries to a data base and receiving back results), and how these departures result in a confused (and confusing) interface. Then, I will analyze the problem domain and, based on this analysis, propose an "amended" version of the interface whose structure is closer to that of the problem domain. I will try to point out how the structure of this solution is more natural and can be understood more easily by the average programmer who knows what the purpose of the library is. There is a problem in my choice of an example: the library that I am analyzing and its interface are very well known to the vast majority of Java programmers. On the one hand, this is an advantage because it it eliminates the necessity of a thorough introduction to it but, on the other hand, it is a problem because I expect that most of the people who read this book will be so accustomed to the interface that they will have been, in a sense, "blinded" to its quirks. In other words, certain characteristics of the interface that, by all reasonable accounts, are rather bizarre, have by now been made familiar by use and may not seem so strange to many of the readers of this book. I have to ask you to make an effort to see the interface to the library as it is in reality, as objectively as possible, trying to forget that you know its classes and methods very well, and trying to put yourself in the shoes of the programmer that encounters it for the very first time.

My proposed library, of course, should be taken as nothing more than a pedagogical experiment: I am not implying that my design should be adopted in lieu of the current data base connectivity interface, even though I consider it superior. Its pedagogical nature will force me to be incomplete, and many functions that should be present in the actual interface will not be present in my design. I do hope, however, that in some future version of the interface somebody will follow some of the design guidelines proposed here.

*   *   *

The Java data base connectivity interface is a standard interface for libraries that allow programs running on a given computer to use data bases (especially relational data bases) that run on the same or other computers. As an interface standard, of course, the implementation of the library is not specified, nor is it specified how the library should communicate with the data base: the data base connectivity standard merely prescribes the classes and methods that the library must make available to the programs that intend to use it. The best way to start our analysis is therefore a simple description of the interface as it is. I shall assume that the reader already had a certain knowledge of the matter, so my treatment will be rather superficial.

The first thing to do in order to use a data base is to create a *connection* to it. This is done through a call to a method of the static class *DriverManager*:

```
Connection c = DriverManager.getConnection(String spec);
```

where *spec* is a string containing the connection parameters to a remote data base, including the internet address of the computer on which it runs, the name of its driver, and so on. Once a connection is established, one creates a *statement*, which, as the documentation reminds us: “is used to send your SQL statements to the DBMS and should not be confused with an SQL statement”[2]. Therefore, we execute the function call

```
Statement s = c.createStatement();
```

Sometimes, instead of a statement, one creates a so-called `PreparedStatement`, which is a string in which certain parts are replaced with a "?" symbol, so that they can be assigned a value later on using an appropriate call. So the fragment

```
PreparedStatement p = new PreparedStatement("This is a ?"
                                            +  "statement.");
  p.setString(1, "beautiful");
```

replaces the first occurrence[3] of the symbol "?" with the string "beautiful" resulting in the statement

"This is a beautiful statement."

Coming back to the regular statement, one can use it either to execute an update operation or a query operation. In the first case one will execute:

```
s.executeUpdate(String upd);
```

where *upd* is a string containing an SQL update statement (such as like a *create table*, *insert*, or something in that family). If one wants to execute a query, one must call a different method, which returns a query result of type *ResultSet*:

```
ResultSet r = s.executeQuery("select name, age"
                  + " from table where age>40");
while( r.next() ) {
    String name = r.getString("name");
    int n = r.getInt("age");
}
```

Note that the *ResultSet* has an iterator associated with it and that, for each execution of the iterator, one can access the values of the columns of the result (in this case a string and an integer) by providing their names (in this case, "name" and "age"). This is the main way in which queries are executed and results are returned using the connectivity library. An equivalent way to obtain the results using a prepared statement is:

```
ResultSet r = p.executeQuery();
```

in which case it is not necessary to specify the query string because it had already been specified when the prepared statement was created. To these basic functions of the library, a few more are added for the sake of completeness. For instance, given a result set r, it is possible to "move" up and down the rows of the table with calls like:

```
r.absolute(n);
```

which returns the n-th row in the table, or:

```
r.previous();
```

which returns the row immediately before the last one that had been accessed, and so on. Also, *transactions* can be enabled so that all operations done on the data base will have no effect until the transaction is *committed* by calling a method of Connection:

```
c.commit();
```

This is, in a very concise form, the way in which the data base connectivity library is used[4].

* * *

My contention is that the design of this interface is flawed (from the point of view of object oriented design) in several important points. In order to see this, I shall start by following the method that I had outlined in the previous chapter, and give a short description of the problem.

> We want to devise a method to use a *data base* that runs on a *computer* that can be either the one on which we are running the library or another one. The library

> should allow us to ask *queries* and other types of commands to the data base, and to return *results* of these operations. Since we are assuming a relational model, each result is a set of *entries*, each one of which is composed of a number of named *attributes* (or columns); we want to be able to see the entries that compose a result one by one. Other aspects of the data base, such as *transactions*, should also be covered.

This description, although clearly insufficient to initiate a "real" design, gives us a general idea of the objects and the classes that are important in our system. We now analyze the actual Java data base connectivity interface in the light of this description. The first statement of the previous example:

```
Connection c = DriverManager.getConnection(String spec);
```

already contains two objects that are not part of the problem domain: the DriverManager and the Connection. The problem specification does not tell us that the data base is accessed through a connection, not that there exists something called a driver, much less something that manages drivers. In the DriverManager class, with its static methods, one can easily recognize one of those intrusions of functional design into object orientation that I have considered earlier. On the other hand, the specification tells us that we intend to use a data base. We have therefore two possibilities, both of which respect the model. One is to use the pair of statements:

```
Computer c = new Computer((String) spec);
Database d = new Database(c, (String) db_spec);
```

in which the connection parameters have been divided in two: the argument of `Computer` specifies how to connect to the computer where the data base runs (e.g. by giving its URL and other connection information), while the parameter `db_spec` contains information on how to use the data base in the given computer. (For all these parameters, I use the data type `String` out of convenience. In the

actual design one will probably have to specify several parameters.) An alternative is to dispose of the class `Computer` altogether, and create the data base directly:

```
Database d = new Database((String) spec, (String) db_spec);
```

The choice between these two solutions depends on the details of the project. In particular, if it is the case that the class `Computer` has some use other than creating a data base, then the first solution is indicated; if the class `Computer` has no other use than creating the data base, then the second solution is better, since it avoids the proliferation of useless classes.

The most troubling part of the standard connectivity interface is probably the `Statement` class, in particular the warning that the writer felt compelled to make (and with good reason) that "[the statement class] should not be confused with an SQL statement." There are many debatable design decisions in the library, with varying degrees of inadequacy, but I must admit that this particular one borders with madness: we are trying to model a problem domain in which an entity called *statement* is prominently present, and the connectivity library ends up creating a class with the same name, but that isn't quite the same thing. This is, the way I see it, courting disaster.

Another terrible cause of confusion is the difference in behavior between the `Statement` and the `PreparedStatement`: while, in one case, the `Statement` is created empty and the actual statement is specified (as a string) only at the time of execution, in the second case the statement is created when the `PreparedStatement` is created (possibly as a template, which is then completed with appropriate calls), and the execution method has an empty parameter list. It is also quite peculiar that, while the `PreparedStatement` is created in the canonical way, using a constructor, the `Statement` should be obtained in a way that makes no sense in the domain of the problem: from a connection. Why should a connection issue statements? I will also notice, as a curiosity, that a `PreparedStatement` when it is created is not prepared at all: in order to have it prepared one must fill in the "?" symbols that appear in it. There is absolutely no reason why the interfaces of these classes, with practically the same function, should be different. As a matter of fact, there is no reason why one should have two classes at all: a single `Statement` class will suffice. The class should have a constructor such as

```
public Statement (String spec);
```

The creation of a query string starting from a template is a problem of string manipulation, and not of data base querying, so the most rational choice for the designer is to have the constructor accept only complete strings, leaving the creation of the string to the program:

```
String x = new String("name");
String y = new String("john");
    :
String qs = "select * from people where "
            + x + " = " + y + ";"
Statement s = new Statement(qs);
```

This solution has the advantage of making the class `Statement` superfluous, since we can obtain the result directly from a query string, as we shall see shortly. Reducing the number of class is always an important design goal, so avoiding string manipulation in the library is doubly advantageous: it makes the interface more coherent, and it makes it smaller. However, the designer might decide to use the class `Statement` to provide some token expansion capability for the convenience of the programmer. If we do this, and since I have an intense dislike for positionally specified quantities, which can lead to bugs when things are shifted around in the template[5], let us change the statement so that the replaceable tokens will be marked, say, by a symbol beginning with '$', and let's have the replacements done based on the same symbol, leading to something like

```
Statement simple = new Statement("select * from table");
Statement complicated = new Statement("select * from table"
                                  + "where $one = $two");
  complicated.set("$one", x);
  complicated.set("$two", y);
```

There is no need to have separate statements for executing queries and "update" operations. As a matter of fact, we don't even need one. Let me explain this starting with the execution of queries.

A query is a way of producing a *result*, which is a class in our design. That is, a query is simply a *constructor* of results and, if we want to be consistently object oriented, we should maintain this characterization. So, the method that executes a query is simply the constructor of the class `Result` (which I will not call ResultSet because what we have here is a result, the structure of which—set or otherwise—is not important at this level of abstraction):

```
public Result(Database d, Statement s);
```

The simple query above, then can be executed as[6]

```
Result r = new Result(d, simple);
```

Of course, in the case of a simple query one might find the creation of a statement too cumbersome, and would prefer to have the possibility of specifying the query directly as a string. This can easily be done by defining a constructor for `Result` such as the following:

```
public Result (Database d, String query) {
    this(d, new Statement(query));
}
```

If we do not define the `Statement` class (e.g. because we decide not to provide any token replacement function), the constructor with this signature will be the only one of the `Result` class. The question now is what do to with the data base commands that are not quite queries. This is a case in which the personal inclination and sensibility of the designer play an important rôle. I like to keep the number of methods low, therefore I'd rather not have two separate ways of sending commands to a database and, since all commands, regardless of their nature, will return a result (in some case it will be just a success/failure flag), I would definitely use the same method in order to execute any kind of commands. So, a fragment of code like this one would create a table:

```
Result r = new Result(d, "create table...");
```

The alternative, namely to have an execute method in the data base class, is also valid:

```
boolean err = d.execute("create table...");
if (err) {
    String s = d.errorMsg();
    // Do something with it
}
```

But let me assume for the moment that we go with the first one[7]. We need to have a better look at the result class because, at this point, it has come to contain quite a few things:

i) a flag to determine whether the last action was successfully completed;

ii) a possible error message or other error specifications in case the action was not successful;

iii) if the statement sent to the data base was a query, the class must contain the results of the query.

Here too we have several possible alternatives. One is to keep the "result" class as a way to read the success/failure flag and the error message, and to resort to another class to store the actual results of the query. (We might call this class *Answer*, just to give it a name.) The second possibility is to keep all the results in the same class, a solution that has the advantage of reducing the number of classes in the interface and that therefore we shall adopt. The class `Result`, then, will have methods such as:

```
public boolean successful();
```

to tell us whether the operation was successful,

```
public String msg();
```

to give is the possible error message, and

```
public boolean has_answer();
```

to tell us whether a `Result` object has an answer attached (in which case it was produced by a query statement) or not (e.g. if it was produced by an insert statement or something of that nature).

We might also define additional utility methods, for instance a method that gives us a reference to the statement that created the results:

```
Statement s = r.statement();
Statement s = new Statement(r);
```

Now that we have an answer, we need to access its elements one by one, a situation that generates a minor semantic problem. The relational model works on *bags* of records[8], that is, on *unordered* entities. Therefore, it doesn't make any sense to try and access the third element of a result, to move up or down, because no order can be imposed to the contents of the `Result` class: the only operation that is allowed is to remove a record from the `Result` (returning it to the calling program, of course), and to check whether an result is empty. On the other hand, most relational data bases allow an *order by* clause in their query language that, in lieu of bags, returns a *list* of records which is, of course, ordered. I will ignore this issue in this pedagogical design but, the way I see it, the best way to solve the problem is to have a specialization of `Result` that is ordered. I will only consider bags here[9].

An answer is composed of two parts: a *schema* and the bag of *records,* that is, of labeled *tuples* (I will use the two terms, records and tuples, interchangeably). The schema contains, essentially, the names and data types of the entries in each record (which are, unlike the records in the answer, ordered: we can ask for the *n*-th element of the schema, as well as for the *n*-th element of a record), while the records contains the actual values of the answer. We can therefore define two classes: `Schema` and `Tuple` to deal with the elements of an answer. Schemas are read-only entities, so there is no need to create new ones, and we can get a reference to one from the result class:

```
Schema s = r.schema();
```

The methods to access the elements of a schema are rather straightforward. Some of them might be:

```
String[] names();
int column_no();
```

which return the names of the columns in the schema, and the number of columns, respectively,

```
String type(String s);
String type(int n);
```

which return the type of a column given either its name or its position in the schema,

```
String name(int c);
```

which returns the name of a column in a given position of the schema, and so on.

One important question at this point is how to obtain the individual records from the result. Let us start with the consideration that the program might want to keep copies of several records at the same time, so the solution used in the Java data base connectivity interface—to use a *next* function that sets a pointer to the next records (within the `ResultSet` object) and then allows access to the columns of that record—is not a good one: if one wants to keep several records in memory for some computing purpose, it is very inconvenient to have to build temporary storage and fill them with the columns of the record one by one; it is much more convenient to access a record as a single entity. So, what we need is, for each element of the answer, an object of class `Tuple` that the calling program can keep in memory as long as it needs. As usual, we might also want to give the calling program the possibility of accessing a *reference* to the records contained in the answer although, for a variety of reasons, I don't think that this is the best way of operating in this case. The correct way to build a record is through a constructor, something like:

```
public Tuple (Result r);
```

that takes a result, selects a random record from it[10] and builds a copy of it. Now that we have the record, we need to remove it from the answer, using a method of the answer class. The iteration that goes through the records of an answer would then be:

```
while (!r.empty()) {
    Tuple t = new Tuple (r);
    r.remove();
}
```

In addition to using the constructor, thereby making it explicit that the tuples are a new copy each time, the presence of a “remove” method gives the programmer control over the removal of the records from the answer. All this can be supplemented with additional semantics stipulations such as the fact that the record constructor will always return the same record for a given answer until the record is removed. That is, in the following fragment

```
Tuple t1 = new Tuple(r);
Tuple t2 = new Tuple(r);
r.remove();
Tuple t3 = new Tuple(r);
Tuple t4 = new Tuple(r);
```

one is guaranteed that `t1` and `t2` are copies of the same tuple, that `t3` and `t4` are copies the same tuple, but that, say, `t1` and `t3` are copies of different records. In the class `Tuple`, of course, one has methods like

```
Object value(String col_name);
```

and

```
Object value(int col_pos);
```

which can be used to obtain the values of the columns given their names or their positions. Since it is cumbersome in Java to convert from objects to simple data types such as integers or floating point numbers, and there is no analogous of the “void” pointer of C, one can define “shortcut” methods, such as

```
int int_value(String col_name);
float float_value (int col_pos);
```

and so on.

* * *

This pretty much completes the interface, except for a few secondary additions that only deserve, in this pedagogical setting, a superficial mention. For instance, in the Java data base connectivity interface the initiation and the commitment of transactions are executed through methods of the class `Connection`. This choice makes, of course, very little sense from a data modeling point of view since connections do not have transactions, nor is it their responsibility to commit them. In the current design all methods pertaining to transactions would be part of the interface of the `Database` class, where they logically belong.

To conclude, I will report a fragment of code that illustrates (again, without going into the details) a typical interaction with a data base using the design that I have developed in this chapter. The fragment goes like this:

```
Computer c = new Computer ("http://www.somewhere.net");
Database d = new Database ("some_database");
Statement s = new Statement ("select x, y, from "
                              + "table1 where a = $q");

      /* read the values to be filled in */
      int q = user_interaction();
      s.set("$q", q);

      Result r = new Result(d, s);
      if (!r.successful()) {
          do_something( r.msg() );
      }
      else if (r.has_answer()) {
          Schema sc = r.schema();
          for (i=0; i<sc.column_no(); i++) {
              System.out.println(sc.name(i));
          }
          while (!r.empty()) {
              Tuple t = new Tuple (r);
              r.remove();
              for (i=0; i<sc.column_no(); i++) {
                  System.out.println(t.value(i).toString());
              }
          }
      }
```

I consider this fragment of code much clearer and more logical, for a person who first approaches the connectivity interface, than the equivalent fragment using the current standard, not to mention the greater possibilities for the programmer offered by the copy semantics for the creation of the records and the possibility of controlling the removal of records from the answers.

Lest there be any doubt, I will state explicitly once again that the previous example should be intended as an example of a design method, and *not* as a proposal to replace the current data base connectivity interface, although I do consider my interface superior from the point of view of object oriented design. My goal here was to show how some standard Java interfaces (and the phenomenon is by no means confined to this one) are designed without a clear analysis of the essential elements and behaviors of the problem. I can't presume to know the decisions and the requirements that shaped the design of this particular interface, but it seems obvious that little attention was paid to the creation of a correct model of the problem, for otherwise useless classes such as the `DriverManager` and the `Connection` would never have been created. Even more baffling are questions like the presence of two statement classes with radically different interfaces and the presence of different method for sending queries and for sending other commands (which, together with the different statement classes makes a total of four different ways to send commands to a data base in blatant violation of my razor: *methodi non sunt multiplicandi sine necessitate*), and the peculiar way in which results are accessed.

If one is doing an object oriented design, it is important that the structure of the library follow the structure of the problem and that the important entities in the problem domain be modeled adequately. One great advantage of the object oriented approach is that the problem domain is the same for the designer of the library and for its users: both can recognize which are the important elements and what behaviors are expected from them. In other words, the formalization of the problem constitutes a common ground on which the designer and the user of a library can communicate. If the design of a library is not based on the structure of the problem, the designer is engaged in a monologue that the user will have to decipher without any initial ground on which to rest. Many of the "bizarre" solutions that we all have encountered in the in the use of Java libraries derive from this "technical autism" of the designer, and on our need to understand on what logical basis were certain decisions taken. A situation all too sad because, if only the designer had been a little more careful he would have found a perfect ground on which he could communicate with us: that of the problem.

VIII. THE INTERNATIONAL IMPERATIVE

The majority of the programs written today are used (also) in countries other than that in which they were developed. So, they have to be translated, a process that involves two activities, called *localization* and *internationalization*. Localization is the process of translating in the *target language* (the language in which the program will be used) all the written material that the user will see (menu entries, captions, labels, error and information messages, prompts, etc.), and to replace its icons with their equivalent in the target culture[1]. Localization should be done for every language, including the language in which the program is developed: engineers and computing professionals are often poor writers and the interface captions they produce are often a hodgepodge of the incomprehensible and the nerdy, in varying proportions. For any program that will be used by non-programmers one should always consider having a non-programmer write the interface. I will consider this process of *endolocalization* later on in the chapter but, by and large, localization will be done by a translator or by a linguist, not by a software designer, so we are not too interested in it here.

We are interested in *internationalization*: the collection of programming techniques and design solutions that make localization easier. Localization is an area of translation, and is studied in philology departments; internationalization is a computing praxis and is (or should be) studied in computing science departments. While internationalization is receiving quite a bit of attention these days, there is an aspect of it that is seldom considered and that we will touch upon in this chapter: that of program development. Nowadays, program development is often an international effort, carried out by multi-lingual teams based in several countries at the same time; if a program is primarily an instrument of communication between programmers, then it is an instrument that must function in a multi-cultural, multi-lingual environment. The code itself can be considered a *lingua franca* among programmers, but the language in which the comments and the technical documentation are written may be more insidious.

This chapter will present some principles of internationalization, and recommend proper practices. Some of these principles are quite obvious and have been presented many times before, some not so much. All in all, the guidelines set forth in this chapter will be based on three principles that are sometimes underplayed but that are the true foundation of internationalization and localization.

i) There is more to internationalization than using Unicode or implementing the *Locale* class. Internationalization is a design culture, and it influences the style of a program in all its phases. The whole design, from the beginning, should be done with internationalization in mind; adding it at the end of the process, almost as an afterthought, will inevitably result in an ersatz.

ii) Localization should not be done for international versions only; localization is not really about adapting to different languages as much as adapting to different cultures and, in this respect, it is necessary to localize the domestic versions of the program as well: the technical, "geeky" culture in which a program is produced is almost never the same as the culture in which the program will be used, and it will be necessary adapt the *domestic interface* in a way that is, *mutatis mutandi*, akin to localization. Internationalization is really a misnomer, since it places the emphasis on the creation of international versions; *culturalization* would probably be a better name for it.

iii) Localization and internationalization should not be limited to the executable program. A program is an instrument of communication between programmers and, these days, the environment in which the program is developed if often multicultural and multilingual. Everything about a program, from the comments of the code to the tone of the technical documentation, should acknowledge this international reality.

In the end, localization and internationalization, like programming itself, are about communication: a program is an artifact that must talk to, and interact with, people with different languages, backgrounds, and cultural references. Adapting outer communication of the program to different cultures is the work of translators, psychologists, and graphic designers. The objective of the computing professional is to make their work easier. Java has libraries and standards that deal with various aspects of internationalization. These are, alas, the easy aspects, the ones that can be solved with some simple programming tricks. The hard ones are part of the discipline of design: one should design from the very beginning with internationalization in mind. Of course, the easy aspects are important too, and the programmer should take care of them, as the first few guidelines show.

* * *

Make your program work with multi-alphabet codes.

You may be justified if your program doesn't work with Phoenician or Cuneiform, but there are no excuses today for writing a program unable to deal with Chinese, Arabic, Cyrillic, or the accents of many European languages. Technically, this entails writing programs that work with two-byte character codes, such as Unicode. Nowadays, this is not hard: there are many input-output libraries that work with Unicode and allow you to use strings with two-byte letters; just pick a good one (one that satisfies the criteria outlined in the previous chapters) and stick with it. New versions of the Java runtime environment come with an internationalization library. It is a bit messy, and I don't particularly like it, but it has two indisputable advantages: it supports Unicode (a widespread standard) and is available just for having the compiler installed. If you don't have anything better around, use it[2].

There is more to using a two-byte code that making captions work with Unicode. For instance, will your program work correctly with Chinese file names? Once upon a time, we used to check whether a character was a lower-case or a upper-case letter using tests such as

```
boolean lower(char a) {
    return a >= 'a' && a <= 'z';
}

boolean upper(char a) {
    return a >= 'A' && a <= 'Z';
}
```

This works well with ASCII and English, but what about other alphabets? The distinction between lower-case and upper-case is pretty much limited to the Latin and Greek alphabets: Chinese has no upper-case, Japanese has three separate alphabets, Arabic has four forms of every letter, and so on. Working with multi-language codes means taking care of these aspects as well. For string comparisons, for instance, one might need to define a *normal form* of a word (similar to the conversion to upper-case for the Latin alphabet), a form that will depend on the specific language (in Arabic, for instance, this might entail that all letters are converted to

their isolated form), and to which all the strings will be converted before comparisons. This standard form is alphabet-dependent, and the program should provide the instruments to specify it during localization.

* * *

Allocate plenty of space for strings.

This guideline has two sides: you should think at the same time about the space strings will take on the screen and in memory. Different languages take up different amounts of space to say the same thing; among western languages, English is fairly compact, and the *pigdin*-ish language often used by technicians is even shorter. Translating a sentence in Spanish or French may increase its occupation of as much as 50%: “a laptop” in French is “un ordinateur portable”, a “catchy tune” in Italian is “un motivetto orecchiabile”. On the other hand, Chinese and, to a somewhat lesser degree, Japanese, are typically more compact than English. This is a good thing if the programmer is an English speaker (making things shorter is generally easier than making them longer), but it requires him to be doubly careful if he is Chinese or Japanese. The issue isn’t just that some languages are inherently more compact than others: languages are part of cultures, and they often produce short expressions for culturally relevant things; other languages, coming from cultures in which the same things are not so relevant, use a periphrasis. Americans move around much more than Spaniards, and they often sell their things when they move, so a “moving sale” becomes for the more stable Spaniards a “venta de muebles por cambio de domicilio”. For an Italian who just finished eating “fare scarpetta” means “to use bread to scoop up the sauce that is left in the plate”. You should not assume that, just because something can be said in your language using a word or a short expression, the same will be true for other languages.

You should consider all this when you decide the size of windows for displaying text (it is better to leave this decision to the translator anyway), or when allocating space for strings; you should avoid as much as possible the static allocation of buffers, and you should make sure that the string length function that you use for dynamic allocation works with multi-byte character codes: with Unicode the size (in bytes) of a string is not equal to its length (in characters), but twice as much!

* * *

Collect all location specific elements in an external file.

Translators should do their work without touching the source code of the program: all the strings that need to be translated, and all the elements that need to be adapted should be placed in a file (the *localization file*) that, after localization, will be included in the program to create the localized version. Technically, there are two solutions that can be used for this inclusion: the localization file can be included in the source code by a pre-compiler, or it can be read when the program is executed. Using a pre-compiler has the advantage of not requiring the addition of a separate localization file to the program distribution, allows more flexibility and more powerful localization functions, and has smaller impact on the structure of the program; it has the disadvantage of requiring a separate compilation for every localized distribution. The other solution has symmetric advantages and disadvantages. The decision on which solution to adopt may depend on many issues and corporate policies, and is not always under the control of the designer. Text meant as an internal code to the program (e.g. names of things to be looked up in a table) must not be translated and should not be placed in the localization file.

The portions of the strings that the program will fill up during execution must be specified using placeholders. Translators are generally familiar with C-like format strings (the ones used in the *printf* function), so the localization file should use them or a variation of them. Note that the *order* in which the placeholders appear may change from language to language, so it is necessary to give different placeholders different names and use plenty of comments. In the format I use, each placeholder is represented by a code that indicates its type (%d for integers, %s for strings, etc.) followed by a number that indicates its identity. For example, given the fragment

```
#
# Placeholders:
#
#  %d1   number of occupied records
#  %d2   available space, in number of records
#
MSG315   There are %d1 occupied records, out of %d2
```

the translator can translate

```
MSG315  De los %d2 registros disponibles, %d1 están ocupados
```

The order in which the parameters appear has changed in the Spanish version, and the program must allow of such changes. The pre-compiler solution makes this easier, because the pre-compiler can alter the order in which the arguments appear in the print statement based on the position of the placeholders, so that the statement

```
System.out.println("There are " + n_occupied +
                         "occupied records,"
                      + " out of " + n_total);
```

would become in the Spanish version

```
System.out.println("De los " + n_total +
                         "registros disponibles "
                   + n_occupied + " están ocupados");
```

(Note that the order of appearance of the varaibles n_total and n_occupied has changed.)

Localized strings can be specified using a suitable pre-compiler instruction, such as

```
System.out.println(#TRANSL(MSG315, n_occupied, n_total));
```

or

```
String out = #TRANSL(MSG315, n_occupied, n_total);
```

The parameters are given in the logical order of the placeholders: the first parameter corresponding to %d1, and so on. The pre-compiler will take care of shuffling them around depending on the order of the placeholders in the message string. There are plenty of pre-compilers that do this but I find them all much more complicated than they ought to be, so I use a home-made one consisting of a few dozen lines of C code.

*   *   *

Do not compose sentences from fragments.

As useful as code reuse may be with programming languages, it doesn't quite work for human languages; each language composes sentences in a unique way, and a combination of pieces that works for a language will not, in general, work for another. Consider a program that displays many messages on things that one should do, such as:

eat this
take this

as well as their negations (throughout this section, I will write in boldface the differences between different forms of the same sentence):

don't eat this
don't take this

An careless programmer might decide to save a whole lot of messages by isolating the "don't" in front of the negation. The message file would look like this:

```
MSGNEG     don't
MSG001     eat this
MSG002     take this
MSG003     drink this
```

and the printing code could be something like

```
if (do_not) {
    str = #TRANSL(MSGNEG);
}
str = str + #TRANSL(MSG002);
```

Now we translate the program in French. The positive sentences read

mange ça
prend ça

and the negative

ne mange**s** **pas** ça
ne prend**s** **pas** ça

Negation in French entails more than placing a prefix in front of an imperative sentence: it requires the insertion of the word «pas» (or some other negation word, such as «point» o «rien») after the verb and, often, a change in the suffix of the verb. The simple solution found by the programmer will make translation impossible.

As another example, consider adjectives. In order to write in English

fast train
fast food
fast cars
fast women

one could use the message file

```
MSGFFF     fast
MSG010     train
MSG011     food
MSG012     cars
MSG013     women
```

and the appropriate combination code to obtain the complete sentences. But in Latin languages the adjective is generally written after the name it refers to, and has to agree with it in gender and number, so that, for instance, in Spanish, one would have to write

el tren rápid**o**
la comida rápid**a**
los coches rápid**os**

making the previous messages unusable. The case of "fast women" presents a different problem, since it is part of a cultural jargon, and can't be translated as it is: regardless of what periphrasis the Spanish translator will decide to use for it, the ad-

jective “rápido” won’t probably appear in the translation. German has the additional complication of possessing declensions, so that adjectives are inflected depending not only on the noun that they modify, but also on the function that the syntagm in the phrase and the preposition that precedes them; so we have:

Das ist mein Wagen (that is my car)
Ich suche mein**en** Wagen (I am looking for my car)
Ich fahre mit mein**em** Wagen (I am travelling with my car)

Internationalization requires that the minimal translation unit be the complete message: every message or paragraph used by the program should be written separately in the message file as a unit; no assembly of sentences or paragraphs from smaller units can be allowed. Even under normal circumstances, translating a program is hard work because the context is often insufficient: in addition to complete paragraphs that create their own context, the programmers should put in the localization file plenty of comments to tell the translator exactly what they mean to communicate and the context in which the message will appear. It goes without saying that nothing of this is completely sufficient, and that the relation between the programmers and the translators should be close enough to allow them to go through various iterations of localization before releasing the localized versions of the program. This iterative development should be done at least once during design and, since philologists in the language of the development team are the most readily available, it is another good reason to localize a program in the development language as well.

* * *

Isolated words translate differently depending on context.

Translation is context-dependent. The same word in the source language often translates to different words in the target language, depending on the context in which it appears. This is not a big problem when words appear as part of a sentence (a sentence is usually sufficient to provide the correct translation context), but it may be for the many isolated words that appear in an interface. Do not assume that, just because your language uses the same words in different parts of an interface,

other languages will. Even a simple “No” can be translated differently depending of whether it is displayed as a warning or on a button that the user should select to reject an option. If the same word is used in different contexts it should be replicated in the localization file as many times as are the contexts in which it is used. Messages like these

```
#
# To appear in a window in which the program approves
# the work of the user
#
MSGOK1  OK

#
# To appear on a button in which the user accepts
# an option
#
MSGOK2  OK
```

might result in a Italian translation

```
MSG0K1  D’accordo
MSGOK2  Si
```

In the other direction, in Italian the word “ciao” is used both as a salutation when people meet and as “good-bye”, but the two uses should correspond to different entries in the localization file, since in many languages they will be translated using two different words. For isolated word, it is crucial that each entry be adequately commented, since there is no context to tell the translator how the word should be translated. Without comments, the translator will be forced to do a tentative translation and then hunt the translated word all over the interface to see if the translation makes sense in the context in which the word is used.

*   *   *

Allow the input layout to change, and the changes to be processed.

The language in which the messages are written is not the only element of a program interface to be affected by localization. Often, the input layout must be

changed as well. Consider, for example, a panel in which an address must be entered. In the US, it might be organized like this:

N. Street (Apt.)
City State ZIP code

where the state would probably be selected from a pull-down list. In Spain, the typical address layout would be (I have left the captions in English rather than translating them to Spanish for the sake of clarity):

Street N. Floor Door
ZIP code City Province

where the province may or may not be selected from a pull-down list (the practice seems to be somewhat less common in Spain than in the US).

The difference in format has two consequences. The first is in the interface: the layout and the input panel must be different for the different localized versions, entailing that the layout should not be set in the program, but in the localization file. The second consequence is for processing: the Spanish layout has seven elements, whilst the US layout has only six. The program must be ready to accommodate inputs consisting of different numbers of elements, and all these elements must be accepted, stored, possibly used as search fields, and properly displayed. Note that there is no reason to associate, within the program, fixed or even meaningful names to the elements of the display. Meaningful names are needed only at the interface, and all the program needs is a mapping from interface names, and position in the internal array structure. For example, the address can be received from the interface as an array

```
String address[SIZE];
```

and a proper internal structure, also depending on the language, would map some of these elements to the fields that can be used for searching and ordering. So, the US localized version would use the convention that the interface name “street” is associated to the first element of the array, while the Spanish localized version would use the convention that the interface element “calle” (street) would be associated to the second element of the array. Some of the elements are numbers, and this fact should be specified somehow since it affects the way things are compared (“57” > “137” if they are compared as string, but 57 < 137 if they are compared as numbers). One should, however, always strive to make as few assumptions as possible about the data type of certain fields. Both in the US and in Spain the postcode is an integer, but in the UK it is a combination of numbers and letters. In Florence, street numbers come in two flavors: black (indicated, in the address, simply with the number, e.g. 31) or red (indicated as 31R). And typically number 31 is R in a completely different building than number 31. One should always use data types flexible enough to accommodate variations in the format of the data, and all the necessary variations should be specified in the localization file.

* * *

Avoid unnecessary checks, or allow them to be switched off.

There is always, upon receiving an input, a strong temptation to check for correctness or, at least, for plausibility. It is a good habit, but using it too liberally may create localization problems. Upon entering a phone number, an American programmer might want to ensure that the area code be three digits long and that the number be seven digits long. These are conventions that hold anywhere in the US, but in Italy an area code may contain between two and four digits, and a phone number between five and eight. The “sanity” check will make it impossible for most Italians to enter their phone number. An equally incautious Italian programmer could use the first digit of the area code to decide whether the phone number is that of a mobile phone: in Italy all land lines have area codes beginning with 0, while mobile phones use area codes that begin with other digits. In this case, the phone numbers of Americans would be accepted, but all of them would be classified as mobile phones.

Sometimes the very logic of the program must be adapted for internationalization: the idea of automatically deciding whether a phone number is that of a mobile

phone or a fixed one is clever and could work (although maybe not with the simple trick of the Italian programmer) for most of Europe, allowing the programmer to remove the "check here if mobile phone" box from the interface, but it wouldn't work in the US, where mobile phones and land lines share the same area codes. Therefore, during the configuration of the program, it should be possible to choose between the automatic verification of cell phone number and the presence of an interface element to do it manually.

The designer of a program should avoid, whenever possible, checks based on culturally specific properties. If the checks are considered too important to be removed, there should be a way in the localization file to disable them or to configure them based on the different conventions that apply in the target culture. These tests are particularly dangerous in web applications since in that case the same version of the program will be used in several countries. Some months ago I connected, from my home in Spain, to the web site of a well known car rental company to rent a car for an upcoming trip to Italy. Being Italian, I used the Italian version of the web page, without realizing that this would make it impossible for me to make a reservation: since the page was in Italian, the programmers had assumed that I was Italian (correct) and that I lived in Italy (wrong). Consequently, the web site refused to accept my Madrid address as a valid billing address for my credit card. This is bad design, and there are absolutely no excuses for it. I got a headache and the company lost a customer[3].

* * *

Beware of time and money.

Dates, time, and money are among the most mutable quantities when one changes culture. If one wants to write a date using numbers, there are at least three ways in which there numbers (day, month, year) are ordered throughout the world, and this without even considering calendars others than the Christian-western one: the Europeans write a date as *day/month/year*, the Americans as *month/day/year*, the Japanese as *year/month/day*. The year can be specified using two or four digit. The character that separate the elements can change, the two most common being the hyphen and the slash. If we start introducing letters, the possibilities multiply manifold, from "Monday, the 23rd of April 2007" to "Lun. 23 Apr. 2007" or *Δέυτερα 23 Απριλιος 2007*. Even the four digit year can cause trouble: in Spain and France it is

written with a point separating the thousands from the hundreds: 2.007; in the UK that point would be a comma, is rarely used for any number, and never when the number represents a year.

Americans use a twelve hour clock with AM/PM indication; Europeans are quite familiar with it but often prefer the twenty-four hour clock, which they always use when the hour indication has some form of officiality or in a schedule (the Americans call this the *military* time). The symbols used for writing down the hour can also change: “4:15pm” in the US is “16.15” in Italy, “16h15” in France, and “16:15 h” in Spain.

Different countries have different currencies that come with different symbols, but this is not the only problem. The U.S. and Europe, among others, have currencies with relatively high face value so that the real currency unit is not the Dollar or the Euro, but the cent, and they represent amounts with decimal numbers with two digits after the decimal point. (The decimal point is a comma in Southern Europe.) A country like Turkey, whose currency has low face value will represent numbers using large integers: prices in the billions are not uncommon. As a matter of fact, prices are often rounded to the closest multiple of 1,000 and entered in the form of thousands of Turkish Liras.

The currency symbol goes sometimes before the number, sometimes after, so a price is written *$14.95* in the US and *14,95€* in Portugal. You can find these differences even within a country. In the US there are two symbols for the Dollar: the unofficial “$” and the official “USD”, the first of which goes before the number, the second after the number: it is “$16.50”, but “16.50 USD”. The problem is even more pronounced when expressing amounts per unit: “$16.50/lb”, “16.50 USD/lb”, “16,50 €/Kg”, “16.5 Euros/Kg”, etc. It must be possible for the people doing localization to configure all these options using the localization file.

* * *

And beware of shoe sizes too!

The actual numbers that one enters in a program may depend on the cultural environment in which the program is used, and comparison references, or thresholds to make decisions using these numbers should never be hard-coded in the program.

My trousers size is 30 in the US, 46 in Italy, and 40 in most of the rest of Europe. My shoe size is 10 ½ in the US, 43 in most of the rest of the world. If in a program you write

```
int size = ask_size();
if (size > 15) {
  System.println("Sorry, bigfoot, we ain't got that size");
}
else {
  :
}
```

your customers will soon start wondering why they *ain't* selling any shoes in Europe. One can either define all the culture-dependent constants in the localization file so that the program will be able to work with all kinds of measurements, or define suitable I/O conversion factors so that the program will always work with the same standard measures (say, a shoe size measure in which the average size if 100). Beware of the conversions, though: firstly, a conversion is not necessarily linear, secondly, you have to consider the decimals: in the US you need at least one decimal bit to express a size such as 10 ½; in Europe you can use integer numbers.

Don't be fooled into thinking that the problem can be bypassed just because certain standards are not widely adopted: the English measurement system is adopted in only three countries (the US, Sierra Leone, Myanmar), totaling less than 5% of the world population, but try telling your American customers that, because of standardization, you program only accepts metric input! Absolute percentage of adoption means nothing: what counts is the conventions used among the potential users of your program.

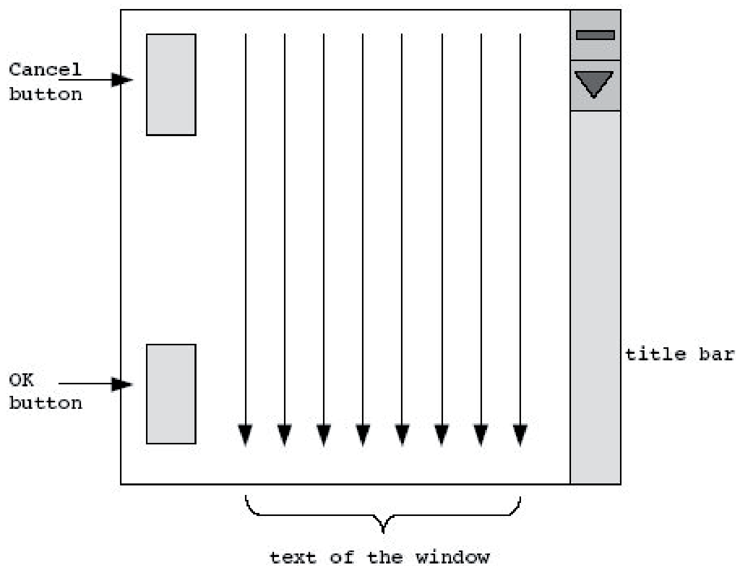
* * *

Allow all dialog windows to be configured

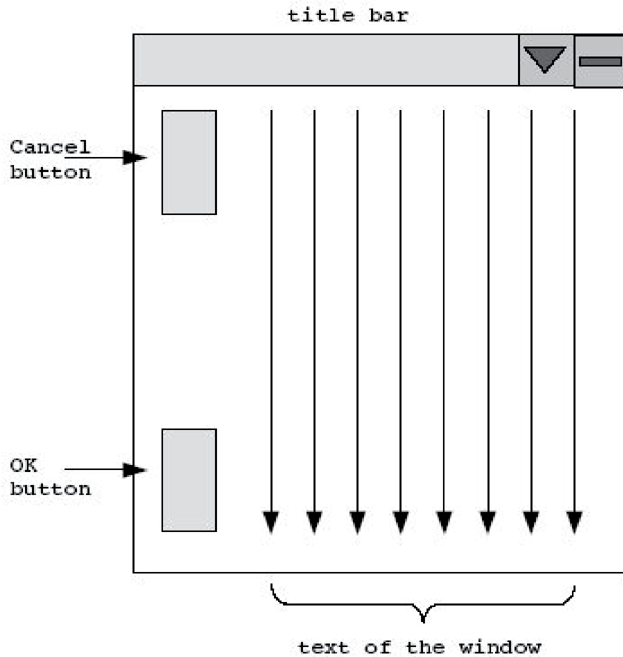
Dialog windows designed by English-speaking programmers have a shape that depends, in part, on the fact that English is written left-to-right, top-to-bottom: the window is typically a rectangle with the horizontal dimension greater than the verti-

cal, with a title on the top (on the "title bar" provided by the operating system), and with the default acceptance button on lower-right corner, that is, in the point where the natural reading flow of a western reader ends. The "OK" button is, in this way, the natural conclusion point of the reading movement, both logically and dynamically. However, different languages are written differently, requiring changes in the window layout during localization. For example Semitic languages, such as Yiddish and Arabic, are written right-to-left. This fits well with the proportions of an English window, but it requires aligning the title on the right end of the title bar and, if the translator decides so, it might require placing the "OK" button in the lower-left corner, where the natural reading flow ends.

Some languages, like Japanese and Taiwan Chinese[4] are written top-to-bottom, a situation that complicates things considerably because the best window for those languages might turn out to be something like this[5]:



Unfortunately, operating systems are western-centric enough to prevent this type of window configuration: the title bar must always be on top, forcing the title to be written horizontally. Still, the localization file should at least allow the translator, should she so deem appropriate, to configure a window in the following way



You might think that this layout is, even for a Taiwanese, harder to understand than the standard one, and you might very well be right. However, the point is that this decision is not the programmer's to make, but the translator's. The purpose of internationalization is to give the translator the instruments to make such decisions, not to make them for her. How these instruments will be used during localization is not a software designer's concern.

* * *

Allow the configuration of icons and colors.

The very name "icons" gives an idea of universality and, consequently, of something that doesn't need translation. In Pierce's semiotic triad, an icon is a sign that signifies by virtue of a resemblance with the *object* (the concept it signifies), and one might believe that, being similarity universal, icons are recognizable by all

cultures without translation. This reasoning is doubly incorrect. On the one hand, as Sonesson pointed out[6], similarity alone is not sufficient for iconicity. Similarity constitutes an *iconic ground* on which iconicity *might* be based. But only certain kinds of similarity give rise to icons, and the selection of the "right" similarity is culturally mediated. On the other hand, and more importantly, many of the icons in a program's interface are not icons at all: their message is symbolic, and depends on the shared presuppositions of a culture.

In most programs, The icon that indicates "open a file" resembles a manila folder of a type very common in the US. In other countries, the prototypical folder might be quite different: the manila color might be quite uncommon, there might be no protruding tab for labels, and so on. The folders that I use in my office in Spain are, all in all, quite different from those that I use in America. They don't much resemble the image on the "open" button of my word processor, since nobody bothered to change the appearance of the folder for the Spanish version of the program. People who have never worked in an office may be unfamiliar with folders to begin with. The "save" icon is the symbol of a floppy disk of a type that is essentially not in use anymore and that many young users of computers will have barely seen. These icons signify what they do by virtue of a convention: we *agree* that the picture of a floppy disk means *save*. This is a symbolic link, as arbitrary as the use of the three symbols /m/, /a/, and /n/ to indicate *man*. Arbitrary, symbolic signification is that of language. It is contingent and culturally mediated. It needs translation. Icons must be translated as much as language because the elements they contain, and the connection between these elements and the action that they represent, are cultural.

When I was giving computer classes to computer illiterate adults in rural areas of Mexico I observed that most people are baffled by "intuitive" computer interfaces. In part this was due to the cultural codes that the interface imposes as "universal" and "natural" but that are based on the presuppositions and the habits of technical office workers and that were not those of my students[7]. Even if a program is sold in a single cultural environment, selecting the right icons is not the programmer's job: the programmer should make it possible for the interface designer or the interface translator to select the icons that they deem most appropriate.

Nor should the programmer or the program designer select colors: colors are cultural as well, and must be localized. I am not talking here of the normal "personalization" process by which a user sets the colors of various screen elements to produce a more appealing look, but of the systematic adoption, during localization, of colors

that convey a message. In the west, black is the color of death, but in China such rôle is covered by white. Purple signifies royalty in Europe and mourning in Brazil; red means danger in the west, but it is the color of nuptials in China. Saffron is sacred to Buddhists. Different cultures "see" different colors because they divide the spectrum in different ways, and create different categories. The color *mabi:ru* of the Hanunóo of the Philippines "includes the range usually covered in English by black, violet, indigo, blue, dark green, gray, and deep shades of other colors and mixtures"[8].

As we mentioned before, these differences may or may not be important, but the decision of whether they are is not the software designer's to make: the designer must simply provide suitable mechanisms so that the people in charge of localization may decide.

* * *

Beware of sorting and case conversion

Sorting often works by comparing the numeric codes of characters and assuming that lower values correspond to characters that come first in the alphabet. This may or may not be the case, as one can easily see just by considering the English language. In the traditional English spelling, the double "o" that sounds like a diphthong is written by placing a *umlaut* on the second "o", so one would write "cool" (the "oo" has a *u* sound and is not a diphthong), but "coöperation" (the "oö" is a diphthong). These two "o" have different ASCII codes (111 vs. 148), but they represent, for the purpose of sorting, the same letter. So, in European languages, do the codes for o, ó (162), ò (149), ô (147), ö (148), ø or those for c, č, ç. On the other hand, in the traditional Spanish spelling, the two ASCII characters "ll" must be considered as a single letter, which comes just after the "l". Things get progressively more complex as we consider alphabets not derived from Latin or Greek. In Arabic the same letter can have up to four different forms (and therefore up to four different codes) depending on whether it is isolated, at the beginning of a word, at the end of a word, or in the middle of it. Chinese ideograms are collected in groups organized at two levels: for every ideogram a *radical* is considered and ideograms with the same radical are grouped together.

For example, the characters: 旦 (*dán*: dawn) and 明 (*míng*: bright) share the radical 日 (*rí*: sun, day). There are about 180 such radicals, and they are ordered, at the

second level of the hierarchy, by the number of strokes necessary to draw them: 日 is a four-stroke radical, 人 (*rén*: man) is a two-stroke one. Within each group the characters are often ordered based on the ordering of their *pyin* transliteration in Latin letters.

Unicode, by and large, respects these groupings, other codes might not. In general, however, sorting depends on the language that is being used: the result of sorting is a simple list in English, it is a two-level hierarchy in Chinese. Sorting should, of course, be localized. If the operating system provides sorting routines (as it should), these should be used, because the national versions of the operating system will (hopefully) replace them with the localized ones.

If this is impossible, the string comparison function on which sorting is based should be made easy to replace, either by implementing it as a set of rules that can be placed in the localization file (a solution that could create considerable efficiency problems) or, in Java, by placing it in a separate jar file with a well documented interface. All string comparisons in the program should be done using this routine.

* * *

Localize in your own language too.

Localization and internationalization should not be limited to programs destined to be distributed internationally: the same principles that we have seen so far should be applied to the “domestic” version of the program. This too should be localized by a translator, a writer, an anthropologist, or a psychologist (or, better yet, all of them). Computing professionals are often poor communicators, immersed in a technical jargon largely unintelligible for the outsider, and there is no excuse for having a computing professional or an engineer write the output messages of a program or create its input panels. *Coeteris paribus*, a program whose messages are written by a programmer will be less usable than one whose outputs are written by somebody who knows the intended audience, its culture and its language forms. An ideal situation is that in which the writer knows nothing about computers: this way, he will go through the same problems that users go through trying to understand the logic of the program and thus he will be able to explain the program better.

Programmers, alas, tend to write everything as if they were writing for other programmers. Recently I tried to log in to the reviewer’s page of a conference only to be faced with a white page containing the message

Parameter r should be int or string and not nonexistent

To this day I have no idea what the parameter “r” is and why it had suddenly dropped out of existence. And I am one of the lucky ones: at least I know what an “int” is, and that a “string” is not just a part of a guitar. Imagine the reaction of somebody who doesn’t know any technical jargon when seeing this message. Messages should always be directed to the people who are using the program, and should make sense in the context of what they are trying to do. In the conference example, I was trying to review a paper, and the message should have talked to me in the language of conferences and paper. It is reviews I cared about, not parameters. As it is, somebody using the system to enter a review for a music theory conference would be flabbergasted to know that something bad happened to the strings of somebody’s beloved violin.

Writing cryptic messages that only make sense in the world of programming is a symptom of a deeper cultural and educational problem, one that affects quite directly the quality of software. Programmers are not be required to write well any more than writers are required to program well, so the domestic version of a program should be localized out of “programmerese” into plain language as much as the foreign versions are.

* * *

Put everything you can in a single localization file.

All aspects of localization discussed thus far should be controlled through a single *localization file*, well documented and divided into sections so that a translator should only worry about the messages section, a graphic designer about the colors and icons section, an interaction expert about formats and layouts, and so on.

It is better to place all these things in a single file rather than having them scattered around in a number of them: having it all in a file reduces the risk of inconsistency after mixing files relative to different languages, makes it easier to pass the complete specification back and forth through email, and makes it easier to keep track of several language versions at once. The file should be as open as possible in format and accessibility with different programs. Its very nature prevents the localization file from being an ASCII text file, but it can very well be a Unicode text file, readable by any text editor that reads Unicode. Creating a binary file and

providing specific “tools” that work with it is unacceptable: the decision of which programs will be used as an aid to localization should be the translator’s, not the programmer’s. Much like a programmer with his favorite editor, many translators have programs that they like to work with, and the file format should allow them to use whatever program they like.

You should use a format that allows the easy introduction of comments and avoid being lured into using ersatz formats just because they are standards. These days, chances are that you will be pushed towards using XML as a syntax for the file. Resist the pressure. The localization file is *grosso modo*, a list of things, not a hierarchy, and there is really no reason to represent it using a syntax designed for trees. Many translators are not familiar with XML, and using it will only make things harder for them. Moreover, XML is hard to comment clearly: comments get lost in the general syntax and it is hard to know what they refer to. Often I use a format similar to that of a configuration file: the first symbol of each line is a tag telling what the line is about, and the rest of the line is the content of the tag; comments are introduced by a “#” and run to the end of the line, and commands can span several lines by terminating all lines except the last with a “\”. You may want to do something different, but my recommendation is to keep it simple: you will have to explain this format to people who are not technicians.

* * *

“Think internationally” when documenting a program

There is more natural language in a program than just the messages in its interface. As we saw repeatedly in this book, a program is an instrument of communication between programmers and, in addition to the formal communication constituted by the code itself, it communicates informally through the comments in its code and its technical documentation.

Nowadays, it is very common for this communication to cross linguistic and cultural barriers—it is not just the users of a program that belong to a multitude of cultures: its developers do as well—and, while the formal code can be assumed to be a *lingua franca* among programmers, the language of the comments is not: this form of programmer-to-programmer communication must be as international as the language in the program interface.

It would be nice to be able to write comments in ten different languages but, even for somebody as polyglot as Charles V[9], this might be too much to ask, so let us assume that you write the program's documentation in English. Even so, you should always remember that you write for people who don't share your linguistic context: some of your readers will use English as a second or third language, and they will have an imperfect mastery of it; other will master English perfectly but will not share your cultural background. When addressing an international audience, you should write a neutral and grammatically correct English. If some of the things that you write are not clear to your audience, they will likely consult a dictionary; if you are using an expression from a jargon, not contained in the dictionary, their progress will be stymied. Limit your jargon to expressions absolutely standard in the computing profession (referring to my previous example, in the technical documentation you can, of course, assume that all your readers will know what a *string* or a *tree* are) and, when in doubt, avoid jargon at all.

Depending of your cultural background, you might have been miffed by my use of certain expressions in this chapter, such as *mutatis mutandis* , *coeteris paribus* or *grosso modo*, or by my use of words like *polyglot* in lieu of the more common *multilingual* (and maybe even by the expression *in lieu* that I just used). In my cultural milieu (in my neck of the woods?), these are quite common interjections in one's English prose but, if in yours they are not, they might have sounded out of place to you, even though surely their use did not prevent you from understanding what I was saying. The same is true when you use expressions derived from, say, American popular culture, such as metaphors based on the language of sports or the military. To say that something is a *slam-dunk* or that it is a *home run* will be equally annoying to people not familiar with basketball and baseball. The use should be avoided of culture-specific metaphors such as "10 yards line" (sports), "all hands meeting" (navy), "cut to the chase" (film), using words in unusual acceptations such as "leverage" for *exploit*, "key" for *important*, using local cultural expressions such as "piece of cake", a "catch-22", or synecdoches like a "suit" (meaning a manager).

It is a good idea to have a dictionary at hand when writing documentation. A prescriptive dictionary such as the Oxford is better for writing than a descriptive one such as the Webster. If a certain connotation doesn't appear in the first two or three acceptations of a word, it is better to avoid it. Most of this is, of course, common sense, and one can easily apply some simple rule of thumbs. Americans, for example, have a very lively, imaginative, and unapologetically metaphorical way

of writing and speaking: unless the documentation and the comments you write will appear unbearably dull to you, you are probably writing in a way that foreigners will find difficult to understand.

IX. A COUNTER-CULTURE MANIFESTO

The previous chapters have discussed, with varying levels of detail, a number of issues of style that arise in the design and the implementation of a software project, with particular emphasis on their implementation in Java. This exploration has evidenced, on the one hand, the relevance of a correct style in the general economy of program development and maintenance and, on the other hand, a number of shortcomings in the way things are routinely done in the industry, especially when it comes to implementing the structural isomorphism between problem and solution and to using in a correct way the instruments made available by object orientation. It should be clear from the previous chapters that most of these shortcomings are not to be imputed to the programming language itself, but to the cultural milieu that has arisen around the development of the applications for which it was intended. Ironically, while we are observing the inflation of a second "dot-com" bubble (albeit, at the time of this writing, in the middle of an economic crisis), many of the bad habits of the internet programming culture are still a remnant of the first one. To the extent that the abysmal quality of their software was a cause of the (deserved) demise of most of the companies that made up the first "dot-com bubble", it is reasonable to expect that the second bubble will meet the same fate: internet software designers do not seem to have learnt the lesson[1]. Hegel wrote that the only thing History teaches us is that people never learn from it[2].

A lot of today's internet programming culture resembles, in remarkable and preoccupying detail, the technical culture of the "dot-com" era. Everybody who has worked for an internet company between, say, 1997 and 2000 will remember an environment in which churning out new versions of whatever it was that the company was producing was more important than assuring its quality[3], in which asking for time to do a proper analysis of the problem and a proper design was frowned upon and seen as a symptom of an "ancient" attitude, the mark of somebody who was not in sync with the Zeitgeist, of a dinosaur (a mark that usually preceded summary

dismissal by just a few days). An environment dominated by silly slogans such as "we have to move fast" (often, alas, without discernible direction), or "we are lean and mean" (which meant that a programmer was asked to do the work of five, and management was mean enough to do the asking without flinching). Most internet companies have ended up in the dustbin of history, where they belong, but the culture that they generated lingers on.

Java as a programming language, many of the standards connected to it, and many of the habits of its programmers, thrived or were created in this kind of industrial environment and are still, in many ways, dependent on it. The risk, of course, is that whole programming environment, a whole way of writing programs, will follow the "dot-com" into the same dustbin of history. This is the challenge to which the computing profession is called to respond. The demise of the small internet companies, while deserved, had a negative consequence for the internet as a social medium and as a common good: it has left a large portion of it into the hands of fewer and larger industrial players. The few successful companies of the age, such as g*oogle*[(TM)], have entered the rank of the giants they used to scoff and to warn us against and, while they still managed to retain some of their ancient patina of "coolness", the paint is wearing thin. Google used to dress itself as the anti-Microsoft—Microsoft being seen as the quintessential representative of the establishment, just the way Microsoft regarded IBM two decades earlier—but, like the pigs and the humans at the end of Orwell's *Animal farm*, it is becoming quite difficult to distinguish the ones from the others. The negative consequence of the old "dot-com" period is to have shown, albeit through a failed example, that it is possible to incorporate the capacity of the internet into the post-industrial mechanism of consumption and to absorb its original anarchic creativity into the neo-dirigist economic environment of the transnational corporation.

Before 1997, it is worth remembering, the internet was mainly the playground of universities, grassroots organizations, and political action groups[4]. Although *Mosaic*, the first modern browser, was released in 1994, in 1995 Microsoft's first "real" operating system, Windows '95, had no built-in provision for networking, let alone a web browser.

The "dot-com" opened up the industrial exploitation of the internet, putting it in the hands of a myriad of small players. With their demise, the industrial exploitation of the internet hasn't disappeared, but has concentrated in a smaller number of larger hands. From the ashes of the internet start-ups, internet commerce was

born and, with it, a use of the internet almost opposite to that of the original universities and grassroots organizations: instead of a medium for socialization and involvement, it became a medium for a more secluded and private performance of the essential post-modern function: that of buying. In this sense, the internet risked becoming a very powerful instrument of social isolation, a situation only in part alleviated by the emergence of the ambiguous phenomenon of the social networks. The programming culture, at least in its mainstream components, is placed in the middle of this scenario, and has become an instrument for the commercialization of the internet. A lot of Java manuals read more like business documents than technical books; terms like "computation" or even "program" are disappearing, while terms like "enterprise information system," and "business logic" (whatever they mean) are all over the place. Even the old, but substantially correct, "data processing" has disappeared, replaced by the quixotical "information technology" (the acronym IT is *de rigeur*). Again, this shift is not exclusive to the Java culture. Today, in a data base manual, one can read sentences like

> Historically, the system analyst studied the *business* requirement and built an application to meet these needs. The user was involved only in describing the business [...]
>
> The end user will know details about the *business* that the developer will not comprehend[5].

These notes are directed to programmers, from whose point of view it doesn't make any difference if a program is written to keep accounting records for a company, to study the interactions of neutrinos with matter, or simply for the fun of it, but they introduce the general idea that programming is an activity affiliated with business, and pretty much nothing else.

The problems of the Java culture derive in great part from the assumption of the ethical model of post-industrialism and of its consequences, in particular of the industrial predilection for quantity over quality. Java programmers, more than any other programming sub-culture, have absorbed the commercial myth of self-justifying quantity, of measuring progress by the number of "features" that a product

implements[6], of the primacy of production over need. As Kranzberg[7] pointed out, it is rarely true that necessity is the mother of invention; the most common direction is the opposite: the invention comes first, and creates a need for it.

All this has led to a shallow culture, in which the formal analysis of a problem is almost forgotten, in which design is often superficial, and in which interfaces are often defined without the due attention to their user. In order to understand this culture and its flaws, it is necessary to go back for a moment to the industrial environment in which it originated. During the XX century, the nature of industrial management has changed quite drastically. By and large, at the beginning of the century, industries were managed by people who were, so to speak, products of the trade: the steel industry was managed by people who knew about steel, railroads were managed by people who knew about railroads, and so on. This was a natural evolution of the way things are done (still today) in craftsmanship. The professional figure of the manager *qua* manager, of the person trained to *manage*, independently of the trade being managed was, in the XIX century, unknown. During the XX century management became progressively independent of the trade that was being managed, and the professional figure of the manager began to emerge, together with an abstract management discipline. A symptom of this evolution, and of the new respectability that management had acquired, is the appearance, in the course of the XX century, of schools of management on university campuses. In the course of the XIX century, and well into the XX, the idea that, say, the director of a farm could go on directing a railroad would have provoked wild laughter, while in the 1990's it appeared quite natural that the CEO of Nabisco would go on to direct IBM. This shift, in a sense, goes together with the affirmation of the market as a primary entity, independent of the goods that are being traded. Many of the consequences of this situation are not of interest for this book, but one is: professional management brought with itself the idea that an industry is an abstract entity, independent of what it produces; this, in turn, has lead to the idea that an industry should be made as independent as possible of the special capacities of its employees. Professional management needs repeatability and predictability, two qualities that don't fit well with the *modus operandi* of an artisan or of a scientist[8]. Or of a creative programmer, for that matter.

The idea of regulating the production of programs, of making as many things as standard as possible, percolated from more traditional industries to the "dot-com" companies when these, after an initial "garage" phase in which they were composed of small groups of engineers working non-stop in a room to build something, were

pressured by their investors to hire professional managers. From the "dot-com" companies, in turn, this idea of repeatability and predictability percolated into the internet programming culture where it appears now quite pervasive. Standards and the almost obsessive adherence to them are a conspicuous result of this process. So that there are no mistakes, it is not my purpose here to deny a place for standards in the internet world: whenever two computers have to exchange data, it is necessary that they use the same code and, in the widespread and loosely connected world of the internet, this is done by proposing standards and inviting people to use them (the internet is clearly too lose a network to have standards *enforced*).

However, one can quite safely say that many standards about program development end up stifling creativity, without serving any positive purpose worth the loss. The idea that a simple set of rules can help people write better programs is a pious illusion: program writing is much more complex and intellectually challenging than that. Such efforts might work for some industries or other organizations. They work well, for instance, for bureaucracies, which are some of the longest lasting organizations in our society, and are thus precisely because in this case the organization is indeed independent of the capacities, the personality, and even the presence of any single member. Bureaucracies, however, work well in this way because their function is to take care of the routine, and the routine can be described and codified in a set of rules. Everybody has experienced the frustration of trying to get a bureaucracy do something out of the codified rules (such as convincing a phone company to give us a refund for an unusual problem[9]): bureaucracies are not designed for this kind of operations; they work well because these operations are relatively rare in their area of activity.

Programming, however, is different: in a programming environment unusual, unique problems are the norm, and routine development is the exception. Many failures of the software industry derive from the self-imposed delusion that it was the other way around. Bureaucracies work by codifying the routine, but programming can't work that way, because in programming the routine is only a marginal activity; the software industry must stimulate and embrace creativity. This consideration is self-defeating and fatalistic only if one abides to a romantic view of creativity as a form of innate genius, an internal force that explodes within the individual in spite of society and against it, not thanks to it. But this is not the case: creativity can, and should, be learnt, and the problems that we have teaching people to be creative derive not from the impossibility of creativity to be taught, but from the wrong way we go about teaching it. You can't teach somebody a direct recipe

or a list of things to know and to do in order to be creative. The only way to teach somebody to be creative is to place them in a stimulating intellectual environment, one in which rejecting and debating accepted truths is so common that they will stop even noticing that they are doing it, and it will be as natural as breathing. The first place in which such an environment should be found is in our educational institutions but, unfortunately, our educational institutions are suffering a transformation that is taking them from teaching creativity, one whose consequences will be negative not only for them away, but for the computing profession, and the general cultural level of our society as well.

In the last few decades the technical and scientific departments of our universities have become closely associated with the industry, a situation that might have been marginally positive were it not for the fact that academia—placed into a deep crisis by dwindling public funding for education and by the departments' increasingly extravagant needs—approached the industry from a subordinate position, absorbing uncritically its purposes, *modus operandi*, and ethos. It might seem as trivial to point it out, but academia and the industry are very different institutions, with different goals (to make money the first, to educate people the second), different horizons (three to five years the first, the sixty years of a student's expected life the second), and different ethics (economic the former, scientific or humanistic the second). To ask academia to work according to industrial values is as absurd as asking a company to work according to academic values. A company that followed academic values would soon go out of business; a university that follows industrial values will soon be intellectually bankrupt, and will fail to provide to its students the service that they need. The industrial values and ethics are neither better nor worse than those of academia: they are simply different; academia and the industry perform different social functions, and the values on which they are based reflect this difference. It goes without saying that it is normal in a complex society that institutions should coexist that pursue different goals and that perform different functions. Alas, in the last two decades the market has been assumed as a universal ethical model; everything, from education to religion, is considered as a product, to be judged strictly according to their exchange value. The first consequence for academia and computing science was a change in the direction and the priorities of research, towards more "applied" research, which meant, in practice, that academia was asked to work on the short term agenda of the industry.

But the industrial influence has brought forth important changes in the sphere of education as well. The word *school* comes from the Greek *skholé*, which means "a place of leisure". A place where, for some time, we can set aside the practical

problems of our daily life and think about a different class of things, things that normally are precluded to us by the pressure of practical problems. The questions that one considers in the *skholé* are not, not could they be, the questions that interest the industry; nevertheless, they are questions that, as human beings, we find profoundly necessary. The purpose of academia is to serve the students, which are, so to speak, its customers. Universities must give their students this leisure space, this *skholé*, which is necessary for their human and intellectual formation.

With the growing influence of the industry, a radical change is taking place: academia is becoming an instrument to serve the needs of the industry, no longer those of the students. The students are no longer the "customers" of academia: they have become its product. And a product they are, which must be produced following the industrial rules: it must be standard, uniform, replicable. The individuality of the student is, from this point of view, not an asset but a liability, and its effects must be countered, as they go against the uniformity of the product. Hardly the premises on which creativity and mathematical culture can develop. One way to obtain this result is to move academia away from an education oriented to the development of the individual and towards a task-oriented education. In other words, by making university education more and more vocational. Initially, vocational education found its way into small, generally private, technical colleges, which offered training for very specific professions and a curriculum shorter than the traditional four years, resulting in a college degree that, at least in the US, was not a university bachelor degree. Many of these colleges did, and still do, an excellent job in vocational training, but there used to be a general consensus that, next to this, students needed a different type of education, one more general in nature, more oriented towards education than training and that, by and large, big universities with their dense four-year curricula and their postgraduate education were the places where this education could be obtained[10]. Things, as I said, have changed in the last two decades and these days, even the curriculum of large and prestigious universities has steered sharply towards the vocational[11]. Vocational training is done, by and large, by *ostentation*, that is, by showing the students *how* something is done. Sometimes this is done using general rules, other times (more often) by showing specific examples from which a general pattern can be inferred.

This form of education is typical of certain, very specific, kinds of job training and, for many industrial needs, it works quite well. Many professional and industrial figures are purely technical, that is, their job is the application of known techniques to the relatively standard problems that are encountered in the everyday

practice. Mechanics, to name one profession, work this way and, for them, an ostensible style of training is quite adequate[12] at least as job training: their education as people is, of course, a different matter. These are professions for which the training provided by technical colleges works best.

The programming profession has a large component of mathematical and linguistic creativity—programming can be equated to proving theorems, and the abstract description of a complex problem requires a considerable linguistic sophistication—and creativity can't be taught by ostentation. Much like in the education of a creative mathematician, the best a teacher can hope to do is to bring to the surface and stimulate the creativity that the student already has, and to teach the technical notions and the *forma mentis* that will allow the student to put his creativity to good use. My electronics professor at the University of Florence used to say that a professor doesn't have to teach anything, all he should do was to transmit to the students the enthusiasm for learning. One has the impression that the education of programmers is lacking in all these aspects. The only way to stimulate creativity is to educate intellectually curious individuals with a rich cultural background. It is much harder to be creative when the education that one receives is specific, technical, and restricted to one area. Creativity emerges—or, at least, is greatly facilitated—at the juncture of different disciplines. This juncture is not quite that of "interdisciplinary" work, so much in fashion today: the difficulties and the problematic of being creative working, say, in bio-informatics are the same as those in theoretical computing science. The juncture I am talking about is that between different cultural aspects of the same person, the different facets of one's knowledge that allow one specific person to see a problem from a new and original point of view, a point of view exclusive to her. To create something new, a mono-thematic education is almost invariably sterile.

To make an imperfect parallel: a lot of people have experienced that learning a second foreign language is easier than learning the first[13]. one possible explanation is that, with only one's native language, one is so tied to its grammar that departing from it might seem so unnatural as to be impossible—English speakers have a hard time connecting with the concordance of the adjective or with the subjunctive, Chinese have a hard time connecting to the use of the article, Latin have a hard time connecting to phrasal verbs—but once one has acquired a first foreign language, all the rules have somehow been relativized by a new syntactic point of view, making it somehow easier to acquire new ones. A first foreign language re-defines the scheme in which languages are learnt and used, allowing one to "see" languages in a different way, and opening up new possibilities for further learning.

Teaching relevant techniques is of course necessary—every craft has an important technical basis that has to be mastered—but it is not sufficient. A good brush technique, a keen sense of color, knowledge of geometry, anatomy, and art history might be sufficient, today, to create a credible cubist painting. Participating in the birth of Cubism, however, requires a cultural reflection on the process of photography, the new meaning that it gives to the "subjectivist" view of reality identified with geometric (Albertian) perspective. It also requires a certain psychological insight to reject such subjectivism and to realize that it doesn't correspond to the visual experience (otherwise the conclusion would have been a sterile declaration of the death of painting), as well as a general idea of non-Euclidean geometries. It requires a leap, a new way of seeing art and representation. Only somebody immersed in an intellectual milieu in which these topics are discussed could have participated in the movement that originated Cubism. It is impossible to teach somebody painting techniques in a vocational way and have that person create something original in modern art: a much wider cultural background is necessary, including history, perception, literature, mathematics, philosophy, etc. The same is true for all creative activities, including programming: teaching the mathematical basis of programming, teaching λ-calculus, logic, and algorithms is necessary, but these things alone will not make a creative programmer. In order to have a creative programming environment it is necessary first to build a complex educational environment, in which various influences are felt and debated. We need a humanistic, as well as scientific, education of programmers. Alas, the direction in which academia is taking the education of young scientists and professionals is, as I have already discussed, very often the exact opposite, a situation that is a direct consequence of the increasing influence of industry on the technical departments of academia, and of the pressure that it is exerting to bring education to a completely vocational model. But, and I am somewhat repeating myself here, universities are cultural institutions devoted to the free pursuit of knowledge: their only responsibility is towards their students and their function is to provide students with the basis for a rich and rewarding intellectual life. It is the interests of the students that academia should look after, not the companies'.

Edsger Dijkstra liked to use in this context a quantity called the *Buxton index*[14]. The Buxton index (I will simply call it the index from now on) is the distance into the future one looks when taking important decisions. For a company it used to be about ten years but it has diminished considerably in the past decades and today, for a high-tech company, is rarely more than two or three years: in three years the economic and technological landscape will have changed so much that one can't really

plan beyond that point. For a young person trying to get married it is probably 40 or 50 years, for a person buying a car is probably between five and ten years (depending on how sensible one is to the lure of consumerism), and for a devout Christian it should be infinity. The index is important because two entities with widely different indices will find it very hard to work together; their priorities and goals will inevitably differ. The entity with the longer index will be prone to accuse the other one of shortsightedness and superficiality, to which the other one will retort with accusation of idle abstraction. Both these accusations have no basis: they are simply the results of different scales of priorities due to different environmental pressures, and it doesn't make any sense to ask who is right and who is wrong. One might, at most, ask whether the index of this or that entity is adequate for its general goals. In the case of an educational institution, the proper index is given by the expected active intellectual life of its students. In the case of 20 year old undergraduate students, this is at least 50 years, probably more. This simple realization problematizes the strict relation between the high-tech industry and academia. A difference of an order of magnitude in the Buxton index is not the best premise for a comfortable and fruitful relationship. The imposition of a vocational model of education is the industry's attempt to correct the discrepancy by shortening the academic index: vocational education goes naturally with a short index because of its teaching techniques and the topics it includes. This, alas, comes to the expense of the long term interests of the people whom academia should defend: the students.

It is not simply the students who suffer. In the long term, the generalized adoption of a vocational education model will also be detrimental for the computing profession and to the software industry, at least to the extent that the industry needs original and creative people. There are structural reasons why the industry—as it is organized now—rejects originality and creativity beyond certain very strict and manageable limits. I have touched briefly on some of these structural reasons, and I will not consider them further, but I will observe that, due to the peculiarities of its product (a product with heavy development investment and zero marginal cost), it is doubtful that the software industry will prosper in the future unless it embraces long term creativity. That is, unless it releases academia from its influence and lets it give students the education they need.

It is not simply the students who suffer but, nevertheless, it is mainly them. Vocational education, with its emphasis on specific, limited, and short-lived techniques rather than on general principles and on the creation of a mindset represents a disservice to the students, who will leave the university with a baggage of bits and pieces of knowledge that will be obsolete in a few years and that, lacking them the

mathematical and logical bases to grasp the general principles behind the specific techniques, will be hard to replace. In a few years these, by now, ex-students will have to struggle and work long hours to keep up with the people just out of college who, of course, have had all the time in the world to study the new idiosyncrasies. Continuing education is a noble goal and should be supported, but it has to rely on solid cultural foundations, and it should be a way to improve the quality of the intellectual life of people, not a way to make already overworked employees more miserable.

* * *

So, if I had to create a "manifesto" of a programming counter-culture, the points that I would stress in education are a broad scientific and humanistic culture and a great attention to the mathematical basis of computing science. My ideal curriculum would have a duration of five years and, during the first two years, it would teach little or no computing; it would concentrate on mathematics (especially discrete mathematics), logic, the history and sociology of technology, epistemology, and philosophy, especially the philosophy of language. To say why mathematics, logic, and epistemology are necessary would be stating the obvious. Philosophy of language is necessary because, as we have seen in this book, programming is a linguistic game. A very specialized one, one based on a formal language, and one in which the normal reasoning based on analogy and metaphor doesn't apply, being replaced by abstraction; a linguistic game that does not live in isolation, because the initial specification of any problem is given in the native language of the development team. For a computing professional, Wittgenstein is more important than Berners-lee, the creator of *html*. The sociology of technology is necessary because computing professionals, programmers in particular, are relevant social figures, due to the increasing relevance of the computing infrastructure in our society, and they must have a sensibility appropriate to their relevance. It is necessary, today more than ever, that the figure of the "geek," the technically very knowledgeable but otherwise ignorant and socially inept engineer disappear from our campuses. The figure of the "hacker," the incredibly proficient programmer who is happy only in front of his console, who has no idea of what is happening outside of his laboratory, who has no political opinion and no social awareness is part of the history of our profession, but it is time that we outgrow it. A programmer, in this era of high relevance and social visibility for the computing profession, must be a responsible figure, aware of the implications and consequences of his work, a sophisticated and conscious social actor, not a person who knows Linux line by line and nothing else.

Many of the decisions that programmers take on seemingly technical bases have important consequences that go beyond the technical factors that went into them. I will make just one example. Only a few years ago, memory and disk space were considered precious resources, and a lot of efforts of programmers and system designers went into making the best possible use of them. These days the general mood seems to be that memory and disk space are so cheap that it is better to spend effort on something else (for example in creating new "features" for the system that one is designing) rather than in devising algorithms for the optimal use of computing resources. This, as a technical position, doesn't make much sense, of course: a good designer should always make the best possible use of the resources available. Its adoption, however, has much more important consequences at the level of marketing, since it is inserted into the general consumeristic trend towards always larger and less efficient programs, that require always more powerful machines. These days, if one wants to run the latest versions of certain programs, it is necessary to buy a new, top-of-the-line computer every two years. Hardware manufacturers have an obvious interest in propagating this state of affairs, and the software industry is not against it either, since it means that they can pack more and more "features" (many of which quite useless) in their programs to lure buyers, or force them into new version by incompatibility with what everybody else is using.

Opinions on whether this state of affairs should be seen as positive or negative will, of course, vary widely, and I suppose that every person will have a different point of view on the matter. A programmer, however, should at least be aware that the decisions on what algorithms are being used, far from being purely a technical matter, are inserted into this complex network of economic and political relations of power, so that he will have at least a sense of the social impact of the technical decisions that are being made. This is but one, rather simple-minded, example of the breadth of the consequences that software design decisions can have. As the influence of the decisions of a programmer widen, the programming profession becomes more complicated, and a better cultural preparation is required to cope with it. Ultimately, this is what counts: a programmer, like everybody else, is a player in a power game whose purpose is not clearly established, nor can it be, beyond the generalist notion that we should operate in order to give everybody a better quality of life. But, what is exactly a good quality of life? Can it be reduced to the possession of goods (or "toys," as the high-tech crowd calls them, with a rather transparent Freudian choice of words), or do other values such as intellectual pleasure play a rôle in it? Being an important player in this game means, among other things, to have the culture and the sensibility to ask these questions. And this, also, is something that one can't find in a set of coding conventions.

* * *

To conclude, the only recipe that I can find for the software quality crisis is to educate programmers to be sophisticated intellectuals and creative mathematicians, then trust them, without trying to micro-manage their work.

I expect quite a bit of resistance on this point, and I am quite sure that I will not be disappointed. I am sure that I will be told that these are the ivory tower dreams of an academic intellectual (using a connotation of the word charged with all the anti-intellectualist disdain that one can muster); that out there, in the “real world” things are different: deadlines are looming and nobody has the time to be a Cubist or to think about Wittgenstein, the *skholé*, or the formally correct way of doing things.

My only answer to this is to repeat what I already said in the first chapter: in the “real world” of industrial software development the majority of projects are delivered late, over budget, not fully certified, and using more resources than they have to. In this situation the software industry doesn’t seem to have much to lose trying something new.

Madrid, Florence, and San Diego, May—December 2009

NOTES

NOTES TO CHAPTER I

1. These are indicative numbers. The estimations vary greatly depending on the data that one analyzes and, most importantly, on the definition of "failure" and "over-budget" that one considers. The Galorath® corporation, on its web site, reports the results of various studies. The *Standish Chaos®* report, for instance, reports for the year 2004, 18% failed projects, 53% "challenged" (not completely successful) projects, and 29% successful project, and they estimate that failed projects cost $55 billion annually. The situation seems to be getting worse: in 2009 the same index reports 24% failed projects, 32% succeeded projects, and 44% challenged projects. Almost one in four projects never got out the door. The data from Oxford University are somewhat more comforting, at least as concerns completely abandoned projects, which form a 10% of the total, with a 74% challenged project and a 16% success rate.

2. It is a well known industry principle that adding more people to a late project will delay it even more. In programmers' terms this is obvious, because the productivity of n programmers is at most $O(n)$, while the communication cost is $O(n^2)$. This is, of course, a very rough model. It works well to understand why communication is so expensive, but it is too blunt an instrument to attempt any kind of measurement. However, according to the same web page cited in the previous note, the American National Institute of Standards and Technology estimates that 80% of development cost involves identifying and correcting defects. That is to say, it is cost related to the use of a program as an instrument of communication between programmers.

3. The *International obfuscated C code contest* is a yearly competition whose purpose is "to show the importance of programming style, in an ironic way". Entries to the contest are syntactically correct programs (they must compile without warning with an ANSI C compiler) that are as impossible to read as the author can muster. The most unreadable correct program is declared the winner of that year's competition.

NOTES TO CHAPTER II

1 Simone Santini. (1996). "The graphical specification of similarity queries," *Journal of Visual Languages and Computing*, 7(4):403--21.

2 One might object that the emphasis on the trickery and idiosyncrasies of a programming language, implied in the statement "educated using Java" is not a good way to educate computing professionals, and I would agree. Computing science department, alas, seem to have a different opinion on the matter.

3 Replacing a well-tricked line of code with several lines or (as is more frequently the case) a function comes with a performance penalty, of course, but I will not consider performance in this book on grounds that, if efficiency is a primary concern, one probably shouldn't be using Java to begin with.

4 N. Wirth. (1985). *Programming in Modula-2* (3rd corrected edition). Springer texts and monographs in computer science. Springer-Verlag:New York, Heidelberg, Berlin.

5 The word "semantic" is losing most of its meaning these days due to over-exploitation and under-definition. When I talk about semantics in this book, I will refer to the very limited and very formal definition used in programming language theory. For the sake of concreteness, one can imagine that, whenever I write *semantics* I really mean *denotational semantics*. See, for example: R.D. Tennet. (1976). "The denotational semantics of programming languages". *Communications of the ACM*, 19(8):437-53.

6 Java doesn't allow free functions: functions can only be defined as methods of an object or as static methods of a class, but I'd rather not muddle my example by making these functions part of an object. I am confident that the reader will have no difficulties making the necessary changes.

7 See, for example, Bjarne Stroustrup. (2000). *The C++ programming language*, special edition. Addison-Wesley: Reading, MA.

8 See note 3 of chapter I.

9 Many C and C++ guidelines restrict the use of macros to the definition of constants or as "guards" to avoid including the same header file twice (see, for

example, Lockeed Martin Corporation. (2005). *Joint strike fighter air vehicle C++ coding standards,* document N. 2RDU00001 Rev C). Since Java doesn't have header files, and constants can be defined using the *static final* class, it needs no macros.

10 Most of the code written in this book will be written in Java. Sometimes, like in this case, I shall use a simple pseudo-code. In order to distinguish it from the Java code, the pseudo-code commands will always be underlined.

11 Doets Kee and Eijck van Jan. (2004). *The Haskell Road to Logic, Maths and Programming.* College Publications.

12 Klaus Didrich, Wolfgang Grieskamp, Christian Maeder and Peter Pepper. (1997). "Programming in the large: The algebraic-functional language Opal 2α". In: *Implementation of Functional Languages, Proceedings of the 9th International Workshop, IFL'97.* Springer:Berlin, Heidelberg.

13 Adele Goldberg and David Robson. (1989). *Smalltalk 80: The Language.* Addison-Wesley: Reading, MA.

14 Jeffrey Ullman. (1998) *Elements of ML programming, ML97 edition*. Prentice Hall:Upper Saddle River, NJ.

15 The internet libraries could not have been designed that early, of course. But the common identification of Java with the internet makes many people forget that the libraries are *not* the language. In this chapter the object of analysis has been Java *qua* programming language, and its libraries, even the most standard and common ones, should be considered independently of it.

16 In 1995, Microsoft was still so uninterested in the internet that Windows '95 didn't even come equipped with a sockets library. One had to wait until 1998 to see a version of Windows (Windows 98) with built-in network support. In hindsight, this lack of attention might have been a blessing. Given the notorious habit of Microsoft to impose its own creations as *de facto* standards (and given that the company has the muscle to do so), it is quite likely that Microsoft's BASIC would have defeated Java as the standard for Internet development, and we would find ourselves, today, with a world-wide web programmed largely using a proprietary language, and, to top it off, that language would be BASIC.

16 Ian Sommerville.(1989). *Software engineering*, 3rd edition. Addison-Wesley: Reading, MA, p. 225

NOTES TO CHAPTER III

1 Bureaucracies have worked under this principle for centuries, and appear to be doing quite well so, maybe, managers are right, although the case of the software industry seems to me closer to that of traditional crafts, in which the personal abilities of the people involved are a precious resource. Bureaucracies work with the predictable and in general solve standard problems using standard procedures. Software problems are seldom standard. In fact, they are seldom twice the same, and the principles that work well for bureaucracies do not necessarily work well for software development. The poor quality of industrial software (see note 1, chapter I) appears to confirm this diagnosis.

2 It is not uncommon for a certain programming communities to make reference to a specific programming environment. The GNU standards, for example (R. Stallman *et al.* (2009). *The GNU coding* standard.) takes *emacs* as its reference editor, and recommends certain guidelines in part because they make editing with *emacs* easier.

3 Geotechnical Software Services. (2008). *Java programming style guidelines*. Published on-line.

4 An exception to this rule are debugging messages, which I will not consider here.

5 Geotechnical Software Services, *op. cit.*

6 *ibid.*, but in this the source cited here is by no means alone.

7 A consequence of this observation (that will pop up under several guises in other parts of the book) is that mastering one or more foreign languages is a useful activity for a programmer. Programming is a linguistic game, and I am quite convinced that mastering Chinese may be more useful to the productivity and to the clarity of thought of a programmer (other than the Chinese, of course: they should maybe think of picking up German) than learning a new programming language or a new library.

8 This way of writing the code is only an example. In this particular case, the best way to write the code would be

```
while (a[i] != NULL) {
    i++;
}
```

The advantage of this code is that it is clearly distinguishable from the fragment

```
while (a[++i] != NULL) {
}
```

that has a different semantics from the first fragment presented here, but is syntactically almost indistinguishable.

9 Good points to start realizing the complexities involved are, among many others: William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. (2007). *Numerical recipes: the art of scientific computing*, 3rd edition. Cambridge University Press, or Gene H. Golub and Charles F. Van Loan. (1996) *Matrix Computations,* 3rd edition. *John Hopkins studies in mathematical sciences.* Johns Hopkins University Press, Baltimore, MA.

10 It goes without saying that one should never start writing optimized code. The first version of the code should be written concentrating exclusively on clarity and good structure. Then, if the performance doesn't comply with the requirements, the execution of a profiler will indicate where and whence optimization is needed. It also goes without saying that one should never start optimizing unless he does so based on the output of a good profiler. Optimizing code without profiling it first is like trying to heat up the sea with an electric blanket: a frustrating waste of time and energy.

NOTES TO CHAPTER IV

1 Not to all programming languages, though: functional programming languages have neither *for* nor *while* loops, and Prolog imposes the capitalization of variables.

2 Ed Post. (1983) “Real programmers don’t use Pascal”, *Datamation*, 29(7)

3 Sun Microsystems. (2007). *Java 2 platform Standard Edition v1.4.2 documentation*. Sun Microsystem: On-line. Entry “Object.finalize.”

4 Bill Venners. (2003). *Obects and Java: Building Object-Oriented,* Multi-Threaded Applications with Java. On-line.

5 This is a case in which the *friend* functions of C++ would come in handy.

6 In this, C++ is clearer than Java: C++ has a distinct data type to indicate references to object so that the return type of get_debate would be different in the two cases.

7 In Java, of course, it is impossible to declare structures, so it will be necessary to simulate them as classes without methods, in which all attributes are public. Since the compiler will not help in enforcing this important distinction, it is essential that it be enforced as part of the programming style.

8 This is the case when a class *A* inherits from another class *B* that has public methods so that *A* either uses the interface of *B* or re-defines some of its functions.

NOTES TO CHAPTER V

1 Designers who work in more established environments are attuned to a more traditional way of doing things, and they do indeed create designs quite independent of the implementation, using reasonably formal means. The result is often a better quality of software than it is commonly obtained in the internet world.

2 There are certain design “notations,” in the form of diagrams, that are partially formalized and in which certain structures have a corresponding program semantics. I am thinking mainly of specification diagrams such as those in UML, especially its “class” diagrams (see, e.g. Meilir Page-Jones. (2000) *Fundamentals of object oriented design in UML*. Pearson: Indianapolis)

The problem, alas, is that UML achieves this level of formalization by being, essentially, a graphic translation of a prototypical object oriented language (in fact, the derivation of the Java classes from an UML diagram can be done automatically, a fact that I regard as one of the greatest weaknesses of UML).

Because of this, I don't see much difference between writing down the structure of a system in UML and writing it down in Java: the dependencies of the design on the peculiarities of the programming language are still there.

As a matter of fact, in this circumstance I personally prefer to do the design by writing directly the "schema" of the Java classes: the result will be just as poor and, at least, I shan't be distracted by the graphics and the artifacts of UML (I find graphs terribly distracting), not to mention that one won't need a very complicated graphic program to do the design: *vi* will be sufficient.

3 Inheritance, at least if seen as a sub-typing operation, can be ambiguous. Consider the following class:

```
class Even {
    private int _key;

    public void next() {
        this._key++;
    }

    public int val() {
        return 2*this._key;
    }
}
```

This class is a partial implementation of the algebra of even numbers: the function *val* will always return an even number. If E is the domain of this data type, and another class inherits from this, then, according to the sub-typing semantics, the domain, S, of the subclass should satisfy $S \subseteq E$. Consider, however, a class that inherits from even and redefines the function val in the following guise:

```
class All extends Even {

    public int val() {
        return this._key;
    }
}
```

The class now returns all integer values, therefore $S \not\subseteq E$ (as a matter of fact, it is now $E \subseteq S$). Inheritance, in this case, does not satisfy the conditions of subtyping.

4 Ian Sommerville, *op. cit.* p. 223

5 *ibid.* p. 225; on the Jackson method, see also M.A. Jackson, *System development*, London:Prentice Hall, 1983.

6 *ibid.* p. 226

7 I will cover these points with more details in the last chapter.

8 It is immediate to realize that this is the case: any sequencial function adds a node and an edge to the graph, and its contribution cancels out in the difference *e(G)-n(G)*.

9 An exception to this rule are functions that return a quantity that depend on the time at which the function was called. The exception is acceptable as long as these functions are used sparingly: the best thing one can do is to use *one* function to determine the current time, and use the time as a parameter to other functions that do time-dependent computation. This way all functions, except one, will truly be functions.

10 See, e.g. Yourdon, E.; Constantine, L L. (1979). *Structured Design: Fundamentals of a Discipline of Computer* Program and Systems Design. Prentice-Hall, Yourdon Press.

11 I refer, here as in other places, to the number of *program points* that read or write these variables, not to the number of times that they are actually read or written during the execution of a program. That is, I am considering the program as a static entity, defined by its text, and not as a dynamic, executable artifact.

12 It is unlikely that you can have conscious experiences without language: an experience becomes conscious only when you can "fixate" it into your consciousness and, while the experience itself is pre-linguistic, this fixation is a linguistic process. You can have experiences only because you have a language

to have them in, and the development of language plays an important rôle in the development of a conscious individuality (see, for example, M. Merlau-Ponty. (1979). *Consciousness and the acquisition of language.* Northwestern University: Evanston, IL.)

13 Sun Microsystems, *op. cit.*

NOTES TO CHAPTER VI

1. The tree class will, in this case, not be a parse tree at all, of course, since the input will not be a complete sentence in the input language. Depending on the implementation, the tree might have been partially built and left somehow suspended, or there may be no tree at all, and the tree class will simply buffer the input until a complete sentence is available. But these are implementation details, and the user of the library should not have to deal with them. From the point of view of the user, the parser will always deal with a tree, albeit a possibly invalid or incomplete one.

NOTES TO CHAPTER VII

1 Maydene Fisher, Jon Ellis, Jonathan Bruce. (2003). *JDBC(TM) API Tutorial and Reference (3rd Edition). Prentice Hall* :Upper Saddle River, NJ.

2 *Introduction to JDBC*, p. 2; Stanford University (on-line)

3 This is not a typo: the numeration of the symbols in a statement starts with 1, and not with 0, as it is customary in computing. The reason for this departure from a well established and rather logical practice is unknown to me. Here as in other case I am under the impression that the obsession with standards of a large part of the internet programming environment does not extend beyond the standards that they create. At least, there are a number of cases in which a pre-existing, standard way of doing things has been ignored, and a new standard has been created.

4 The Java Data base Connection library contains, of course, may more classes, and implements many more functions. However, I am not interested in the details of the library here, but only in the general model that was used to create a structure to carry out the most common functions.

5 Consider the template

```
select * from table where ? = 3 and ? = 'foo'
```

in which the first '?' is to be replaced by an integer variable and the second by a string. One can change it into the equivalent statement

```
select * from table where ? = 'foo' and ? = 3
```

and, for the life of me, I can't see why this change should require changing the calls

```
p.setInt(1, (int) x);
p.setString(2, (String) s);
```

to

```
p.setInt(2, (int) x);
p.setString(1, (String) s);
```

since these calls may be lexically (and temporally) very far away from the point in which the statement change occurs.

6 This, of course, doesn't mean that the query is actually executed at this time: the method might as well be asynchronous, and the calling program be blocked to wait for an answer only when the results are actually needed. Every Java programmer should be quite familiar with these concepts, and, as they have no impact over the design of the interface, I will ignore them here.

7 The second alternative makes probably more sense from a practical point of view, but the first one has a greater pedagogical value.

8 A *bag* is a set in which elements can be repeated several times, that is, a bag records, along with every element, the number of times it appears. To put it in another way, a bag is a set of pairs (x,n), where x is an element of the set and n is the number of times x appears in the bag. This means, for instance that the union of bags, unlike that of sets, is not idempotent. If A is a set, $A \cup A = A$ *while, if* B *is a bag,* $B \oplus B \neq B$ *(unless* B *is the empty bag) because, for every element* $(x,n) \in B$*, we have* $(x,2n) \in B \oplus B$.

9 SQL also allows a *distinct* clause that removes copies from the result, returning a set. There is no need to consider this as a special case since the collection of sets is clearly isomorphic to the sub-collection of bags containing only one copy of each element by the isomorphism

$$\{x_1,\ldots x_n\} \rightarrow \{(x_1,1)\ldots(x_n,1)\}$$

That is, as long as we don't operate on them but only uses them to return results, sets can be represented as bags. (see, Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom. (2008). *Database Systems: The Complete Book* (2nd Edition), Prentice Hall:Upper Saddle River, NJ.)

10 The randomness of the record is a necessary semantic condition to ensure that the result is indeed a bag, and not a list.

NOTES TO CHAPTER VIII

1 Yes: the icons. In spite of their name, most of the icons in a program are not iconic at all. They are symbolic signs, which signify through a largely arbitrary cultural code, and which should be adapted to the culture that will receive the program. See, for example, Umberto Eco. (1978). *A theory of semiotics*. Indiana University Press: Bloomington, IN.

The concept of translation as the search for a functional equivalent in the target culture derived from translation theorists, especially Nida (Eugene A. Nida. (1964). *Towards a science of translating.* Brill: Leiden), interpretation theory (James S. Holmes. (1978). "Translation theory, translation studies, and the translator". In: P.A. Harguelin (ed.) *La Traduction: une profesion/ Translation: a profession*. Conseil des traducteurs et interprèts du Canada:

Montréal), and the *skopos* theory (Katharina Reiss. (1995). *Grundfragen der Übersetzungswissenschaft*. Facultas Universitätsverlag).

2 Some software producers do not use Unicode but their own code, trying to muscle it into becoming a *de facto* standard. Don't follow them in their madness. There are few areas in which standards are really useful, and this is one of them. Refuse, flatly, to follow these vendors in their delirium of greatness, and always insist on using open standards not created by any specific company.

3 Customers, I have always felt, should do their part in enforcing software quality standards by refusing to patronize companies that want them to use poorly designed software or interfaces. The presence of a flaw like this should be ground enough to convince any customer to use another rental company. It was, in my case, and I haven't rented from the incriminated company ever since.

4 The People's republic of China has, decades ago, adopted a writing form that simplified some ideograms and that is written in the western way: left-to-right, top-to-bottom.

5 Whether this is the best layout or not for Japanese or Taiwan Chinese is a question that should be answered by usage studies and linguistics. The point of this section, however, is that the answer to this question should not be given by a software designer, and that the software designer should be ready for whatever conclusion is reached in usability studies: we should allow people to implement their own solutions rather than imposing ours.

6 Göran Sonesson. (1999). Iconicity in the ecology of semiosis. Johansson, T. D, Skov., Martin, & Brogaard, Berit (eds.) *Iconicity.* NSU Press: Aarhus.

7 The transformation of contingent, cultural phenomena into universal, natural one, is one of the functions of what Barthes calls *myth* (Roland Barthes. (1970). M*ythologies.* Seuil: Paris) and Althusser *ideology* (Luis Althusser. (2008). *On ideology*. Verso: London). Both can be seen as an imposition of power.

8 Marshall Blonsky. (1985). *On Signs*. The Johns Hopkins University Press: Baltimore, MA.

The very idea of abstract color, in the way we mean it in western cultures, that is, as a function of the reflectance spectrum of a material, independent of other properties and of the uses of the material, is a cultural artifact. In the same context of the investigation of the Hanuóo color system, Conklin wrote:

> First, there is the opposition between dark and light... Second, there is an opposition between dryness or dessication and wetness or freshness (succulent) in visible components of the natural environment which are reflected by the terms *rara'* ["red"] and *latuy* ["green"] respectively. This distinction is of particular significance in terms of plant life... A shiny, wet, brown-colored section of a newly-cut bamboo is *malatuy* ["green"] (not *marara'* ["red"]). [Note: in the Hanuóo language the prefix *ma-* is the syntactic marker of an adjective.]

(Harold C. Conklin. (1955). "The Hanuóo color categories", *Southwestern journal of anthropology*. 11(4):339-44)

7 The anecdote has that Charles V, king of Spain, was asked in which language he talked during daily life. He allegedly answered: "I speak Spanish to God, French to men, Italian to women, and German to my horse."

NOTES TO CHAPTER IX

1 Poor software quality was not the only—and maybe not even the main—cause of the summary dismissal of the "dot-com". Unreasonable financial expectations and the ridiculus marketing idea that you could make a truckload of money without producing anything, simply by "going IPO" were undoubtedly stronger reasons. However, financiers and marketeers too have a very short memory and a very long learning curve, so I regard the previous remark as little consolation for the second bubble.

2 G.W.F. Hegel. (1997). *Vorlesungen über die Philosophie der Geschichte*. Reclam, Ditzingen. English Translation: G.W.F. Hegel. (2010). *The philosophy of History.* Transl. by J. Sebree. Forgotten Books (fac-simile of the Colonial press 1900 Edition).

3 That is, if the company had a clear idea of what it was producing, which was not always the case.

4 It is interesting how, at least in Europe, the political left was the first to discover the power of the internet. To the best of my knowledge, for instance, for instance, the first Italian newspaper to provide its articles on-line was “Il Manifesto,” whose tag-line reads “communist newspaper.” Much like the personal computer in the 1970’s or radio in the 1920’s, the internet was seen as a potentially revolutionary instrument, capable of subverting ancient structures of power and to distribute power in a more democratic way. Much like the personal computer and the radio before, the internet is ending up consolidating those very structures of (economic) power that it set to subvert.

5 Kevin Loney and George Koch, *Oracle 8i, the complete reference*, Osborne/McGraw Hill, p. 4; emphasis mine.

6 This cultural position is, of course, not exclusive to Java, but has extended to other programming environments although, in these cases it is often the case of programming environments internal to specific software producers. As an example, one of the companies famous for worrying more about creating dozens of (sometimes useless) features that look good into marketing brochures than for consolidating and improving the quality of its software is Microsoft although, for commercial and strategic reasons, development at Microsoft is probably not done in Java in any significant measure.

7 Cited in: Javier Ordoñez. (2001) *Ciencia, tecnología e historia,* FCE: Madrid, p. 51

8 Companies have always had a very conflictual relationship with scientists, and this is why we need to maintain academia absolutely independent of industrial needs: industry and academia are both necessary in order to use the talents of two very different kinds of people.

9 Years ago, a public phone stole my last 20 cents. (This was back then, you understand, when we still had public phones.) I asked the operator if I could have the line anyway, but she said that this was not possible. They could, however, send me a refund if I gave her my address. I still have, hanging in my house, the 20 cents check that I received in that occasion. A monument to the rigidity of large companies and to the money it costs them: the whole process of sending

me a check must have cost the company between five and ten dollars; giving me the line would have cost them 20 cents. Also a warning to myself, should habits and acquired customs ever come in the way of doing things right.

10 In the US—but other countries are following the same path—this problem is made much more serious by the abysmal state of High School education that, in some cases, is absolutely non-existent. In my father's generation, a High School graduate was a person of considerable culture, who could read Latin and Greek, and who could actively participate in a literary or a scientific debate. In my generation a High School graduate was a person of reasonable culture, who spoke some Latin, a foreign language, knew literature and science. Today it seems that things have kept degrading. This seem to be part of a trend away from worrying about the education of people: as long as they are trained to do their job properly, that is all the education they need. I keep thinking, maliciously, that part of the reasons is that educated people ask too many questions they shouldn't, but I hope somebody will show me that I am just being too cynical.

11 I don't want this insistence on the uniqueness of large universities to be taken as elitist. I would be delighted if university education were of a more uniform quality. It is just that I would see this happening by raising the quality of small universities rather and not by lowering the quality of the good ones.

12 This is not always quite true. Everybody who has had car trouble in a developing country knows that, for lack or excessive cost of spare parts, mechanics (and other artisans) often perform true miracles of practical engineering.

They have, in general, very little formal training, and nobody ever explained to them how things were supposed to be done. Like in the medieval guilds, they have learnt by working with more experienced people. Just by "being there". It doesn't speak well of our industrial culture the fact that we have almost no use for this kind of people and this kind of (slow but amazingly effective) learning anymore, that we privilege speed of training, and are giving technicians an ersatz of the training they could receive.

13 *Coeteris paribus*, of course: if you are French, your first foreign language is Spanish, and your second one is Japanese, you might qualify as an exception.

14 Edsger W. Dijkstra. (1994). *The strength of the academic enterprise.* Manuscript EWD 1175. The Dijkstra archives.

BIBLIOGRAPHY

Althusser, L. (2008). *On ideology*. Verso: London

Barthes, R. (1970). M*ythologies.* Seuil: Paris

Blonsky, M. (1985). *On Signs*. The Johns Hopkins University Press: Baltimore, MA.

Conklin, H.C. (1955). "The Hanuóo color categories", *Southwestern journal of anthropology*. 11(4):339-44

Didrich, K., Grieskamp, W., Maeder, C., and Pepper, P. (1997). "Programming in the large: The algebraic-functional language Opal 2α". In: *Implementation of Functional Languages, Proceedings of the 9th International Workshop, IFL'97.* Springer:Berlin, Heidelberg.Dijkstra, E.W. (1976). *A discipline of programming.* Prentice Hall:Upper Saddle River, NJ.

— (1994). *The strength of the academic enterprise.* Manuscript EWD 1175. The Dijkstra archives.

Eco, U. (1978). *A theory of semiotics*. Indiana University Press: Bloomington, IN.

— (1963). *Diario Minimo.* Mondadori:Milano.

Fisher, M. Ellis, J. Bruce, J. (2003). *JDBC(TM) API Tutorial and Reference (3rd Edition). Prentice Hall* :Upper Saddle River, NJ.

Garcia-Molina, H., Ullman, J.D., Widom, J.. (2008). *Database Systems: The Complete Book* (2nd Edition), Prentice Hall:Upper Saddle River, NJ.

Geotechnical Software Services. (2008). *Java programming style guidelines*. Online.

Goldberg, A. and Robson, D.. (1989). *Smalltalk 80: The Language.* Addison-Wesley: Reading, MA.

Golub, G.H. and Van Loan, C.H. (1996) *Matrix Computations,* 3rd edition. *John Hopkins studies in mathematical sciences.* Johns Hopkins University Press, Baltimore, MA.

Hegel, G.W.F. (1997). *Vorlesungen über die Philosophie der Geschichte*. Reclam, Ditzingen. English Translation: G.W.F. Hegel. (2010). *The philosophy of History.* Transl. by J. Sebree. Forgotten Books

Holmes, J.S. (1978). "Translation theory, translation studies, and the translator". In: P.A. Harguelin (ed.) *La Traduction: une profesion/Translation: a profession*. Conseil des traducteurs et interprèts du Canada: Montréal

Jackson, M.A. Jackson, *System development*, London:Prentice Hall, 1983. *System development*, London:Prentice Hall, 1983.

Kee, D. and van Jan, E.. (2004). *The Haskell Road to Logic, Maths and Programming.* College Publications: London.

Lockeed Martin Corporation. (2005). *Joint strike fighter air vehicle C++ coding standards,* document N. 2RDU00001 Rev C.

Loney, L. and Koch, G. (2007). *Oracle 8i, the complete reference*, Osborne/McGraw Hill.

Merlau-Ponty, M. (1979). *Consciousness and the acquisition of language.* Northwestern University: Evanston, IL.

Nida, E.A. (1964). *Towards a science of translating.* Brill: Leiden.

Ordoñez, J.. (2001) *Ciencia, tecnología e historia,* FCE: Madrid.

Page-Jones, M. (2000) *Fundamentals of object oriented design in UML*. Pearson: Indianapolis.

Post, E. (1983) "Real programmers don't use Pascal", *Datamation*, 29(7).

Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (2007). *Numerical recipes: the art of scientific computing*, 3rd edition. Cambridge University Press.

Reiss, K. (1995). *Grundfragen der Übersetzungswissenschaft*. Facultas Universitätsverlag

Santini, S. (1996). "The graphical specification of similarity queries," *Journal of Visual Languages and Computing*, 7(4):403—21.

Sommerville, I. (1989). *Software engineering*, 3rd edition. Addison-Wesley: Reading, MA.

Sonesson, G. (1999). Iconicity in the ecology of semiosis. Johansson, T. D, Skov., Martin, & Brogaard, Berit (eds.) *Iconicity.* NSU Press: Aarhus.

Stallman, R. *et al.* (2009). *The GNU coding* standard. GNU:On-line.

Stroustrup, B. (2000). *The C++ programming language*, special edition. Addison-Wesley: Reading, MA.

Strunk, W. Jr., White E. B. (1962). *The elements of style.* McMillan.

Sun Microsystems. (2007). *Java 2 platform Standard Edition v1.4.2 documentation*. Sun Microsystem: On-line.

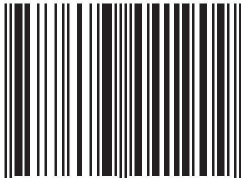
Tennet, R. D. (1976). "The denotational semantics of programming languages". *Communications of the ACM*, 19(8):437-53.

Ullman, J.. (1998) *Elements of ML programming, ML97 edition*. Prentice Hall:Upper Saddle River, NJ.

Venners, B. (2003). *Obects and Java: Building Object-Oriented,* Multi-Threaded Applications with Java. On-line.

Wirth, N. (1985). *Programming in Modula-2* (3rd corrected edition). Springer texts and monographs in computer science. Springer-Verlag:New York, Heidelberg, Berlin.

Yourdon, E.; Constantine, L L. (1979). *Structured Design: Fundamentals of a Discipline of Computer* Program and Systems Design. Prentice-Hall, Yourdon Press.



e-ISBN: 978-84-8344-221-0