19

On methodology

Entirely devoted to methodology, the next few chapters — making up part D of this book — examine how to address the issues facing object-oriented projects: how to find the classes; how not to misuse inheritance; the place of object-oriented analysis; fundamental design ideas ("patterns"); how to teach the method; the new software lifecycle. The result will, I hope, help you understand how best to take advantage of the techniques that we have now finished exploring.

It is appropriate, before going into the study of the rules, to reflect on the role of methodology in software. This will be an opportunity to define meta-rules — rules on how to make rules — which will help us devise sound methodological advice and separate the best from the rest in the methodological literature. In passing we will devise a taxonomy of rules, finding out that certain kinds are more desirable than others. Finally we will reflect on the attractive and dangerous role of *metaphors*, and take a short lesson in modesty.

19.1 SOFTWARE METHODOLOGY: WHY AND WHAT

People want guidance. The quest for Principles of Truth, which one only has to follow to succeed, is neither new nor specific to software.

The software literature, including for the past few years its object-oriented branch, has capitalized on this eagerness and attempted to offer recipes. This has resulted in much useful advice being made available (along with some more questionable ideas).

We must remember, however, that there is no easy path to quality software. Earlier chapters have pointed out several times that software construction is a challenging task. In the past few years our grasp of the issues has vastly improved, as illustrated in particular by the techniques presented in this book, but at the same time the size and ambition of what we are trying to do has been growing even faster, so the problem remains as difficult as it ever was.

It is important, then, to know the benefits and limitations of software methodology. From the following chapters and from the rest of the object-oriented literature, you are entitled to expect good advice, and the benefit of other people's experience. But neither here nor there will you find a sure-fire way to produce good software.

A comparison made in an earlier chapter helps set the limits of what you can expect. In many respect, building a software system is similar to developing a mathematical theory. Mathematics, as software construction, can be taught, including the general principles that help talented students produce brilliant results; but no teaching can guarantee success.

Not all recipe-style approaches are doomed. If you sufficiently restrict the application domain until you are left with a basic set of problem patterns, then it may be possible to define a teachable step-by-step process; this has occurred in some areas of business data processing, where methodologists have identified a small number of widely applicable solution schemes. The eventual fate of such schemes, of course, is to be subsumed by software packages or reusable libraries. But as soon as you open up the problem domain, no simplistic approach will work; the designer must exert his best powers of invention. A method will help through general guidelines, through the example of previous successful designs — also the example of what does *not* work — but not much more.

Keep these observations in mind both when reading part D and when going on to the methodology literature, where some methods make exaggerated claims. That is not necessarily a reason for rejecting them wholesale, as they may still include some useful advice; but they should be taken with a grain of salt.

A point of terminology: it has become customary in some of the literature to talk about specific "methodologies", really meaning methods (actually even less: variants of a single general method, the object-oriented method). This practice may be viewed as just another mildly irritating example of verbal inflation — such as talking of repairmen as *maintenance engineers* — but is damaging since it leads readers to suspect that if the label is inflated the contents must be oversold. This book only uses the word *methodology* in the singular and sticks to the meanings that common dictionaries give for it: the study of methods; the "application of the principles of reasoning to scientific and philosophical inquiry"; and a system of methods.

19.2 DEVISING GOOD RULES: ADVICE TO THE ADVISORS

Before going into specific rules for using object-oriented techniques, it is necessary to ask ourselves what we should be looking for. The methodologist is entrusted with a serious responsibility: telling software developers how to write their software, and how not to write it. In a field where religious metaphors come up so often, it is hard to avoid the comparison with preachers or directors of conscience. Such a position, as is well known, is subject to abuse; it is appropriate, then, to define a few rules on rules: advice for the advisors.

The need for methodology guidelines

The field of software development methodology is not new. Its origins may be traced to Dijkstra's famous *Go To Statement Considered Harmful* letter and subsequent publications by the same author and his colleagues on structured programming. But not all subsequent methodological work has upheld their standards.

[Dijkstra 1968].

It is relatively easy indeed to legislate about software construction, but the danger is great of producing rules that are useless, poorly thought out, or even harmful. The following guidelines, based on an analysis of the role of methodology in software, may help us avoid such pitfalls.

Theory

The first duty of an advisor is to base his advice on a consistent view of the target area:

> ***Theoretical Basis* methodology principle**
>
> Software methodology rules must be based on a theory of the underlying subject.

Dijkstra's example is still a good guide here. He did not just attack the Goto instruction for reasons of taste or opinion, but supported his suggested ban by a carefully woven chain of reasoning. One may disagree with some of that argument, but not deny that the conclusion is backed by a well thought-out view of the software development process. To counter Dijkstra's view you must find a flaw in his theory and provide your own replacement for that theory.

Practice

The theory is the deductive part of software methodology. But rules that would only be rooted in theory could be dangerous. The empirical component is just as important:

> ***Practical Basis* methodology principle**
>
> Software methodology rules should backed by extensive practical experience.

Perhaps one day someone will disprove this principle by devising a brilliant and applicable method of software construction through the sole power of abstract reasoning. In physics, after all, some of the most directly practical advances originated with theoreticians who never came close an experiment. But in software engineering the case has not occurred — all the great methodologists have also been programmers and project leaders on large developments — and seems unlikely to occur. Object technology in particular is among other things, an intellectual tool to build large and complex systems; the only approach, in fact, that has attempted consistently and comprehensively to reach this goal. One can master the essential concepts through taking classes, reading the literature, performing small-scale experiments and thinking further, but that is not preparation enough to give good methodological advice. The experience of playing a key role in the building of large systems — thousands of classes, hundreds of thousands of lines — is indispensable.

Such an experience must include all activities of the software lifecycle: analysis, design, implementation, and of course maintenance (the final reckoning, at which one recognizes whether the solution adopted at earlier stages stands the test of time and change, or collapses miserably).

Analysis experience, or even analysis and design experience, is not enough. More than once I have seen analysis consultants who do their job, charge their fees, and leave the company with no more than "bubbles and arrows" — an analysis document. The company then has to pick up the pieces and do the hard work; sometimes the analyst's work turns out to be totally useless as it has missed some of the most important practical constraints. An "analysis only" approach belies the fundamental ideas of *seamlessness* and *reversibility*, the integrated lifecycle that characterizes object technology, where analysis and design are interwoven with implementation and maintenance. Someone who misses part of this picture is not equipped to give methodological advice.

Reuse

Having played a key part in some large projects is necessary but not sufficient. In the object-oriented field the Practical Basis precept yields a corollary: the need for practical *reusability* experience.

Among the distinctive properties of the method is its ability to yield reusable components. No one can claim to be an expert who has not produced a *reused* O-O library; not just components claimed to be reusable, but a library that has actually been reused by a substantial number of people outside of the original group. Hence the next precept:

Reuse Experience methodology principle

To claim expert status in the object-oriented field, one must have played a key role in the development of a class library that has successfully been reused by widely different projects in widely different contexts.

A typology of rules

Next we should turn to the form of methodology rules. What kind of advice is effective in software development methodology?

A rule may be *advisory* (inviting you to follow a certain style) or *absolute* (enjoining you to work in a certain way); and it may be phrased in a *positive* form (telling you what you should do) or in *negative* form (telling you what you should not do). This gives four kinds:

Classification of methodological rules

- Absolute positive: "Always do *a*".
- Absolute negative: "Never use *b*".
- Advisory positive: "Use *c* whenever possible".
- Advisory negative: "Avoid *d* whenever possible".

The requirements are slightly different in each case.

Absolute positives

Rules of the absolute positive kind are the most useful for software developers, since they provide precise and unambiguous guidance.

Unfortunately, they are also the least common in the methodological literature, partly for a good reason (for such precise advice, it is sometimes possible to write tools that carry out the desired tasks automatically, removing the need for methodological intervention), but mostly because advisors are too cautious to commit themselves, like a lawyer who never quite answers "yes" or "no" to a question for fear of being blamed for the consequences if his client does act on the basis of the answer.

Yet such rules are badly needed:

***Absolute Positives* methodology principle**

In devising methodological rules, favor absolute positives, and for each such rule examine whether it is possible to enforce the rule automatically through tools or language constructs.

Absolute negatives

Absolute negatives are a sensitive area. One wishes that every methodologist who followed in Dijkstra's footsteps had taken the same care to justify his negatives as Dijkstra did with the Goto. The following precept applies to such rules:

***Absolute Negatives* methodology principle**

Any absolute negative must be backed by a precise explanation of why the author considers the rejected mechanism bad practice, and accompanied by a precise description of how to substitute other mechanisms for it.

Advisories

Advisory rules, positive or negative, are fraught with the risk of uselessness.

It is said that to distinguish between a *principle* and a *platitude* you must consider the negation of the property: only if it is a principle does the negation still make sense, whether or not you agree with it. For example the often quoted software methodology advice "Use variable names that are meaningful" is a platitude, not a principle, since no one in his right mind would suggest using meaningless variable names. To turn this rule into a principle, you must define precise standards for naming variables. Of course in so doing you may find that some readers will disagree with those standards, which is why platitudes are so much more comfortable; but it is the role of a methodological advisor to take such risks.

Advisory rules, by avoiding absolute injunctions, are particularly prone to becoming platitudes, as especially reflected in qualifications of the form "*whenever possible*" or, for advisory negatives, "*unless you absolutely need to*", the most dishonest formula in software methodology.

The next precept helps avoid this risk by keeping us honest:

Advisory Rules methodology principle

In devising advisory rules (positive or negative), use principles, not platitudes.

To help make the distinction, examine the rules' negation.

Here is an example of advisory negative, extracted from the discussion of type conversions (*casts*) in the C++ reference book:

> *Explicit type conversion is best avoided. Using a cast suppresses the type checking provided by the compiler and will therefore lead to surprises unless the programmer really was right.*

From [Ellis 1990].

This is accompanied by no explanation of how the programmer can find out whether he "*really was right*". So the reader is introduced to a certain language mechanism (type casts); warned, rightly, that it is dangerous and will "*lead to surprises*"; advised implicitly that the mechanism may sometimes be needed; but given no clue as to how to spot the legitimate uses.

Such advice is essentially useless; more precisely, it has a *negative* effect — impressing on the reader that the tool being described, in this case a programming language, is marred by areas of insecurity and uncertainty, and should not be trusted at all.

Exceptions

Many rules have exceptions. But if you present a software methodology rule and wish to indicate that it may not always apply, you should say precisely what cases justify exceptions. Otherwise the rule will be ineffective: each time a developer runs into a delicate case (that is to say, each time he truly needs your advice), he will be entitled to think that the rule does not apply.

Consider the following paragraph from an article about software methodology, coming after the presentation of a rather strict set of rules:

> *The strict version of the class form of the Law of Demeter is intended to be a guideline, not an absolute restriction. The minimization version of the law's class form gives you a choice of how strongly you want to follow the strict version of the law: the more nonpreferred acquaintance classes you use, the less strongly you adhere to the strict version. In some situations, the cost of obeying the strict version may be greater than the benefits.*

From [Lieberherr 1989].

It is difficult, after reading this extract, to decide how serious the authors are about their own rule; when should you apply it, and when is it OK to violate it?

What is wrong in not the presence of exceptions in a general guideline. Because software design is a complex task, it is sometimes inevitable (although always undesirable) to add to an absolute positive "*Always do X in situation A*" or an absolute negative "*Never do Y in situation A*" the qualification "*except in cases B, C and D*". Such a qualified rule remains an absolute positive or negative: simply, its domain of application is not the whole of *A*, but *A* deprived of *B*, *C* and *D*. What is unacceptable, however, is the contrast between a precise, prescriptive rule, and a vague provision for exceptions ("*in some situations, the cost may be greater than the benefits*" — what situations?). Later in the cited article, an example is shown that violates the rule, but the exception is justified in terms of ad hoc arguments. It should have been part of the rule:

Exceptions Included methodology principle

If a methodological rule presents a generally applicable guideline which may suffer exceptions, the exceptions must be stated as part of the rule.

If exceptions to a rule are included in the rule, they cease to be exceptions to the rule! This is why the principle talks about the "guideline" associated with a rule. There may be exceptions to the guideline, but they are not exceptions to the rule if the rule observes the above principle. In "*Cross the street only when the traffic lights are red, except if the lights are out of order*", the guideline "cross only on red" has an exception, but the rule as a whole does not.

This principle turns every rule of the form "Do this..." into an absolute positive, and every rule of the form "Do not do that..." into an absolute negative.

Self-doubt is an admirable quality in many circumstances of life, but not one that we expect to find in software methodology rules. One could almost argue that a wishy-washy methodologist is worse than a brilliant one who is occasionally wrong. The wishy-washy advice is largely useless, as it comes with so many blanket qualifications that you are never sure if it applies to your case of the moment; whereas if you study a methodological precept and decide that you disagree with it, you must try to refute the author's arguments with your own, and regardless of the outcome you will have learned something: either you fail, and gain a deeper, more personal appreciation of the rule and its relevance to your problem; or you succeed, and discover the rule's limitations, gaining some insights that the rule's author may have missed.

Abstraction and precision

A common theme of the last few principles is that methodological advice should be precise and directive.

This is of course more fully applicable for precise rules than for general design guidelines. When looking for advice on how to discover the right classes or how to devise the best inheritance hierarchy, you cannot expect step-1-step-2-step-3 recipes.

But even then generality and abstraction do not necessarily mean vagueness. Many of the principles of object-oriented design cover high-level issues; they will not do your work for you. Yet they are precise enough to be directly applicable, and to allow deciding without ambiguity whether they apply in any particular case.

If it is baroque, fix it

The advice on C++ type casts quoted earlier illustrates a general problem of advisory negatives: recommendations of this kind owe their existence to limitations of the underlying tool or language. For a perfect tool we would never have to give advisory negatives; every facility would be accompanied by a clear definition of when it is appropriate and when it is not — a criterion of the absolute kind, not advisory. No tool is perfect, but for a decent one the number of advisory negatives should remain very small. If in teaching the proper use of the tool you find yourself frequently resorting to comments of the form "Try to stay away from this mechanism unless you absolutely need it", then most likely the problem is what you are teaching about, not your teaching of it.

In such a case one should abandon trying to give advice, and improve the tool instead, or build a better one.

Typical phrases that signal this situation are

... unless you know what you are doing.

... unless you absolutely have to.

Avoid ... if you can.

Try not to ...

It is generally preferable not to ...

Better stay away from ...

The C/C++/Java literature has a particular fondness for such formulae. Typical is this advice: "*Don't write to your data structure unless you have to*", from the same C++ expert who in an earlier chapter was warning us against too much use of O-O mechanisms.

Advice from [Bright 1995]. See page 516.

This advice is puzzling. Why would developers write to a data structure for no reason?

Rampant Problem of Programmers Writing to Data Structures When They Don't Have To Worries US Software Industry*. Why do they do it? Says Jill Kindsoul (not her real name), a Senior Software Engineer in Santa Barbara, California: "My heart goes out to the poor things. It can feel so lonely out there in swap space! I consider it my duty to write to each one of my objects' fields at least once a day, even if it's just with its own previous value. Sometimes I come back during the week-end just for it." The actions of programmers like Jill are a growing concern for the principal software vendors, all rumored to have set up special task forces to deal with the issue.*

(*Imaginary media report.*)

Another case of trying to address language flaws through methodological advice — making language users responsible for someone else's errors — was cited in an earlier chapter: the Java designers' recommendation ("*a programmer could still mess up the object*...") against using direct field assignments $a.x := y$, in violation of basic information hiding principles. It is a surprising approach, if you think a construct is bad, and just happen to be designing a programming language, to include the construct anyway and then write a book enjoining the language's future users to avoid it.

"A simple notion of book", page 221.

The "Law of Demeter" cited earlier also provides an example. It restricts the type of x, in a call $x.f(...)$ appearing in a routine r of a class C, to: types of arguments of r; types of attributes of C; creation types (types of u in $!!\ u\ ...$) for creation instructions appearing in

"Exceptions", page 668.

r. Such a rule, if justified, should be made part of the language. But as the authors themselves imply in the quoted excerpt this would be too harsh. The rule would make it impossible, for example, to write a call *my_stack*.*item*.*some_routine* which applies *some_routine* to the topmost element of *my_stack*; yet any alternative phrasing is heavier and less clear.

For the first few weeks after the initial design of the notation of this book, years ago, multi-dot calls of the form *a*.*b*.*c* were not supported. This limitation proved insufferable and we did not rest until it was removed.

"SELECTIVE EXPORTS AND INFORMATION HIDING", 7.8, page 191.

Examination of the rationale for the Law, and for its exceptions, suggests that the authors may not have considered the notion of *selective export*, through which one can export a feature of a class *C* to specific clients having a close relation to *C*, while keeping it away from all other clients. With this mechanism, there may be no need for a Demeter-like law.

These observations yield our last precept:

Fixing What Is Broken methodology principle

If you encounter the need for many advisory negatives:

- Examine the supporting tool or language to determine if this reflects deficiencies in the underlying design.
- If so, consider the possibility of shifting over some of the effort from documenting that design to correcting it.
- Also consider the possibility of eliminating the problem altogether by switching to a better tool.

19.3 ON USING METAPHORS

ANDROMAQUE:
I do not understand abstractions.
CASSANDRA:
As you like. Let us resort to metaphors.

Jean Giraudoux, *The Trojan War Will Not Happen*, Act I.

In this meta-methodological discussion it is useful to reflect briefly on the scope and limits of a powerful expository tool: metaphors.

Everyone uses metaphors — analogies — to discuss and teach technical topics. This book is no exception, with such central metaphors as inheritance and Design by Contract. The name of our entire subject, indeed, is a metaphor: when we use the word "object" to talk about some computing concept, we rely on a term loaded with everyday connections, which we hijack for a very specific purpose.

In scientific discourse metaphors are powerful, but they are dangerous. This is particularly applicable to software, and even more to software methodology.

A colleague with whom I used to attend software engineering conferences once swore that he would walk out the next time he heard an automotive comparison ("*if programs were like cars*…"). Had he kept the pledge, he would not have attended many talks.

Are metaphors good or bad? They can be very good, or very bad, depending on the purposes for which they are used.

Scientists use metaphors to guide their research; many have reported how they rely on concrete, visual images to explore the most abstract concepts. The great mathematician Hadamard, for example, describes the vivid images — clouds, red balls colliding, "*a kind of ribbon, which is thicker or darker at the place corresponding to the possibly important terms*" of a mathematical series — to which he and his peers have resorted to solve difficult problems in the most abstract realms of analysis and algebra.

[Hadamard 1945].

Metaphors can be excellent teaching tools. The great scientist-expositors — the Einsteins, Feynmans, Sagans — are peerless in conveying difficult ideas by appealing to analogies with concepts from everyday's experience. This is the best.

But the worst also exists. If we start taking metaphors at their face value, and deducing properties of the domain under study from properties of the metaphor, we are in serious trouble. A pseudo-syllogism ("*Proof by analogy*") of the form

$$\frac{\begin{array}{c}A \textit{ resembles } B\\ B \textit{ has property } p\end{array}}{\textit{Ergo}: A \textit{ has property } p}$$

is usually fallacious because the conclusion (*A has property p*) is precise whereas the first premise (*A resembles B*) is not. What matters is how exactly *A* is like *B*, and, even more, how *A* is *un*like *B*; clearly some properties of *B* must be different from those of *A*, otherwise *A* and *B* would be the same thing (as in those stories by Borges or Pérec in which a novel or painting is about itself, or in the language that the academicians of Laputa in *Gulliver's Travels* devised from the observation that "*since words are only names for things, it would be more convenient for all men to carry about them such things as were necessary to express the particular business they are to discourse on*"). A metaphor is defined by what differs as much as by what is common. But then to justify the conclusion we have to check that *p* only involves the common part. Once Hadamard had intuited his result, he knew he had to prove it step by step using the austere rites of mathematics; and many a student of a Feynman or Laurent Schwartz has realized, when faced with the week's homework, that brilliant images are only the beginning of the process.

Swift, Gulliver's Travels, *Part 3, "A Voyage to Laputa, etc.", chapter 5.*

The more alluring the metaphor, the greater the danger of falling into twisted reasoning of the above form. Think for example of the analogy so commonly used in the reusability literature, this book included, between software components and the "chips" of our hardware colleagues, through such terms as "software IC" (coined and trademarked by Brad Cox). Up to where do we use the metaphor to help us gain insights, and where do we start confusing the real thing *A* with the metaphor *B*?

Bachelard's fascinating book on the *Formation of the Scientific Mind*, which shows some of the best minds of the eighteenth century struggling with the transition from magical modes of reasoning to the scientific method, tells a story that anyone who is ever tempted to use a metaphor in scientific discourse should keep in mind. In trying to

[Bachelard 1960].

understand the nature of air, the great physicist-philosopher Réaumur used the then common metaphor of a sponge — which, as Bachelard shows, goes back at least to Descartes. Why not? Many good physics teachers occasionally resort to such gimmicks to capture students' attention and convey a point, supported or not by a bit of clowning in the classroom or the TV studio. But then things start to go wrong: the sponge *becomes* the air!

Réaumur, in Memoirs of the [*French*] *Royal Academy of Sciences, 1731. Quoted by Bachelard, p. 74.*

A very common idea is to consider air as being like cotton, like wool, like a sponge, and much more spongious even than any other bodies or collections of bodies to which they may be compared. This idea is particularly adequate to explain why air can also become extremely rarefied, and occupy a volume considerably bigger than what we had seen it occupy a moment before.

Air is like a sponge, so air expands like a sponge! And now comes none other than Benjamin Franklin, who finds sponges so convincing as to use them to explain … electricity. If matter is like a sponge, electric current must of course be *like* a liquid that flows through a sponge:

B. Franklin, in "Experiences and observations on electricity, expressed in several letters to P. Collinson of London's Royal Society". Translated back from the 1752 French text quoted in Bachelard, p. 77.

Common matter is a kind of sponge for the electric fluid. A sponge could not receive water if the parts which make up the water were bigger than the pores of the sponge; it would only receive it very slowly if there was no mutual attraction between its parts and the sponge's parts; the sponge would fill up faster if the mutual attraction between the water's parts did not create an obstacle, requiring that some force be applied to separate them; finally, the filling up would be very fast if, instead of attraction, there was mutual repulsion between the water's parts, concurring with the sponge's attraction. This is the precise situation with electrical matter and common matter.

Comments Bachelard: "*Franklin only thinks in sponge terms. The sponge, for him,* [*has become*] *an* empirical category." He adds, with a touch of mockery: "*Perhaps, in his youth,* [Franklin] *had marveled at such a simple object* [the sponge]. *I have often surprised children being fascinated by the sight of a blotter «drinking» ink"*.

The Réaumur and Franklin quotations were not culled from a Usenet posting by an undergraduate who has yet to be taught to pour a few drops of intellectual rigor into his enthusiasm. They emanate from intellectual giants of their time, each of them responsible for decisive scientific advances. They should serve as a sobering influence when we discuss software concepts, and help us keep things in perspective the next time we see an author getting a bit carried away by his own analogies.

19.4 THE IMPORTANCE OF BEING HUMBLE

One final word of general advice as we prepare to study specific rules of design. To produce great products, designers, even the best ones, should never overestimate the value of their experience. Every ambitious software project is a new challenge: there are no sure recipes.

The design of a large software product is an intellectual adventure. Too much self-confidence can hurt. The more books you have read (or written), the more classes you have taken (or taught), the more programming languages you know (or designed), the more O-O software you have examined (or produced), the more requirements documents you have tried to decipher (or make decipherable), the more design patterns you have learned (or devised), the more design meetings you have attended (or led), the more talented co-workers you have met (or hired), the more projects you have helped (or managed), the better you will be equipped to deal with a new development. But do not think that your experience makes you infallible. In advanced software design there is no substitute for fresh thinking and creative insights. Every new problem calls for new ideas; everyone, from the seasoned project leader to the latest recruit, can have the right insight on any particular issue; and everyone can go wrong. What distinguishes the great designer is not necessarily that he has fewer bad ideas, but that he knows how to discard them, swallow his pride, and retain the good ideas whether or not he originated them. Incompetence and inexperience are obvious obstacles in the quest for the right solution; conceit can be just as bad.

No one will be surprised by these comments who has heard (although not necessarily believed) Luciano Pavarotti stating that he faces stage fright every night. One of the reasons the best people are best is that they are toughest with themselves. This rule is particularly relevant in software design, where there is always the risk of lapsing into intellectual laziness and making easy but wrong decisions, which may later be sorely regretted.

19.5 BIBLIOGRAPHICAL NOTES

The “advice to the advisors” part of this chapter is based on [M 1995b].

I first heard the definition of the difference between principles and platitudes from a talk by Joseph Gurvets at TOOLS EUROPE 1992. I owe to Éric Bezault the comment on the relevance of selective exports to the Law of Demeter.

EXERCISES

E19.1 Self-applying the rules

Perform a critique of the methodological rules of this book in the light of the precepts of this chapter. The list of all rules appears in Appendix C.

E19.2 Library rules

[M 1994a] contains an extensive set of rules, both design principles and style standards, for building library classes. Perform a critique of these rules in the light of the precepts of this chapter.

E19.3 Application of the rules

Examine the software methodology book of your choice, and the rules it gives, in the light of this chapter’s precepts.

E19.4 Metaphors on the Net

Follow for a week or two the discussions of object technology in the Usenet newsgroup devoted to it, *comp.object*. Track the use of metaphors to talk about software concepts.

Examine whether these metaphors are valuable, and whether any of them leads its author to make improper “proof by analogy” inferences.