ON THE ROLE OF METHODOLOGY: ADVICE TO THE ADVISORS

BERTRAND MEYER
Interactive Software Engineering Inc., 270 Storke Road, Suite 7
Goleta, California CA 93117, USA

1. The Need for Methodology Guidelines

The field of software development methodology is not new. Its origins may be traced to Dijkstra's famous "Goto" letter [1] and subsequent publications by the same author and his colleagues on structured programming. More recent literature, however, has not always been up to the intellectual standards of that early work.

The present discussion was prompted by a critical examination of the currently available methodological work in the object-oriented area. This leads to a number of precepts that all methodologists should follow. Even a casual look at the object-oriented literature yields many counter-examples: rules that are unusable, poorly thought out, or simply wrong.

This article focuses on the precepts rather than the counter-examples, if only to avoid the unpleasantness of having to take well-known methodologists to task.

2. Theory and Practice

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 natural to compare methodologists to preachers or to 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 first duty of an advisor is to base his advice on a consistent view of the target area:

> Precept 1 (*Theoretical Basis*):
>
> Software methodology rules must be based on a theory of the underlying domain.

Much of the current object-oriented literature, especially in the O-O analysis and design area, neglects this rule. To quote a comment made elsewhere [2]:

Books on analyis and design are full of peremptory statements: Prototype! Avoid multiple inheritance! Underline the nouns! Do not use inheritance for implementation!

No wonder software developers sometimes feel like soldiers being ordered into battle – and may wonder whether this impression is a harbinger of their eventual fate.

Were it always rooted on sound scientific ground, this aplomb would be good. But much of the advice that you will find in the methodological literature – in particular in its object-oriented subset – is based on opinions, on experience from older techniques whose lessons may not be applicable any more, or on implicit and possibly invalid assumptions. Some of it is moot or just wrong.

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 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.

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

> Precept 2 (*Practical Basis*):
>
> A software methodologist should have extensive practical experience in the field in which he is giving advice.

It is all too apparent that many O-O books have not had the benefit of such experience.

3. Reuse

In the object-oriented field the Practical Basis precept yields a more specific consequence. It is not enough to have participated in significant projects.

The cornerstone of the method is reuse, and the real test of expertise is to have produced a reused O-O library; not just components that are 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:

> Precept 3 (*Reuse Experience*):
>
> In the object-oriented field, no one can claim to be an expert who has not played a major role in the design of a reusable class library that has successfully been reused by widely different projects.

4. A Typology of Rules in Software Methodology

The next precepts will address the form of the rules given. One may distinguish four kinds of rule in software methodology:

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

5. Absolute Positives

Among the above categories, absolute positive rules are perhaps 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, as a lawyer who never quite answers "yes" or "no" to a question for fear of being blamed for the consequences if the client does act on the basis of the answer.

Yet such rules are badly needed:

> Precept 4 (*Favoring Absolute Positives*):
>
> 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.

6. 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:

> Precept 5 (*Care in Absolute Negatives*):
>
> Any absolute negative must be backed by a precise explanation of why the rejected mechanism is considered bad practice, and accompanied by a precise description of how to substitute other mechanisms for it.

7. Advisory Rules

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 (see [2] for such rules in the case of object-oriented libraries). 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 as a methodological advisor it is your responsibility 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 possible in software methodology. The next precept helps avoid this risk by keeping us honest:

> Precept 6 (*Advisory Rules*):
>
> In devising advisory rules (positive or negative), use principles, not platitudes.
>
> To help make the distinction, examine the rules' negation.

Advisory negatives are often simply a result of bad design for the underlying tool or language. An extreme example can be found in C++ manuals, and even in the official reference book for that language, which includes a large number of rules of the form "Avoid this construct if you can".

It may respectfully be remarked in such a case that beyond a certain number of such rules one should abandon trying to give advice, and improve the tool instead. Hence our last precept:

> Precept 7 (*Fixing What Is Broken*):
>
> If you encounter the need for many advisory negatives, examine the supporting tool or language to determine if some of the burden of methodological advice can be shifted over to an improvement of the underlying design.

8. References

1. Edsger W. Dijkstra: Go to Statement Considered Harmful, Communications of the ACM, 15, 10, October 1972, pages 859-866.
2. Bertrand Meyer: Reusable Software: The Base Object-Oriented Component Libraries, Prentice Hall, 1994.