Dr. Carlo Pescio

Listen to Your Tools and Materials

Published in IEEE Software, September/October 2006

Design principles are intended to guide us as we create models and ultimately code. They're often in the form, "Do this, so you'll obtain that," or "Do this, or that will happen." They discipline the way we use our materials to create an artifact, whether it's a GUI mockup, a UML diagram, or a piece of code. In this view, design is a monologue: the designer shapes the material. The material isn't talking back, except by assuming the shape we impose into it.

According to Donald Schön, professor emeritus of urban design at the Massachusetts Institute of Technology, design is *not* a monologue: it's a reflective conversation with our materials. Indeed, our materials are constantly talking back to us. We just need to listen closely.

"You're not listening!"

Not long ago, while browsing newsgroups on Google, I stumbled upon a posting that caught my attention although it was unrelated to my search. The author argued that textual specifications are easier to maintain than diagrams. He invariably found himself adding things—for example, classes—into "the most crowded spot in the diagram," so he favored text over pictures.

This might seem quite logical. We could easily argue that a textual specification has no such thing as a crowded spot and that it's therefore easier to maintain. We could also point out several issues that arise from the sequential nature of textual specifications (see Philip Armour[1] on the physical nature of models), but let's focus on the "crowded spot."

When you find yourself adding a new class in the most crowded spot of a diagram, you can curse the diagram or you can "listen" to your material. The diagram is talking to you, and it's saying, "I'm wrong! Please fix me!" A crowded spot can be the consequence of (among other things)

- *a bad layout*. In this case, the design might be right, but the presentation is lacking. Because diagrams are largely a communication device, it might not be a bad idea to simply adjust the layout.

- *a bad partitioning of responsibilities among classes*. In this case, the crowded spot is a group of strongly coupled classes. Even worse, it's an unstable group of strongly coupled classes, because you're finding yourself adding more. You'd better fix your design.

What, then, is the best representation? The one that's talking back to you, pointing out a potential design problem, or the one supinely accepting your orders?

What's our material?

Our material is knowledge, or information. We acquire, understand, filter, and structure information; and we encode it in a variety of formats: text, diagrams, code, and so on. We leave some encoding to the tools we use, from graphical encoding (I don't actually draw the class—my tool does that; I only pick the class tool) to machine-language encoding (a compiler's quintessential work).

In the end, as Armour says,[1] we obtain "executable knowledge." In this sense, the diagram isn't a material, just a representation. We use a tool to represent the material, which is intangible knowledge. However, as the diagram-versus-text example shows, representation matters. So I'll call both the information and its representation "materials." What's peculiar with software is that, in many cases, the tools and the materials have the same nature. A (class) library is often considered a tool, but it has the same nature as our own code (a material). An integrated development environment (IDE) is usually considered a tool, but in some cases—for example, in Smalltalk—the editor, interpreter, library, and our code are basically blended into a single entity.

So, even though Schön talks exclusively about materials, we can translate his ideas into the realm of software and thus extend the conversation to include the information (the actual material), its representation, and the tools we use to manipulate the representation.

Design as a reflective conversation with materials

I came across Schön's ideas a few years ago in an interview that Terry Winograd included in *Bringing Design to Software*.[2] I subsequently read Schön's work and found his theories and experiments highly relevant for software practitioners. I especially recommend *The Reflective Practitioner*,[3] which gave me terminology to communicate some experiences and thought processes we routinely encounter during software engineering practice.

Reflection in action is a central notion in Schön's observations of professional performance behavior. Reflection in action occurs as part of the performance itself. It has the character of an intimate conversation, so a personal example can better communicate it. In a recent design session, I noticed that two classes were ending up with the same, repeated associations. This is the precursor of a few "bad smells," as Martin Fowler termed programming problems like duplicated code and duplicated associations.[4] I usually deal with such problems during design, when refactoring is cheaper. An obvious transformation would be to merge the two classes, but the separation of those two classes was the cornerstone of my design. The classes would eventually evolve under different forces, and I felt I needed a strong separation of concerns.

However, this early design decision was now working against me. How interesting. I'm obviously missing another key abstraction here. Is there a subset of common responsibilities that I can factor out of the two classes, while still keeping the two concepts apart? What if ...

I was talking to my material, by changing its shape, and it was talking back—not necessarily in the way I initially expected. Schön called this kind of feedback from our material *backtalking*, and he called the process I described *reflective conversation with materials*: "thinking about what we are doing while doing it, in such a way as to influence further doing."[3]

An essential trait of reflection in action is the seamless nature of the reasoning. Over time, we learn to proceed with our performance even when the results surprise us, without any noticeable interruption. In other cases, we need to perform *exploratory experiments* ("what if" scenarios) to understand the global impact of a local change. The problem is often unclear, and we try to understand it better by observing our materials' response as we shape them. We understand by doing, in a totally seamless way.

Things don't always flow so smoothly. Sometimes we need to stop and think, outside our performance. We need to pause, review a project's state, and possibly criticize some key assumptions (in which case, it's useful to have recorded them[5,6]). We might need a radically different solution or even a complete restructuring of the problem under different terms. Schön calls this process *reflection on action*. In this case, our conversation with the material must stop for a while. We move our attention to our repertoire of strategies and tactics and our experience applying them. We might discover a lack of knowledge: Armour would say that in this moment, second-order ignorance becomes first-order ignorance,[1] a fundamental step in solving a wicked problem.

Good tools talk back softly

We spend a lot of time dealing with our tools. If we want this ongoing conversation to be pleasant, it has to be soft. Good tools provide simple, unobtrusive backtalk. Syntax coloring and error highlighting[7] exemplify immediate, gentle feedback. In contrast, a dialog box that pops out of nowhere as soon as you add a class to a diagram, asking you to provide dozens of details you don't even know yet, is a tool with poor communication skills. Such details matter over the course of a project's development.

Good libraries will also talk back smoothly. The library might not provide the formatter you need, but a properly named formatter interface will tell you that you can add your own. Look a little further inside the library and you'll find a formatter factory to deal with creational issues, so your formatter will nicely integrate within the framework. Finally, it will give you the source code for a simple formatter to further clarify what you need to do. This is the epitome of a pleasant conversation.

Bad libraries, on the other hand, will force you into unpleasant conversations, and you'll try to subvert the tool to avoid talking—for example, you might end up writing macros or tools to avoid programming directly with the library. As Diomidis Spinelli recently noted in this magazine, "Programmers using macros or wizards to save repetitive typing are programming at the wrong abstraction level."[7] In those cases, the library is telling you, "I'm wrong! I don't know your basic needs. I'm a general-purpose library. You probably need a tool closer to your business problems." The macro or wizard is just a way to keep the material from talking. It could be (barely) acceptable if the library is a third-party product you have to live with. But if the library is yours, don't shut it up. Listen to your materials, and act accordingly. You might simply need a reusable, domain-specific layer over a general-purpose library.

Sometimes the tool is talking a subtle language. It slows down our performance to remind us we need to shape it better. I've lost count of how many times I've been presented huge UML or entity-relationship diagrams that require continuous scrolling, impairing my performance in using them. The diagram was begging the designer to find a reasonable partitioning, one that could fit on a single, if large, screen. Printing the diagram on even larger paper and working from there is a common way of shutting up our material. It works in the short term, and I use it on early diagrams of large systems, when the architecture isn't necessarily stable. But I listen, and break large diagrams into subsystems as soon as I can.

We must also learn to listen carefully because the material can lie. A skillful use of composite states and history pseudostates in UML2 can turn a complex diagram into a seemingly simple picture. However, complexity will resurface later, during coding (unless you use a code generator) and especially during testing. This doesn't mean we shouldn't aim for simple diagrams, but we must understand when we're hiding complexity, rather than removing it.

Enabling backtalk in tools

Software engineering has grown into a complex discipline, yet many programmers use only an editor, a compiler, and a debugger—most often in an IDE. Programmers might seem conservative for not adopting new tools, but the concepts of flow (see the related sidebar), reflection in action, and backtalk illuminate why they might postpone changing tools.

In fact, too many tools are designed with a monologue in mind: the developer needs something, uses the tool, and gets a result. For instance, a developer who wants to remove duplicated code can run a code-clones-detection tool, get a report, study the report, and find the source files. Then he or she can manually remove duplicated code or use a refactoring editor to help with this tedious task.

Consider an alternative approach, based on the concepts of backtalk and reflection in action. Programmers aren't usually concerned with removing code clones; they're concerned with developing high-quality software, which includes removing—or better yet—preventing "bad smells." Given modern computing power, a sophisticated IDE could provide a code-clones-detection tool to run in the background, providing gentle backtalk while the programmer writes or maintains code. Backtalk should be unobtrusive, but one mouse click could turn it into a view of the duplicated portions in different files. Here the programmer could decide whether to remove the clone, and the editor could help by automating the code transformation. Although the final result is the same, the overall experience would be completely different. The monologue between the programmer and materials requires a conscious switch between tools at the risk of interrupting flow performance, while a reflective conversation with the material lets the programmer listen to a smarter tool and act on its unobtrusive suggestions.

Along the same lines, several metrics are associated with problematic code. Some of them, such as cyclomatic complexity, are almost 30 years old; others, like the Chidamber-Kemerer suite for object-oriented design, are now over 10 years old. We also have experimental guidelines for applications, such as those from the Software Assurance Technology Center at NASA Goddard.[8] Tools abound, but adoption is lagging. Once again, Schön's theory of action suggests recasting those tools into more integrated environments, offering gentle backtalk during coding, instead of a report long afterward. This seemingly trivial change would provide tools more aligned with high-performance developers' needs.

Tools such as xUnit are moving unit testing closer to a backtalk experience by using a green light to constantly tell you that your code is working properly. This also reinforces your perception of being in control and further enables a state of flow.

Enabling reflection in action

Backtalk is a necessary condition for reflection in action, just as immediate feedback is a condition for flow performance. However, reflection in action requires some additional support from our tools and materials, as well as the proper mind-set.

According to Schön, surprise is a common trigger for reflection in action. We apply a local change, but the material responds with a global change that's not aligned with our expectations. Because we reflect in the course of action—applying changes and listening to the backtalk—we need support for action-asprobe,[9] not simply for action-as-change. Most tools offer only partial support—in many cases, nothing more than an undo feature. This is extremely limiting.

Consider a common situation during a design session with a computer-aided software engineering tool: you discover some issues with your model and want to apply some radical changes to the diagram as a local experiment. You could save the model in a temporary file and start to work from there. If you use a version-control system, you can also create branches. This can be useful if your changes involve shared components.

However, if you're evaluating a few design alternatives, the entropy consequent to creating several temporary files and branches, the difficulty of comparing changes between diagrams, and the nonsupport for recording design rationale add up to an unpleasant experience. You're no longer engaged in a conversation with your material. Instead, you're trying to convince your tools to do what you need, before you can get back to your material. Flow is gone and won't come back easily.

Some tools provide an easy way around this problem. You can clone the diagram and use the clone as a sandbox inside the same project. However, you must be careful, because even some high-priced tools clone only the diagram, not the underlying model. So, any change you apply to the cloned diagram will be reflected in the model shared with the original diagram. Tools can support reflection in action much better once their creators account for the nature of localized experiments. We want to apply a change to the material and see what happens. If the result isn't good, we want to keep track of the experiment, record what went wrong, and get back to where we were. We don't want to lose track of our work, and we don't want to be bothered about creating temporary workspaces or branches. We need a simple, integrated way of conducting experiments with the materials and either recording the dead end and backtracking or using the results as the candidate product for further work.

Chatty material and style

A language that requires us to declare our variables and their types before we use them is chatty. In most cases, the compiler wouldn't need our declarations—it could figure out the type by itself. Still, the language is asking us to declare our intent so that the compiler can talk back and say, "It seems that you wanted *employee* to be of type *Person*, but you're assigning a *double* to it." The compiler is one of the few tools we all listen to. The right language can make it more useful.

Our style matters as well. In many languages, we have the choice of declaring all variables at the beginning of the outmost block—the function or method, for example—or declaring each individual variable just when we need it:

```
void f()
{
int x ;
int y ;
// many rows later
y = x * x ;
// ...
}
```

versus

```
void f()
{
int x = 2 ;
// many rows later
int y = x * x ;
// ...
}
```

Many programmers who grew up with languages allowing only the first choice, such as C, keep their habits when moving to more flexible languages, such as C++, Java, and C#. I've often heard it argued that the first style is better because you can just look at the beginning of a function—no matter how long—to know the type of any local variable.

Is this sound advice? Well, no. It's a way of shutting up our material. If we're lost in our code—that is, if we're in the middle of something and can't easily see the declaration of our variables nearby—our material is telling us something: "I'm wrong. My creator didn't give me enough structure; I'm a long list of instructions. Please split me into smaller functions!" By adopting a style that makes long functions less unpleasant, we're telling our material that we don't care—that it had better shut up because we'll find the variables right at the beginning. We shouldn't be surprised when, after a few years' silence, our material decides to stop talking. The information might be there, but it's encoded in a way we no longer understand.

Using a chatty style can also eliminate some need for conscious effort. Consider the *dependency-inversion principle*:[10] Abstractions shouldn't depend on details; details should depend on abstractions. A base class or interface is an example abstraction, with the detail being, in this case, the derived class. For a long time now, developers have adopted a de facto standard of drawing base classes on top of the derived classes. Abstraction layers are drawn bottom-up, as far as inheritance is concerned, with concrete classes on the bottom, abstract classes up on top, and inheritance dependencies going up.

Consider, however, the implications of extending this diagram layout approach to its natural end, where all the dependencies must go up:

- We would immediately recognize a circular dependency, which is often a design weakness, without any effort because we couldn't draw it without breaking this layout rule.
- All the potentially reusable, domain-neutral classes will be on the top of the diagram, with concrete layers emerging on the bottom. This is already true for the inheritance dependency, so we don't need to learn a new habit.
- Whenever we place a class in a diagram, the surroundings will tell us if that's the right place: Can we draw the dependencies in the appropriate direction? Are the neighbor classes at the same abstraction level?

Adopting this style requires breaking a few habits. Many people draw associations without any kind of rule, except trying not to cross lines into a messy web (which is obviously a good idea). So, in many cases, we see associations coming out every edge of a class in a "spider" style, as figure 1 shows.

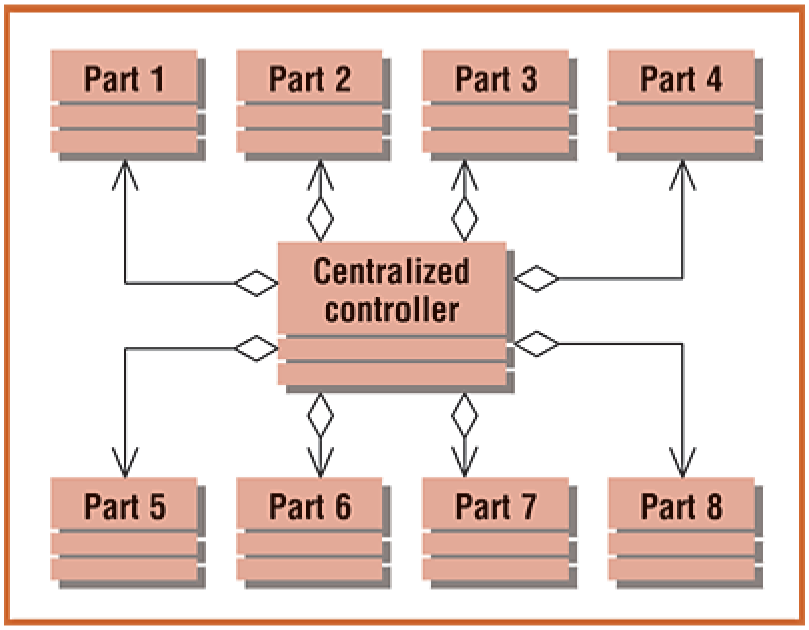

Figure 1. A "spider" style uses space efficiently but doesn't talk back enough.

The spider style is efficient in using available space. If you try to convert a similar diagram to the "dependencies go up" style, you end up using a larger area, as figure 2 shows.

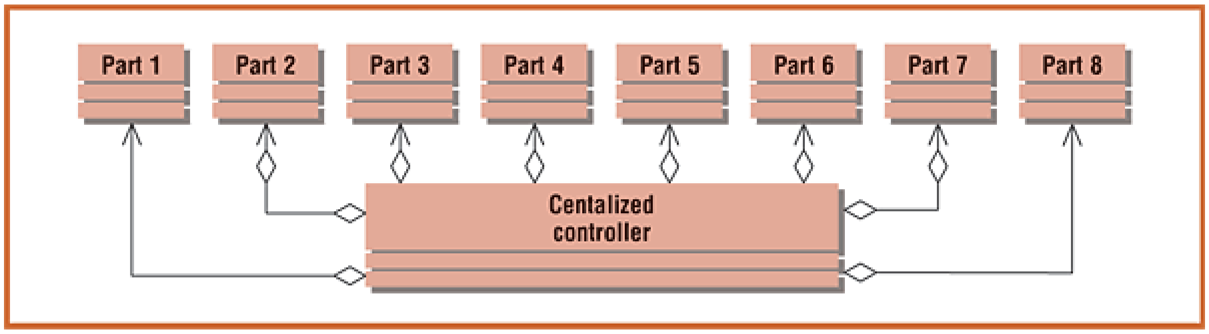

Figure 2. The "dependencies go up" style takes more space but reveals problematic design at a glance.

This might make the designer feel some discomfort—for example, the design might not fit the screen or paper. But this is good because, once again, our material is talking back: "Seems that you have a god class here. Remember, centralized control has some disadvantages. Are you sure you don't want to spread intelligence around?" The spider style, with its efficient use of space, is just a way to keep the material quiet—a good reason to move away from it.

There are several other ways to make our material talk back effectively. By using Peter Coad's suggestion of coloring the class diagram,[11] we can literally look at the diagram for the proper balance of colors (read: responsibilities). I have successfully added a few colors to Coad's set (which is intended more for analysis than design) to represent fundamental concepts such as extension points. I've also applied colors to other diagrams such as component diagrams. I'm experimenting with color to represent stereotypical interactions in the class diagram and so to suggest the dynamic view inside the static view and give the material more expressive power.

Conclusion

A stronger appreciation of Schön's work could help us create better tools and materials—and use them better, too. Materials that support reflection in action give us immediate, visible, yet gentle backtalk, and they influence the kind of tools we can build to shape them. For example, the syntactic simplicity of Java and C#, together with their support for reflection, have enabled a wealth of tools compared to, for instance, C++. Therefore, when designing new materials (a programming language, a design notation), we should ask whether the material can talk back and whether we can easily build tools on it to foster further backtalk.

We also need tools with built-in support for the tactical reasoning typical of reflection in action, where exploratory manipulation is as common as backtracking, yet capable of recording dead ends for future reflection on action. Along the same lines, we need tools that make recording design rationale an easy and integral part of our performance, again to support reflection on action.

Schön's models of action are also valuable when teaching design. We already teach principles, patterns, and techniques, but we can go one step further and make students aware of their own cognitive processes. This would help them turn a gut feeling into a more articulate form of reflective conversation with materials.

Acknowledgments

I thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] P.G. Armour, *The Laws of Software Process,* Auerbach Publications, 2004.

[2] T. Winograd ed., "Reflective Conversation with Materials: An Interview with Donald Schön by John Bennett," *Bringing Design to Software,* Addison-Wesley, 1996.

[3] D. Schön, *The Reflective Practitioner,* Basic Books, 1983.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

[5] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software,* vol. 22, no. 2, 2005, pp. 19–27.

[6] C. Pescio, "When Past Solutions Cause Future Problems," *IEEE Software,* vol. 14, no. 5, 1997, pp. 19–21.

[7] D. Spinellis, "Dear Editor," *IEEE Software,* vol. 22, no. 2, 2005, pp. 14–15.

[8] L.H. Rosenberg, "Applying and Interpreting Object Oriented Metrics," *Proc. Software Technology Conf.,* NASA, 1998; http://satc.gsfc.nasa.gov/ support/STC_APR98/apply_oo/apply_oo.html.

[9] C. Argyris, R. Putnam, and D. McLain Smith, *Action Science,* Jossey-Bass, 1985.

[10] R. Martin, "The Dependency Inversion Principle," *C++ Report,* May 1996, pp. 61–66.

[11] P. Coad, "Show Your Colors," *The Coad Letter,* no. 44, 1997.

Carlo Pescio is a consultant and mentor for several companies across Europe, where he has designed software for medical devices, industrial process control, banking, finance, CAD, and several other fields.
His research interests focus on software design, team dynamics, and diagrammatic reasoning.
He received his Laurea degree, magna cum laude in computer science, from the University of Genoa. He is a member of the IEEE Computer Society, the IEEE Technical Council on Software Engineering, and the ACM. Contact him at Via Bernardo Forte 2/3, 17100 Savona SV, Italy; pescio@eptacom.net.

Flow, Performance, and Reflection in Action

Most software professionals have experienced a state of mind where all their attention is focused on a particular task, like design or coding, and external stimuli, including time, seem to disappear. Tom DeMarco and Timothy Lister1 described this state of flow as "a condition of deep, nearly meditative involvement. ·· There is no consciousness of effort; the work just seems to, well, flow."

The flow state often correlates to high-performance and creative experiences. The psychologist Mihaly Csikszentmihalyi has explored it in depth.2,3 He reported a set of common conditions that seem to enable it:

1. clear goals
2. unambiguous and immediate feedback
3. skills that just match challenges
4. merging of action and awareness
5. centering of attention on a limited stimulus field
6. sense of potential control
7. loss of self-consciousness
8. altered sense of time
9. autotelic (or "self directed") experience

Condition 2 is strongly related with Donald Schön's concept of backtalk. The material is talking back while we're shaping it. This contrasts sharply with other forms of feedback, such as creating and running a prototype (see the sidebar "Backtalk versus Feedback").

Condition 4 is like Schön's concept of reflection in action: the separation between the action and the reflection disappears. This is opposed to reflection on action, when you have to stop and think about what you're doing outside the performance itself.

Condition 5 has some interesting implications for tool (and material) designers. Integrated tools are inherently better suited to enable this condition. Uniform notations for activities such as analysis and detailed design also reduce the conceptual size of the stimulus field. The best tools and materials enable conditions 2 and 5—by design.

Finding flow isn't easy, and we should avoid losing focus. Condition 5 is easily broken when we encounter a problem that we're not ready to face and so move away from our task to seek more information. We must learn to postpone distractions. I often use visual cues to remind myself of a surprise when I'm not prepared to deal with it immediately. For instance, I might give a class an awful color to remind myself that the class needs some reworking later, when the surroundings have taken their shape. This lets me restrict the locus of attention and go ahead without interruptions, at least for a while.

References

[1] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams,* 2nd ed., Dorset House, 1999. [2] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience,* Harper and Row, 1990. [3] M. Csikszentmihalyi, *Finding Flow,* Basic Books, 1997.

Backtalk versus Feedback

Software developers routinely get some form of feedback from their products. Feedback is similar to backtalk, because in both cases we're learning while building the product. Backtalk, however, is an immediate form of information, which we get from the material as *we shape it*. This is a necessary condition to engage in a reflective conversation.

Some forms of feedback, therefore, don't qualify as backtalk. Rapid prototyping, for instance, is a way to obtain feedback from the product (Is it doing what I intended it to do?) and from the potential users (Do they like it; will they use it?). Such feedback, however, isn't backtalk because it's not engaging us in a reflective conversation with the material as we shape it. In fact, in rapid prototyping we can still see distinct phases of problem setting (What should I do?), problem solving (How do I get that?), and finally feedback (Did I do it right?).

Backtalk takes place at a finer granularity. The material is talking during problem setting and problem solving. The final feedback comes from the *product*, not from the material.

An executable specification is closer to engaging us in a reflective conversation. However, being executable is not a necessary or sufficient condition. We listen to backtalk to understand the functional and nonfunctional implications of our design choices. Being executable can help with the first but not with the second.

We can also obtain feedback through a design or code review. Once again, this can provide important information, but it's not a form of backtalk: we're getting information from other human beings (or possibly from static or dynamic analysis tools), not from our material and not while we shape it. Pair programming provides timely feedback, but from another human being, not from material.

Case-based reasoning is another approach to leverage knowledge.1 CBR tools usually require the user to set the problem (the "pull model"); then they provide suggestions drawn from knowledge bases. The pull model doesn't engage the user in a reflective conversation. A "push model" CBR, where suggestions come directly from the tool working in the background while we shape our material, can be seen as a form of backtalk, provided that it's unobtrusive enough to avoid distracting us from the main course of action.

Reference

[1] I. Rus, M. Lindvall, and S.S. Sinha, *Knowledge Management in Software Engineering,* tech. report, Data and Analysis Center for Software, 2001.