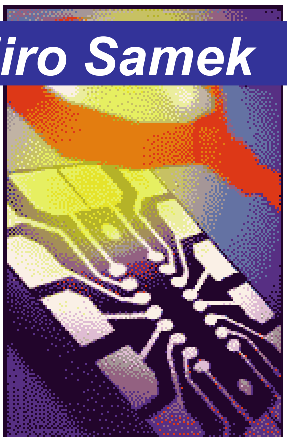
Miro Samek



C/C++ Users Journal
Advanced Solutions for Professional Developers

Patterns of Thinking

Names, metaphors, better programming, and the politics of language.

Several years ago, I once attended an "Object-Oriented Analysis and Design" training. As most such courses go, the instructor began with brushing up on the fundamental OO concepts. When explaining inheritance, the instructor spontaneously compared inheriting from a class to passing traits from parents to the offspring in a family. At first, this "family tree" metaphor seemed to make sense, and most attendees nodded approvingly. However, when the instructor discussed inheritance further, she mentioned that inheritance establishes the "is a" (is-a-kind-of) relationship between the child and the parent classes. The child class has all the attributes and behaviors of the parent class but is typically more specialized. At this point, one attendee objected that he did not have all of his father's attributes and behaviors, and he did not feel like a more specialized version of his father. For example, he had no coronary disease "attribute," did not support the smoking "operation," and did not feel fully substitutable for his father (or grandmother, for that matter). Consequently, the Liskov Substitution Principle did not apply to a family tree as it should to a class hierarchy.

The "family tree" metaphor was not working very well, although it inspired much of the terminology used in OOP (parent, child, ancestor, sibling, class family, etc.). I do not recall exactly how the instructor helped resolve the confusion, but I do remember that she did not offer any better analogy for class inheritance. The instructor probably made this analogy up "on the fly" without paying much attention. However, the other possibility is that the inaccurate analogy reflected *her* way of thinking about inheritance. Both possibilities offer interesting insights. First, the story indicates that analogies and metaphors appear in software frequently and spontaneously, but their role is usually underestimated. That is unfortunate because analogies and metaphors, when chosen correctly, may help students climb the learning curve far more quickly. Conversely, severe difficulties arise when the analogies chosen are incomplete, inaccurate, or irrelevant. The second insight from the story is that analogies and metaphors offer a unique way to quickly "look" into somebody's mind—the ability critical in the software business.

Coming back to the unfortunate "family tree" metaphor, I participated in a group discussion on this matter during a break. In a short brainstorming session, we came up with a much better metaphor based on the classification of living organisms in biology. This "biological classification" metaphor had no problems explaining correctly the "is a" relationship between classes and what it means that a subclass is a more specialized version of the superclass. For instance, a class `RedRose` *is a* kind of `Flower`, and a `Flower` *is a* kind of `Plant`. `RedRose` has all the attributes of a `Flower` and a `Plant` but is more specialized. In a group of `Flowers`, some of them may very well be `RedRoses`, because it always makes sense to substitute a `RedRose` for a `Flower`. The "biology classification" metaphor provides countless more examples that illustrate virtually all the nuances and concepts encountered in class inheritance, including polymorphism. Such a metaphor is more valuable than one might think because it establishes the correct *thinking* pattern about the fundamental software concept.

System Metaphor and Conceptual Integrity

Metaphors and analogies have been used in software for decades. For example, it is not an exaggeration to say that the venerable desktop metaphor, the spreadsheet metaphor, or the shopping cart metaphor have changed civilization. All these are examples of *system* metaphors, that is, metaphors that provide a general idea of how a whole system works. The system metaphor has been the subject of many studies, books, and papers [1]. Recently, eXtreme Programming (XP) has brought the system metaphor once again to the attention of developers by elevating it to a critical programming practice [2].

Contrary to widespread beliefs, the primary intention of the desktop, the spreadsheet, or the shopping cart metaphors is *not* to exactly imitate office furniture, accounting pads, or physical aisles in a supermarket. Indeed, users do not expect the computer screen to be exactly as hard to erase or change as the actual paper is or the virtual aisles of an online store to be precisely as cluttered and hard to navigate as real ones are. In other words, the function of a system metaphor is not a strict mapping between the source and target domains. Instead, the critically important role of a system metaphor is in establishing a consistent user's *conceptual model*, which represents what the user is likely to think, and how the user is likely to respond.

As observed in XP, however, a consistent conceptual model is not just valuable for the end-users; it is invaluable for software developers because it provides the most challenging part of the design. With such a model in hand, designers do not need to invent potentially inconsistent policies and behaviors—they can consult the real-life model to see how *it* solves various problems. The system metaphor thus serves as an objective arbiter in resolving various design conflicts and ultimately guards the *conceptual integrity* of the design.

Metaphors and Learning

While the system metaphor can vastly benefit software developers, it is primarily a side effect of conceptual integrity, which is then reflected in a better software structure. However, other types of metaphors and analogies, specifically aimed at guiding the learning process and improving the understanding of new concepts, can benefit software developers much more directly.

In order to appreciate how such "concept metaphors" influence learning new software concepts, it is interesting to realize how humans organize and modify their knowledge. When people learn new things, they automatically try to map new concepts to familiar ones in the spontaneous process of "making analogies". A problem occurs when these spontaneous analogies are incorrect. The new knowledge interferes with the old knowledge (learning interference), and the learning process is more complicated than it would be if the individual did not have the conflicting knowledge in the first place.

For example, a study on retraining procedure-oriented developers in object-oriented programming [3] shows clearly that software developers are more likely to recall rather than forget their existing knowledge. This research revealed that attempts to map familiar procedural concepts often resulted in inaccurate analogies, which increased the time it took the learners to use the object-oriented development model effectively. The study further emphasized the role of the instructor in helping the learners make correct analogies. Instructors who are aware of any incorrect analogies their students make can guide the students toward understanding which aspects of the analogies are correct and which are incorrect. It is much better for the instructor to offer correct analogies explicitly rather than to rely on the spontaneous analogies made up (consciously or subconsciously) by the students. In this spirit, I offered the mapping between procedural and object-oriented programming in an earlier installment of this column [4]. Similarly, in the June installment [5], I showed that many familiar OOP concepts have their counterparts in hierarchical state machines.

Of course, the benefits of concept metaphors for improving the learning process apply more generally than just to entirely new concepts. Metaphors and analogies have a unique capacity to shed new light on old ideas and thus help us discover new aspects of already familiar concepts. For example, the "assertions as fuses" analogy [6] offers a novel view of assertions as a controlled-damage protection mechanism, which complements the "assertions as contracts" metaphor that underlies the design by contract methodology.

Mining Concept Metaphors

In response to a recent installment of this column dedicated to hierarchical state machines [5], a reader wrote me: “*while [traditional] state machines supply an abstract representation of a system in the real world, the concept of states within states does not correspond to any reality*." At first, this statement stroke me as being exactly backwards, because I always thought that the only "real" reactive systems correspond to hierarchical state machines, of which the flat (non-hierarchical) versions are merely special, somewhat degenerated corner cases. After a while, however, I realized that most probably, the reader's mental model of a state machine and mine were based on different metaphors. Metaphors must be involved here since only through analogies and metaphors abstract software concepts can correspond to any "reality" and thus can gain meaning to us. The whole discussion of what it means to "understand" a software concept borders on philosophy. However, given that analogies and metaphors can be beneficial for understanding and mastering new concepts, the pragmatic question is: What is a good metaphor for a state machine?

Just as in the case of the "biological classification" metaphor for class inheritance mentioned earlier, a good strategy for mining concept metaphors seems to be to look for examples of the concept successfully applied in existing systems (e.g., biological classification as an example of class taxonomy).

The Quantum Metaphor

It should come as no surprise that the source domain for the abstract concept metaphor, such as state machine, is most likely to be science and not the familiar physical objects or everyday activities, as in the case of the much less abstract system metaphor. Therefore, the problem of finding a good metaphor for the state machine concept boils down to finding a good example of stateful behavior in other branches of science. As it turns out, such behavior is fundamental and pervasive in the microscopic world of elementary particles, atoms, and molecules, as explained by the laws of quantum physics [7].

A possible critique of the “quantum metaphor” for state machines might be that most programmers are not familiar enough with the source domain — contemporary physics in this case. However, the physics background necessary to benefit from the metaphor is really at the level of high-school textbooks and popular science articles. I believe that most programmers have heard about electrons, atoms, quantum numbers, quantum states, or quantum leaps.

For anybody who makes an effort to understand the basic principles governing the microscopic world, the "quantum metaphor" might be precious. It has a rich structure and correctly explains many specific aspects of state machines (all these are criteria that Kim Halskov Madsen proposed for choosing successful metaphors [1]). The most interesting observation is that virtually all quantum states are hierarchically decomposed (degenerated in physics language). Degeneration is always an indication of some *symmetry* in the system. For example, the degeneration of the substates of angular momentum comes from the spherical symmetry of the atom. Simple experiments, such as subjecting atoms to magnetic fields, lower the symmetry and partially remove the degeneration allowing us to "see" the state hierarchy (Zeeman effect). The concept of nested states corresponds thus to physical reality and is not just theoretical.

I found the quantum metaphor very valuable, and in fact, it has shaped my understanding of state machines. As always with a good analogy, the quantum metaphor explains the fundamental character of state nesting and its function in capturing and representing the symmetries existing in the system. This observation alone (i.e., the link between state nesting and the symmetry of a system) is already a sufficient payoff from the metaphor. However, there is so much more to learn from the quantum analogy. For instance, the mechanism of quantum leaps provides excellent insights into the proper role of the run-to-completion (RTC) processing model in state machines. RTC makes the system unobservable "in-between" states. Likewise, the virtual particle exchange mechanism provides an excellent communication model among concurrent state machines (active object computing model).

The main point is that in contrast to system metaphors, "concept metaphors" involve more abstract thinking and typically draw from science. In other words, do not sign off on all of the stuff you once learned in math, biology, chemistry, or physics because it can come in handy in your software work. The results might surprise you.

Metaphors and Social Intelligence

The beneficial impact of metaphors on improving communication within software teams has been observed already for the system metaphor. As a natural byproduct, the source domain of such a metaphor provides a standard system of names for objects and their relationships. That can become jargon in the best sense: a powerful, specialized, shorthand vocabulary for all software project stakeholders.

However, actual case studies indicate that while metaphorical representations indeed dominate software discussions, it is not system metaphors that are most commonly applied. As it turns out, the most frequently used are anthropomorphic comparisons known as naïve psychology [8]. Developers often speak in terms of what a software component "needs," "knows," "is trying to do," or "thinks" is happening. These anthropomorphic metaphors (identified in some 70 percent of all spoken representations in the Herbsleb's study) strike many as sloppy and imprecise. Dijkstra, for example, has gone so far as to propose that computer science faculty implement a system of fines to stamp out such sloppiness among students.

However, the new research suggests that instead of fighting the "sloppy" metaphorical representations, software practitioners and teachers could use them to their advantage. Naïve psychology seems to be a special kind of metaphor. It exploits a specific cognitive capability that evolved due to natural selection favoring individuals more capable of handling the increasingly complex social interactions in early human societies [9]. These natural skills, which develop without any special instruction in all intact humans, match many tasks encountered in software development. In designing or understanding a complex, multi-component software system, it is vital to be able to keep track of: what state each component is in, what it is "trying" to do, what it "knows" about the rest of the system, how is it likely to behave, how is it likely to fail, and so on. The skills of naïve psychology seem to allow a software developer to keep track of this sort of information with little effort because of a built-in capability known as social intelligence.

The pervasive use of naïve psychology in virtually all software discussions, although certainly not part of any standard software engineering curriculum, indicates how critical role social intelligence plays in collaborative software development. Perhaps nowhere is it more evident than in pair programming advocated by XP. In pair programming, the software is created in small incremental steps. Ideas move back and forth between the more stable media such as a computer screen or a whiteboard and the transitory verbal medium of conversation between the two developers. The point is that most of the critical, creative work occurs during these periods of discussion when the primary representation is just the spoken language. If you have ever tried pair-programming, you must have noticed how much of the discussions revolve around anthropomorphic representations and role-playing. By insisting on informal, high-bandwidth, face-to-face communication between pairs of developers, XP makes much better use of social intelligence than other "heavyweight" methodologies. Indeed, due to engaging social intelligence, a pair of programmers typically creates more code and better code than these two developers working separately.

Another way of tapping the potential of social intelligence is to employ programming paradigms and techniques that offer more opportunities to use naïve psychology than others. For example, the object-oriented method encourages partitioning the problem into specialized, encapsulated "society" of objects, which collaborate on a common task. More specific to embedded software development, the active object computing model [7] fits this paradigm even better because autonomous stateful active objects (actors) resemble a human society closer still [10]. There is growing evidence that such programming models and techniques reduce the need for clarifications in discussing design ideas.

An Adjunct to Existing Methods

Most software engineering tasks are sufficiently complex that many kinds of methods and specific cognitive abilities of the brain must be used in turn, or better yet—simultaneously. For example, the particular value of visual languages (such as UML), lies in tapping the potential of high-bandwidth "spatial intelligence", as opposed to "lexical intelligence" used with textual information. While clearly, not all visual representations are helpful for all purposes, when a visualization matches a task well, the visual processing capabilities of the brain allow a tremendous amount of information to be conveyed in compact form. Moreover, this effect is not just limited to faster information transfer; the quality of the very *thinking* has been found to be improved [11].

In this context, metaphorical representations based on naïve psychology are an essential adjunct to other techniques because they explicitly use another powerful cognitive capability of the human brain identified as the "social intelligence." In this sense, the old saying "a picture is worth a thousand words" can be modified to "a metaphor is worth a thousand words."

Other types of metaphors (concept metaphors and system metaphors to some degree) seem to complement the design-pattern approach to software development. Design patterns excel in capturing solutions to recurring software problems, but they have not been able to capture the patterns of thinking that experts use. Metaphors and analogies offer a unique view into experts' minds and convey the particular way the experts see and understand the problem.

A Word of Caution

The special cognitive abilities of the human brain, such as the "social intelligence" (or the "spatial intelligence", for that matter) did not evolve to solve software problems. Analogies and metaphors (or visualization techniques) are just a trick to take advantage of these powerful "hardware accelerators" of the brain to solve different problems than those for which they were designed. The catch is that it might not work with all people or might work differently for different people. If the "quantum metaphor" for state machines does not do the trick for you, move on, and try to understand hierarchical state machines some other way. One excellent explanation of how state hierarchy arises naturally in every non-trivial reactive system appeared as a point/counterpoint debate between Steve Mellor and Bran Selic, published in the March 2000 issue of the *ESP* magazine [12].

End Around Check

I am sure you have used system metaphors, concept metaphors, or naïve psychology before, but perhaps you did not fully realize the fact. I hope that this article will make you more aware of the different roles metaphors and analogies can play in software development and that there is more to it than just the venerable system metaphor. I hope you start paying more attention to how you and others use metaphors to better utilize the natural god-given abilities built into your brain. I hope you start thinking more about metaphors, learn their limits, and remember their critical aspects. Perhaps you will start collecting metaphors?

In doing my online research for this article, I was amazed at the number and quality of publications related to metaphors in software. A slow but noticeable groundswell is developing in favor of metaphorical thinking in software. It appears that metaphors and analogies have the potential to one day become as hot a topic as design patterns are today.

I want to close with the quotation from Stefan Banach, a Polish mathematician who founded modern functional analysis and made significant contributions to the theory of topological vector spaces. He wrote: *"Good mathematicians see analogies between theorems and theories. The very best ones see analogies between analogies."*

References

[1] Kim Halskov Madsen, "A Guide to Metaphorical Design", *Communication of the ACM*, December 1994.
[2] Kent Beck, *eXtreme Programming Explained*, Addison-Wesley, 2000.
[3] Mary L. Manns and H. James Nelson, "Retraining Procedure-Oriented Developers: An Issue of Skill Transfer", *Journal of Object-Oriented Programming, Vol. 9, No. 7*, 1996.
[4] Miro Samek, "OOP in C" sidebar in "Who Moved my State?", *C/C++ Users Journal*, April 2003.
[5] Miro Samek, "Déjà Vu", *C/C++ Users Journal*, June 2003.
[6] Miro Samek, "An Exception or a Bug?", *C/C++ Users Journal*, August 2003.
[7] Miro Samek, "Quantum Programming for Embedded Systems: Toward a Hassle-Free Multithreading", *C/C++ Users Journal*, March 2003.
[8] James D. Herbsleb, "Metaphorical Representation in Collaborative Software Engineering", Available online at <www.cs.cmu.edu/~jdh/collaboratory/research_papers/WACC_99.pdf>
[9] Nicholas Humphrey, *The Inner Eye: Social Intelligence in Evolution*, Oxford Paperbacks, 2002.
[10] Arthur Allen, "Actor-Based Computing: Vision Forestalled, Vision Fulfilled" <www.charis.com/-documents/agent98.pdf>
[11] David Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development", *Computer*, January 1992, available online at <www.wisdom.weizmann.ac.il/~dharel/SCANNED.-PAPERS/BitingTheSilverBullet.pdf>
[12] Steve Mellor and Bran Selic. "Point/Counterpoint", *Embedded Systems Programming*, March 2000. Available online at <www.embedded.com/2000/0003/0003feat1.htm> and <www.embedded.com/-2000/0003/0003feat2.htm>. ❑

Miro Samek is the author of *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002. He is the lead software architect at IntegriNautics Corporation (Menlo Park, CA) and a consultant to industry. Miro previously worked at GE Medical Systems, where he has developed safety-critical, real-time software for diagnostic imaging X-ray machines. He earned his Ph. D. in nuclear physics at GSI (Darmstadt, Germany) where he conducted heavy-ion experiments. Miro welcomes feedback and can be reached at `miro@quantum-leaps.com`.