

Computer Science Adapts Techniques from Engineering and Mathematics

Karl Lieberherr, Northeastern University, Boston

lieber@ccs.neu.edu

www.ccs.neu.edu/home/lieber

in response to:

<http://www.ccs.neu.edu/research/demeter/papers/mary-cs/workshop/mary-shaw.html>

Computer Science heavily relies on applying ideas from mathematics and engineering. We demonstrate this view with a few examples from algorithms to software construction.

We first investigate a simple principle from engineering and how the principle has been applied in computer science and the innovations it has suggested. The principle says that each component should only know about closely related components. We call this principle the generalized Law of Demeter (LoD) [LoD, www.ccs.neu.edu/research/demeter] and view it as an engineering principle. The LoD is an application of the more general principle of low coupling between components. The important motivation behind the LoD is that developers and maintainers of a component should only have to know about a limited set of other components to reduce the maintenance burden. The clause "closely related" has the intention to cut down on the number of other components one has to think about when considering the current component.

The LoD that is known in object-oriented programming [LoD] is an application of the generalized LoD where the components are methods and closely related methods are basically methods of argument and immediate part objects.

The LoD has influenced our work on Aspect-Oriented Programming (AOP) [Kiczales, Huersch-Lopes]. AOP is about localizing the implementation of crosscutting concerns. An ad-hoc implementation of a crosscutting concern leads to scattering of the concern's code across multiple classes. When implementing a functional concern that involves the traversal of several objects and we follow the LoD in the straight-forward way, we also end up with code that is scattered across many different classes. In this view, the LoD is anti-AOP in that LoD promotes program maintenance at the cost of increased scattering. However, not following the LoD and encapsulating all code in the source class of the traversal in the usual way is even worse. It leads to tangling and duplication of class graph information in the source class. This scenario shows that encapsulation of a crosscutting concern may not be sufficient. In addition, we need to free ourselves from the details of another concern, in this case the class graph or structural concern to obtain a system that is easy to build and maintain.

In order to remove the strong coupling to the structural concern, we put the structural concern in an abstracted form into the interface of the component that implements a functional concern. This allows us to filter out implementational details unrelated to the

current behavior. We put into the interface only the information that is absolutely needed. To use the component, we have to map this abstracted class graph into a larger more specialized class graph. Does such a design formulated with an abstracted class graph follow the LoD? Yes, all classes mentioned in the abstracted class graph are considered closely related and we may use their interfaces. The LoD and clean encapsulation of crosscutting concerns can coexist very well by applying Polya's inventor paradox.

Polya's inventor paradox [PolyaSolve] from mathematics is a heuristic that invites us to look for situations where to solve a more general problem is easier than solving the given concrete one. If the mapping of the abstracted class graph to the concrete class graph can be expressed very succinctly, we have here an application of Polya's inventor paradox. Writing the code for the abstracted class graph plus the mapping code is easier than writing the code directly for the concrete class graph. This application of Polya's inventor paradox is known as Adaptive Programming (AP). Instead of writing code for one data structure, we write code for an entire family of potentially very different data structures. Polya's inventor paradox is an example of the reduction technique mentioned earlier where the transformation goes from specific to general.

The LoD has brought us to neatly encapsulate crosscutting traversal-style concerns using an approach that at first sight seems to violate the LoD. Once we had this in place, other crosscutting concerns followed. The synchronization and data transfer concerns were localized in [Lopes 97]. This led to the first version of AspectJ that later became a general purpose AOP language [AspectJ].

Above we have used a principle of software architecture: to make all interfaces explicit, resulting in components with ports linked by connectors. The mapping code from the abstracted class graph to the concrete class graph is a part of the connectors.

Now we switch to another topic showing how a principle from mathematics is adapted to CS problems. Mathematics uses the concept of problem reduction that reduces one problem to an equivalent but a simpler problem that is more amenable to formal treatment. This technique is also very useful in computer science, for example, it is used to find computational thresholds or to convince someone that it is unlikely to find an efficient algorithm for certain problems. As an example, consider the set of equation systems where all equations are of the form $x+y+z = 1$. All variables are either one or zero and our goal is to find an efficient algorithm that satisfies the optimum fraction of the equations in a given equation system. The technique of problem reduction is used to show that the fraction $4/9$ of the equations can be satisfied by an efficient algorithm. It is also used to show that it is unlikely that there is a better algorithm because the set of equation systems where the fraction $4/9 + \epsilon$ can be satisfied is NP-complete for arbitrary small ϵ [GoldenRatio].

Computer Science reuses techniques from engineering and mathematics and adapts them to the new context. We have discussed how a low coupling principle from engineering and techniques from mathematics have influenced the development of algorithms and software construction methodology from the Law of Demeter to AOP.

References:

[AspectJ]

Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mike Kersten and Jeffrey Palm and William Griswold, An Overview of AspectJ, European Conference on Object-Oriented Programming, 2001, Budapest, Springer Verlag.

[GoldenRatio]

Karl J. Lieberherr and Ernst Specker, Complexity of Partial Satisfaction, Journal of the ACM, 1981, pages 411-421, Vol. 28(2).

[LoD]

Karl J. Lieberherr and Ian Holland, Assuring Good Style for Object-Oriented Programs, IEEE Software, 1989, September, pages 38-48.

[Lopes 97]

Cristina Isabel Videira Lopes, D: A Language Framework for Distributed Programming, Northeastern University, 1997, 274 pages.

[PolyaSolve]

George Polya, How to solve it, Princeton University Press, 1949.

[Kiczales]

Gregor Kiczales and John Lamping and Anurag Mendhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming, 1997, pages 220-242, Springer Verlag.

[Huersch-Lopes]

Walter L. Huersch and Cristina Videira Lopes, Separation of Concerns, College of Computer Science, Northeastern University, 1995, February", NU-CCS-95-03.