James O. Coplien
Multi-Paradigm Design for C++

Coplien, James O. *Multi-paradigm design for C++* (Addison-Wesley, 1999). ISBN 0-201-82467-1

Book review and commentary by David R. Miller
January, 2001

ABSTRACT

Not for the faint of heart, this book transcends specific software design paradigms by examining what makes a good design good. The discussion is mostly at a level of abstraction beyond specific design techniques, but when Coplien descends from lofty abstract concepts to give specific examples, he does so in the context of the processes we go through every day in applying various C++ design paradigms. His abstract concepts are so general that they could apply to the process of analyzing any type of problem – not just software problems. In short, this book makes you think about how you think; it analyzes how you analyze.

Audience

This book assumes the reader is familiar with software design paradigms such as functional decomposition, objects, and patterns, and with C++ programming techniques such as object derivation, inheritance, function overloading, and templates. The reader who is familiar with modern design methodologies and with C++ language features will be best equipped to follow Coplien's discussions by matching the concrete examples in the book to the abstract concepts.

The Bottom Line

Coplien's discussions are easier to follow if you know where he is taking you. Although the author tries to organize the discussion of several abstract concepts, the organization is hard to follow. The discussion sometimes focuses on one concept, then meanders to another, only to jump to some other abstraction. When you intertwine the discussions of multiple interrelated abstract concepts with specific applications in C++, you have a book that is not always easy to follow.

The book's thesis, if we oversimplify it, is that to solve a problem in software you must analyze two domains: the problem domain and the solution domain. The problem domain is

approximated by the functional specifications of a product, but analysis of the problem domain can be generalized so that it encompasses the limits of the business market and company market goals.

The solution domain consists of the tools normally at the disposal of a C++ programmer, such as object-oriented design, traditional functional decomposition, C++ templates, application frameworks, databases, and patterns.

The two domain spaces can be characterized as N-dimensional spaces, where each dimension is a "variability" or "parameter" in the problem space, or in the tool set modeled by the solution space. Often, identification of a commonality in a domain points to a dimension of variability, or vice versa.

Domains may be decomposed into subdomains. Traditional goals of cohesion and decoupling apply when partitioning a domain.

The dimensions of the C++ solution space are relatively well-characterized by prior art. Once you identify a set of dimensions of variability in the problem space, then the final step is to transform or map the dimensions of the problem space to the dimensions of the solution space.

Let's do a Coplien trick here and switch to a concrete C++ example. In the C++ solution space, the technique of function overloading[1] adds a dimension to the solution space. That dimension represents variability of algorithm, but not of function name or semantics. C++ templates add a different dimension to the solution space, predominately uncorrelated with the preceding one. Along the dimension added by templates, there is variability in data type, but not in algorithm or behavior.[2]

Now jump to the other side of the fence to a hypothetical problem domain. Suppose that within the problem domain we find several ways in which we must sort sets of data, although there are several types of data that we must sort. That requirement, which could be considered a sub-domain of the overall problem space, lives in a domain space in which a principal dimension is variability in data type (but commonality in algorithm). Mapping dimensions from the two domain spaces, we see that the principal dimension of the sorting subdomain maps readily onto the dimension of variability that C++ templates provide.

That's an academic way of explaining why we reach into our C++ toolbox and solve the problem with a function such as template sort<T>().

Simple examples of mapping problem domain dimensions to solution domain dimensions are trivial and intuitive. It gets more complicated when a domain, especially the problem domain, must be decomposed into either a hierarchy or a structure of subdomains, and where the needs of a subdomain are characterized by multiple dimensions of variability. According to Coplien, the processes of decomposing domains and of finding the correct dimensions of the domain space are part science and part art.

---

[1] Which, by the way, is unrelated to object-oriented design or programming.

[2] Template specialization is an example of what Coplien calls "negative variability." It is an exception to what is normally a "commonality" in a paradigm.

Why is this Book Important?

This is Coplien's first major book since his classic *Advanced C++ Programming Styles and Idioms,* published in 1992.[3] In that book, Coplien (or "Cope" as he is often known) presented examples of recurring programming problems and their solutions in a C++ context. Even though the book never applied the term "pattern,"[4] his 1992 book was, at the time, one of the most rigorous and lucid presentations of what we would later know as Patterns. It took another three years for the "Gang of Four" to popularize that growing design paradigm in their pivotal book on patterns[5].

Again in 1999 Cope speaks to us in an academic voice about abstract matters that are difficult to grasp. But we should listen. If this is a repeat of his 1992 accomplishment, then perhaps this is an introduction of important things to come in the world of software analysis.[6]

Some of the concepts in the book are difficult to grasp at first. A staff writer of a journal whose name you would recognize[7] didn't bother giving the book a proper review, saying,

> *"Multi-Paradigm DESIGN for C++* is written at such a high level of abstraction that I often couldn't tell exactly what point the author was trying to make. Occasionally, I would be able to seize on something concrete…. Most of the time, though, I felt like I was in the presence of someone who'd had a glimpse of something fundamental, but whose every attempt to put it into words ended up sounding banal."

I, too, had to re-read some sections where Coplien introduced a concept, because it was only later that I realized that the concept was a necessary pillar in building the final thesis. Or I would have to re-read a section in order to unravel the intertwined discussions of orthogonal concepts.

But once in a while, at least once every few pages, Cope writes a sentence or two that is profound in its lucidity and simplicity. At those places I would stop, smile, and recognize that

---

[3] Addison-Wesley, 1992.

[4] It wasn't that Coplien was ignorant of the concept of patterns or of that term. The term "pattern" was coined by the architect Christopher Alexander in a series of books beginning with *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977. The jump from traditional architecture to software design was made in the paper by Kent Beck (Apple Computer) and Ward Cunningham (of Tektronix), "Using Pattern Languages for Object-Oriented Programs," presented at OOPSLA-87. Coplien embraced the concept, apparently understanding its essence, and applied it rigorously to C++ in his 1992 book, *Advanced C++ Programming Styles and Idioms*. The essence of patterns is so ethereal that Alexander wrote "the single central quality which makes the difference [between good and bad buildings] cannot be named," reminiscent of the opening verse of the *Tao Te Ching*. Appropriately, the name of Coplien's column on patterns in the *C++ Journal* was "The Column Without a Name."

[5] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Just as we had to wait for a Yourdon to popularize functional decomposition, or a Booch to bring us objects, or the "Gang of Four" to explain patterns, perhaps we will have to wait for another author to rephrase Coplien's thoughts for a popular audience.

[7] That would be *Dr. Dobbs Journal*, April, 1999.

this author has an intuitive grasp of issues that give us endless trouble and confusion in our profession.

Domains and Dimensions

A central concept is that a domain is an abstract "space" of an arbitrary number of dimensions. Each dimension in a domain is a characteristic or attribute that varies in the domain. Domains can be decomposed, applying traditional goals of cohesion and decoupling.

Something that varies in a domain implies that there is something that is constant, and vice versa. Consider a telephone directory service. Regardless of whether it is electronic, hardcopy, or voice activated, it has certain characteristics that are common, or constant, and other characteristics that vary. The function and algorithm are constant, but the input parameter – the name you are looking up – is variable. In a telephone directory, the variability of the input value could be modeled as one of the dimensions of the problem space.

Expanding our Horizons

Coplien reminds us several times to expand and generalize our problem domain based on a good understanding of the company's business model and the industry in which the problem is a specific instance. A company cannot sit still; it must change over time, and it must change in various ways – to keep up with the changing industry it serves, and to develop product lines according to the company's business goals. We should expect something in our problem space to change over time. On p. 36 he says,

> "The designer must understand not only the commonalities that are stable across space and over time, but also the trends in change across space and over time."

By anticipating change, we can model our problem space with the proper dimensions of variability, and map them onto solution paradigms that will best allow us to adapt and extend our solution as the requirements change. And we know that software requirements do change.

Objects

Coplien regards the object paradigm as one of many tools in the C++ tool chest. As with the other tools at our disposal, we should characterize its dimensions of variability and map its dimensions onto the dimensions of the problem space when there is a good fit. Coplien warns about the overzealous tendency to see everything as objects:

> "...one of the advantages claimed for the object paradigm, whose argument goes something like this: Presume that your programming language can express "objects." During analysis, look for "objects" as units of abstraction. When you go from analysis through design and implementation, you will find – surprise! – that you can express the abstractions of analysis in your implementation language."

In discussing how "objects" relate to multi-paradigm design, Cope explains on p. 61,

> "The common abstraction of contemporary object-oriented analysis – the class – is a second-order abstraction that unifies commonality in structure and signature. Because commonality analysis goes beyond classes, and even beyond types and responsibilities, it is a more general technique than object-oriented analysis alone."

Patterns

Coplien is not able to fit patterns into his multi-paradigm design methodology in a general way. In fact, he nearly dismisses them from the multi-paradigm solution domain, considering them abstractions of a different kind.

However, Coplien is a pattern advocate and he gives credit to the value of patterns as we would expect him to do. Except for more trivial patterns which he prefers to call "idioms," and certain patterns that implement negative variability (explained later), he considers patterns to be solutions to broader problems and places them outside the scope of the domain analysis that he proposes for C++ programming techniques. He writes,

> "...patterns can be thought of as nicknames for constructs that are part of a programming language culture but not part of the language itself..."

The author does manage to compile a table of half a dozen traditional patterns which lists their factors of "commonality" and "variability," and presents a few examples of how they might fit into his multi-paradigm methodology. It's an awkward fit, however. More work must be done in this area before patterns in general can fit into Coplien's multi-paradigm analysis, or to prove some classes of patterns are indeed qualitatively different from other design paradigms.

Loose Ends

Complexity theory

This book passes by many side trails that beg for further exploration. For example, Coplien touches on complexity theory without identifying it as such. He simplifies the concept of software complexity in two ways. First he proposes that it is some peculiarity within Western culture that causes us to organize our world into hierarchies. Second, complexity in general can be identified by an organization of multiple, interacting hierarchies, not all of which have a common root.[8] Coplien could have taken a detour to make a deeper examination of complexity, perhaps with metrics for quantifying complexity in much the same way that

---

[8] p. 157f.

Robert Martin proposed metrics for "abstractness" and "instability,"[9] or the measures of software quality proposed by Lakos based on leveling dependencies between components.[10] In the field of software QA, we have various metrics of code complexity, but not of problem domain complexity.

There exists a large body of loosely related literature that is often combined under the term "complexity sciences." Much of modern complexity research is applied to models of human social behavior or to natural systems. However, certain branches of research apply to the generalized modeling of complex systems having arbitrary spatial and temporal dimensions. Perhaps there is work to be done to find ways to apply modern complexity theory to computer science.

Lightweight methodologies

"Extreme programming[11]," and other so-called "lightweight methodologies" seem to be at odds with *Multi-Paradigm Design* in one principal area. Coplien suggests that we expand the problem domain during the analysis phase to include not only the immediate programming project, but a generalization of the problem that anticipates future changes, or future related products. Extreme Programming (XP) prefers that we not spend too much time thinking of the future, and get to work programming only what is necessary for the incremental goal that is at hand.

The battle cry of XP is "YAGNI" ("You Aren't Going To Need It"), and its motto is "do the simplest thing that could possibly work." In XP, attention is given to incremental goals toward implementing the program, and code is written in the most straightforward way to support only the current incremental goal. If the next incremental step cannot stand on the work that was done in the previous step, then some code may have to be refactored to support the requirements of the new goal. The reasoning is that you don't want to spend time adding capabilities that aren't needed in the current iteration of coding. By only coding what is required to support the current iteration, your testing is not only simplified for that iteration but can be more complete. Efficient refactoring techniques and efficient coding techniques are used in each subsequent iteration of coding to accommodate that iteration's needs, but no further.

Coplien stands on more traditional ground by emphasizing more thorough design at the beginning of a project. I don't believe Coplien wants us to use solutions that are overly complex for the sake of some beauty that can only be demonstrated by semi-formal proofs. He wants us to look at the tools that we have in our toolbox, and with knowledge and awareness pull out the one that is best suited for the present business, and the future business needs. In

---

[9] Martin, Robert Cecil. *Designing Object-Oriented C++ Applications Using the Booch Method*. (Prentice Hall, 1995). See the section titled, "Metrics," pp. 246ff.

[10] First published in Lakos, John, *Large-Scale C++ Software Design*. (Addison-Wesley, 1996). A subsequent summary with refinements appeared in several installments of the *C++ Report* in 1996, which was reprinted in *More C++ Gems*, Martin, Robert C., Ed. (Cambridge University Press, 2000), pp. 113-199.

[11] For an overview of Extreme Programming, or "XP," see Kent Beck's *eXtreme Programming Explained: Embrace Change*. (Addison-Wesley, 1999).

other words, if we have two techniques by which we can solve a problem and they are otherwise equally good solutions, why not choose the one that will make it easier to change the software in the future, or to reuse parts of the software in related products?

Change is inevitable in the world of software. XP expects change, and manages it by being nimble and adaptable. But that only works in an environment in which the cost of being nimble and adaptable is near zero. Coplien's approach, being more traditional, manages change by using programming techniques that are designed to be extensible and reusable. But that works only if the designer can predict what aspects of the program will change and to what extent. Perhaps there are lessons that can be learned and applied from both approaches.

Quality of domain dimensionality representation

Much of the first half of *Multi-Paradigm Design* is devoted to concepts of "commonality" and "variability," and how to identify those attributes in a domain. Those are important concepts because the different classes of variability form the different dimensions of the domain space. More than once, Coplien takes us to the point at which any further discussion of domain space would have to involve a mathematical treatment of dimensionality. But he always stops short of that, sometimes with the reminder that a "good" design cannot be entirely formalized because it requires experience and intuition as well as theory.

Some might like to explore Coplien's ideas of domain modeling in a more mathematically rigorous manner. For example, if classes of variability form dimensions of domain space, we might find ways to devise a "merit test" for a chosen dimension. A dimension has merit, for example, if it is uncorrelated with all other dimensions, and if it has a "useful" (meaningful) distribution of values along its axis. But first we would require more fundamental metrics. Suppose we have an axis of variability in data structure and another axis of variability in data type. How do we quantify the amount of difference between two data types? Or between two data structures?

If we can calibrate the differences of values along the dimensions of our domain space, then we can also identify cases where two or more dimensions might be more or less correlated only at limited ranges in the domain space. Consider, for example, the two dimensions of data type and data structure. Suppose we have a subdomain in which there is an abstraction that is a collection, say a vector, of stacks of different types of containers. Those abstractions might be represented in C++ as:

- vector<stack<map<T1,T2> > >
- vector<stack<set<int> > >
- vector<stack<vector<T3> > >

and so forth. This data varies only in its innermost type. It has a considerable amount of constant structure and a relatively small variation in structure. It also has a complex type with relatively minor variations. In the two-dimensional domain space of data structure vs. data type, in the subdomain of this particular item, movement along either axis implies movement along the other. Within this constrained area of the domain space there is a high degree of correlation between the dimensions. Intuition tells us that we have a set of dimensions with low merit. Perhaps the problem space could be characterized by a different set of dimensions. A more formal treatment of merit would help us choose an optimal set of dimensions.

Alternate dimensions

Coplien alludes to the fact that problem domains and solution domains can be represented by different sets of dimensions. His only definition of a "good" set of dimensions is that the set from one domain easily transforms into the set from the opposite domain, yet he refrains from giving a formal treatment of the transformational analysis. That is the proverbial "exercise left for the reader." In one chapter, Coplien gives a detailed example analysis of a problem by decomposing it into multiple decoupled subdomains with subdomains that map into multiple solution paradigms. The actual transformation, however, lacks formality. He excuses himself by asserting that "…good designers often know when to express the transformation through design patterns or idioms that are difficult to codify."[12]

Elsewhere he lists many factors that affect the ill-defined step of transformational analysis, including business policies, tool support, and sociological forces. Here we are left on our own. We have no formal suggestions from Coplien on how to find alternate models of our domains that may form an easier mapping in the event that after domain analysis, there is not a good mapping of dimensions across the domains.

Coplien partially skirts the issue by recommending that we work from both sides toward a solution. That is, we keep in mind the commonalities and variabilities of the various tools in our C++ tool chest while we analyze the problem domain, so that we are influenced – but not controlled – by the characteristics of the solution space when looking for an optimum model of the problem domain.

In our human experience, we find countless ways to see our universe as multidimensional "spaces." Often, there are alternate ways of expressing dimensions for the same space. In our physical environment, we identify location in a three-dimensional space, (x,y,z). Or we identify a point in space at a particular time using a four-dimensional "space." Instead of "this far east, this far north, and this far up," we sometimes switch to a polar system of latitude, longitude, and altitude, or altitude and azimuth. Color is defined in a three-dimensional space of (R,G,B), or (H,S,V), or in one of several other three- or four-dimensional spaces. A person's personality can be plotted as a point in four-dimensional space proposed by Jung and widely known now as the Myers-Briggs Type Indicator.[13] And so on.

How do we generalize this? How do we find the set of dimensions that best reduces complexity for our human minds? What are the qualities of a set of dimensions that makes a "space" comprehensible to us? Here again is a link to complexity theory. Assuming that "spaces" in our natural world can be represented by different dimensionalities, and assuming that it is natural for us to switch representations to match a particular context of thought, then it must be possible to formalize that process in such a way that it helps us understand how to form the most useful representation of a software problem space or solution space.

---

[12] p. 210.

[13] The extrema of the four dimensions are extroversion/introversion, intuition/sensing, thinking/feeling, and judgment/perception. Isabel Briggs-Myers and McCaulley, Mary H. *Manual: A Guide to the Development and Use of the Myers Briggs Type Indicator*. (Consulting Psychologists Press, 1985).

Negative variability

Coplien introduces a characteristic found in software problems that he calls "negative variability." Informally, a negative variability is a characteristic that is predominately constant but occasionally has exceptional values. Such characteristics are given special treatment.

The book gives us a table of C++ programming techniques that shows which ones can implement different kinds of negative variability. For example, template specializations can be used to implement exceptional values in what would otherwise be a constant characteristic. Each template specialization parameter can specify a negative variability within a characteristic that otherwise has constant name and behavior.

Coplien does not give us a cogent explanation for treating negative variability separately.[14] It would be possible to generalize the multi-paradigm design methodology so that it includes dimensions of variability in which the variation has arbitrary or unusual distributions along the dimension. By doing so, we could extend our merit tests for dimensionality (proposed above) to include the distribution along each axis of the domain. The transformational analysis to map one domain's dimensions onto the other domain's dimensions could consider the distribution along each axis to find the optimum mapping

History

The concept of domain analysis seems to have taken shape in the late 1970's.[15] In the 1990's, the term "multi-paradigm design" began to appear in literature[16,17] At about the same time, David M. Weiss and others seems to have developed the ideas of commonality and variability analysis as part of domain analysis. Coplien traces the roots of commonality analysis as far

---

[14] Coplien treats this more fully in his Ph.D. thesis. He writes, "negative variability can be used to formally characterize function argument defaulting, template specialization, certain properties of preprocessor constructs, and even most common design patterns. This is a powerful unifying notion of design that can be used to advantage in a design method and in its supporting notations and processes." He also cites common conventions ("inheritance with cancellation"). It seems that this is *apropos* to Coplien's thesis in that negative variability is a kind of variability, or a class of domain dimensionality. He admits that negative variability makes sense as a characteristic of domain analysis only when the negative variability is relatively small compared to the related commonality. A sufficiently large negative variability can be inverted and viewed as a positive variability. He prefers to find and use positive variabilities when possible. Again from his doctoral thesis: "Positive variability is the preferred mode of design under multi-paradigm design because it most often leads to design support by additional formalisms...." *Multi-Paradigm Design*. (Vrije Universiteit Brussel, Faculteit Wetenschappen – Departement Informatica, July 8, 2000).

[15] Neighbors, J. M. "Software Construction Using Components." (University of California, Irvine, 1980).

[16] Barton, John, and Lee Nackman. *Scientific and Engineering C++*. (Addison-Wesley, 1994).

[17] Budd, Timothy. *Multi-Paradigm Programming in Leda*. (Addison-Wesley, 1995).

back as 1976[18] or even earlier.[19] Coplien experimented with some ideas about multi-paradigm design in his column about patterns that appeared in the *C++ Journal* in the 90's[20], and published a paper on the subject in 1994.[21]

Summary

Multi-Paradigm Design for C++ forces you to think about the way you look at problems. It makes you think about the nature of complexity. It does not give you cookbook techniques for solving programming problems; you still have to figure out the best way to solve those yourself. If you enjoy following the latest thoughts in design methodology, then you may enjoy expanding your thinking with the ideas discussed in this book. It can help by suggesting ways to formalize a model of the way we solve problems. Just don't lose sight of the art and intuition involved in software design. As Coplien says on p. 165, "Multi-paradigm design becomes an audit for that intuition and provides techniques and vocabulary to regularize the design."

---

[18] Parnas, D. L. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering,* March 1976.

[19] In his Ph.D. thesis (*ibid*.) Coplien adds a reference to a publication by Dijkstra in 1968 in which a software analysis of "families" was based on variabilities in design.

[20] The column was aptly named, "The Column Without a Name." See footnote 4.

[21] "Multiparadigm design for C++". *Proceedings of the 1994 SIGS OOP/C++ World*. (SIGS Publications, 1994).

Multi-Paradigm Design for C++ book. Read reviews from world's largest community for readers. Coplien provides invaluable advice on design ing complex, mu... Coplien provides invaluable advice on design ing complex, multiparadigm programs. He describes each parad igm and explains how it solves design problems at hand and h ow it applies to C++. ' Get A Copy. Kindle Store. Multi-paradigm design is a metaparadigm: it enables to select the appropriate paradigm among those supported by a programming language for a feature being modeled in a process called transformational analysis. A paradigm model is a basis for multi-paradigm design. Feature modeling appears to be appropriate to represent a paradigm model. Such a model is proposed here for AspectJ language upon the confrontation of multi-paradigm design and feature modeling. Subsequently, the new transformational analysis is discussed. Discover the world's research.