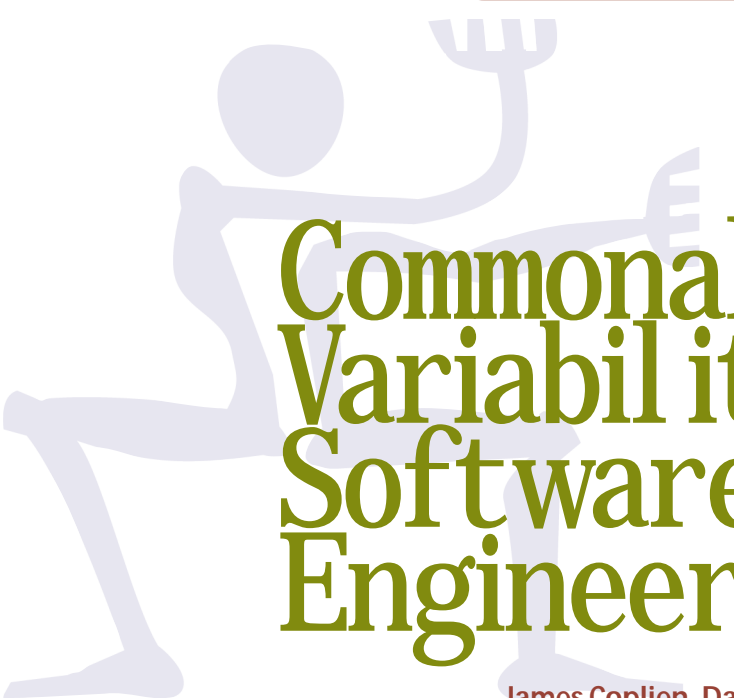




## Experience Report

- Software engineers are under tremendous pressure to develop new system versions in less time. The authors show the benefits of explicitly identifying the common and variable aspects of the different versions of a system.



# Commonality and Variability in Software Engineering

James Coplien, Daniel Hoffman, and David Weiss, Bell Labs

**I**ncreasingly, software engineers spend their time creating *software families* consisting of similar systems with many variations. While developers are pressed to build these families, they have no effective means for doing so. They are asked to create and reuse libraries of components but find those libraries costly to build and of limited value. They search for the right decomposition of their software into modules or classes, but have limited guidance in finding those decompositions, especially in the face of constraints on performance, reliability, and ease of use.

Scope, commonality, and variability (SCV) analysis gives software engineers a systematic way of thinking about and identifying the product family they are creating. Among other things, it helps developers

- ◆ create a design that contributes to reuse and ease of change,
- ◆ predict how a design might fail or succeed as it evolves, and
- ◆ identify opportunities for automating the creation of family members.



For example, when designers know that a product will have to control and monitor a variety of similar devices, they can establish requirements for device variabilities. In this case, designers might use standardized, abstract interfaces to each device, and encapsulate the device control code into separate modules that users can access through the interface.<sup>1</sup> Here, the commonality is the interface, and the variability is the device control code. The cost and

**We have used the FAST approach in over 25 domains and have seen immediate payoff.**

speed with which a new device can be incorporated into the product line will depend on how well it conforms to the abstract interface. Astute product-line designers may try to influence industry standards so that device suppliers will build products that easily conform to the company's abstract interfaces, thereby giving the company a competitive advantage.

The themes of commonality and variability implicitly pervade many aspects of software development today, as we describe in the boxed text, "SCV Analysis: Origins and Related Work" on p. 39. SCV analysis synthesizes these ideas. Here we describe SCV analysis precisely, and discuss the benefits and challenges of its application. We also discuss our Family-Oriented Abstraction, Specification, and Translation (FAST) approach, which uses SCV analysis to identify, formalize, and document commonalities and variabilities. At Lucent Technologies, we have used the FAST approach in more than 25 domains and have seen immediate payoff, both in overall productivity and changes in the way developers think about their design problems.

## A Model for SCV Analysis

To achieve both precision and abstractness, we specify commonality and variability in terms of sets. A *commonality* is an assumption held uniformly across a given set of objects (S). Frequently, such assumptions are attributes with the same values for all elements of S. Conversely, a *variability* is an assumption true of only some elements of S, or an attribute with different values for at least two elements of S. We can illustrate these concepts in three simple examples.

### Model examples

For our first example, let S be the set of all circles, triangles, and squares. We can assume that every element of S is planar and has an area, and thus both "planarity" and "has an area" are S commonalities. Because triangles and squares differ in number of sides and how their areas are computed, "number of sides" and "formula for area" are variabilities. Our second example concerns the familiar "isa" relationship, which gives rise to a hierarchical structure in which commonality increases with specialization. For example, let  $S_0$  be the set of all four-sided polygons.

All elements of  $S_0$  are planar and have four sides. However, two elements of  $S_0$  may differ in the lengths of their sides and size of their interior angles, and may be convex or concave. Let  $S_1$  be the set of all rectangles. Because a rectangle is a four-sided polygon,  $S_1$  inherits the commonalities of  $S_0$  and has others as well, including "concave," "opposite sides are equal in length," and "interior angles are all right angles." Then, let  $S_2$  be the set of all squares. Because a square is a rectangle,  $S_2$  inherits the commonalities of  $S_0$  and  $S_1$ , and has some new ones including "all four sides are of equal length." As this example shows, changing S can significantly affect the commonalities and variabilities. Generally, reducing the size of S increases the commonalities and reduces the variabilities.

Our third example uses factoring in algebra. Consider the following set S of two formulas:

$$(x - y)^2 \quad (1)$$

$$(x^2 + y)(x - y) \quad (2)$$

The common factor in the formulas is  $x - y$ . The remainders— $x - y$  for (1) and  $x^2 + y$  for (2)—are variabilities. If we introduce even a small change, such as replacing the minus with a plus in equation (1), then  $x - y$  is no longer a common factor. Also, if (1) and (2) are rewritten as  $x^3 - x^2y + xy - y$  and  $x(x + y) + y + (y - 3x)$ , they are algebraically equivalent, but the commonality and variability are much harder to identify.

### Benefits of systematic analysis

Most efforts in designing C and V into a program are directed toward deciding what decisions can be changed at runtime and what cannot. For example, some operating systems let a hardware device—such as a printer or a network interface—be added at runtime and some do not. Many of these deci-

## SCV ANALYSIS: ORIGINS AND RELATED WORK

*The art of progress is to preserve order amid change, and to preserve change amid order.*  
—Alfred North Whitehead

SCV analysis has its roots in work by David Parnas, Edsger Dijkstra, and Harlan Mills.

### Information Hiding

Parnas' information hiding principle<sup>1</sup> encodes commonality as a module's interface and variability as a module's secret. In some sense, developers used information hiding long before Parnas published his seminal paper. For example, device drivers were commonly used well before 1972, providing abstract interfaces to the underlying hardware. Parnas identified and made explicit a common aspect of good software development practice. He provided the basis for teaching information hiding, for reviewing the resulting designs, and for applying the technique to a variety of modules.

### Program Families

Early work by Dijkstra recognized a key relationship between design decisions and program families;<sup>2</sup> each alternative corresponds to a new family member.

Parnas identified the fundamental motivation for creating program families: "We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members."<sup>3</sup>

### Program Factoring

Factoring plays an important role in SCV analysis. Mills viewed programs as algebraic expressions and showed how operations such as factoring and substitution could be usefully applied to source code.<sup>4</sup> Mills took the algebraic viewpoint literally, often reducing programs to their prime factors: irreducible constructs from structured programming.

### Domain Engineering

Concepts of commonality and variability are appearing widely in domain engineering. Domain engineering in hardware product lines was in practical use at least 30 years ago. The IBM 360 hardware family is a classic example, providing strict upward compatibility for machine code across a product line containing six models and spanning a performance range of 50:1.

Software domain engineering is less mature, but is being

actively pursued. In these efforts, commonality and variability appear repeatedly, but are often implicit.

There are several examples of this. Hans Schmid<sup>5</sup> presents the design of a family of software controllers for automated machining. The design focuses on "hot spots" (variabilities) and makes use of design patterns to encapsulate them. W. Lam<sup>6</sup> describes a process for variability analysis based on variability templates and a variability hierarchy. This process was used to develop an initial architecture for a family of software controllers for jet engines; abstract interfaces encapsulated the variabilities. Jacques Meekel<sup>7</sup> describes the design of the FLEX kernel, a real-time operating system for a wireless-pager product family. Variabilities were identified and classified as feature, device, and performance variabilities, and techniques were developed to handle each type.

Work in domain-specific architectures<sup>8</sup> and software development processes based on domain engineering<sup>9,10</sup> explicitly use commonality and variability analysis.

## REFERENCES

1. D.L. Parnas, "On the Criteria to be Used in Decomposing a System into Modules," *Comm. ACM*, Dec. 1972, ACM Press, New York, pp. 1053-1058.
2. E.W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds., Academic Press, London, 1972.
3. D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.*, Mar. 1976, pp. 1-9.
4. H. Mills et al., *Structured Programming: Theory and Practice*, Addison Wesley Longman, Reading, Mass., 1979.
5. H.A. Schmid, "Creating Applications from Components: A Manufacturing Framework Design," *IEEE Software*, Nov. 1996, pp. 67-75.
6. W. Lam, "Creating Reusable Architectures: Initial Experience Report," *Software Eng. Notes*, Vol. 22, No. 4, July 1997, pp. 39-43.
7. J. Meekel, T.B. Horton, and C. Mellone, "Architecting for Domain Variability," *Second Int'l Workshop on Development and Evolution of Software Architectures for Product Families*, LNCS 1429, Springer Verlag, Berlin, 1988, pp. 205-213.
8. D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Eng. and Methodology*, ACM Press, New York, Oct. 1992, pp. 355-398.
9. W. Tracz, "LILEANNA: A Parameterized Programming Language," *Selected Papers from the 2nd Int'l Workshop on Software Reliability*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1993, pp. 66-78.
10. G.J. Campbell et al., "Reuse in Command and Control Systems," *IEEE Software*, Sept. 1994, pp. 70-79.

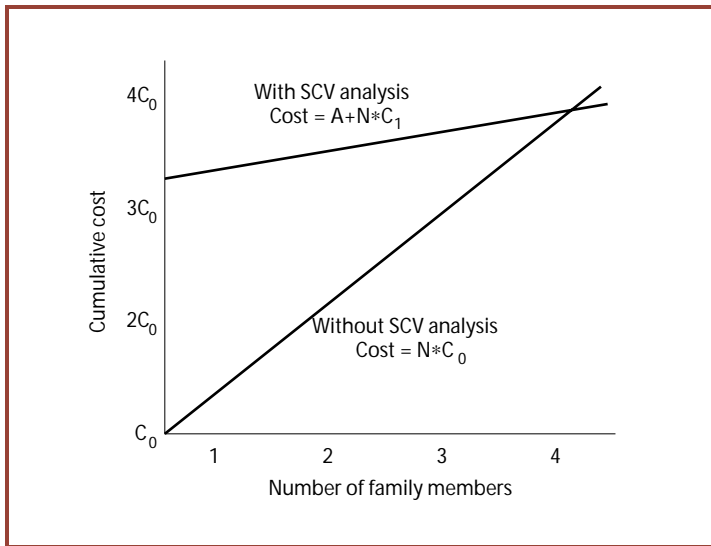


Figure 1. Automation tradeoffs with and without SCV analysis.

sions are made when the code is written and are particularly difficult to discover and change after the code is in use. As a result, the inherent power in software also becomes a liability, making the ability to define and control C and V both a blessing and a curse. We suggest that both software developers and software users would benefit if SCV analysis were applied systematically throughout the software development life cycle.

The benefits of this systematic approach include opportunities for rapid new development due to reuse, as well as decreased development costs and the rapid creation of new family members stemming from automation, reuse, and ease of change (through encapsulation of variabilities).

## Making a Case for Automation

When commonalities are invariant and variabilities precisely defined, developers create opportunities for high-payoff automation. The benefit of automation is well known in manufacturing industries, where production lines for items such as automobiles, television sets, and personal computers rely on a family architecture to improve production efficiency.

To automate assembly lines, designers create family architectures with known and precisely determined variability. Regardless of whether the line is fully or partially automated, certain product as-

pects, such as the chassis, must remain the same for all products, and variability must be confined to a predetermined range of values (that is, the variabilities are parameterized).

Manufacturers invest in tool sets that repeat the same task sequence with predetermined variability introduced at specific steps. For software, such tools might be generators that take as input a parameterization of variabilities (such as a table or other specification), and then generate the desired software.

But automation involves tradeoffs. Figure 1 shows a simple economic model that compares the costs of automation with and without SCV analysis. Let's suppose we are producing members of a family for which we have analyzed the commonality and variability, and established the variability parameters. Suppose that, initially, the average cost to develop a single family member manually is  $C_0$ , and that the cost of completing this development with automation is  $C_1$ . Suppose further that the cost of the analysis and automation is  $A$ . The cost for developing  $N$  family members is then  $N * C_0$  before SCV analysis and  $A + N * C_1$  after. To make a business case for the automation effort,  $C_1$  must be smaller (typically much smaller) than  $C_0$ , and  $N$  must be large enough to recover the upfront investment,  $A$ . In Figure 1, the break-even point is at  $N = 4$ .

Naturally, in practice, the situation is much more complex than Figure 1 suggests. Among other considerations are time to market and the change in production costs over time. Nonetheless, Figure 1 illustrates a fundamental tradeoff that is often ignored in software automation efforts.

## Applying SCV Analysis

We use five main steps in SCV analysis.

1. Establish the scope: the collection of objects under consideration.
2. Identify the commonalities and variabilities.
3. Bound the variabilities by placing specific limits—such as maximum values—on each variability.
4. Exploit the commonalities.
5. Accommodate the variabilities.

SCV analysis can be applied to narrow and broad programming tasks. To create specific programs, programmers use SCV analysis to craft individual lines of code that permit certain variability and take advantage of commonality. In broader tasks—such as domain engineering—programmers use SCV analysis to create systems that will be long-lived and exist

in many variations; that is, they create software product lines.

SCV thinking has always been a driving force in language design. Many language features are focused primarily on exploiting commonality and accommodating variability. Language mechanisms are designed to support a particular paradigm or group of paradigms such as procedural, functional, object-based, and object-oriented. Each of these paradigms reflects a recurring pairing of commonalities and variabilities. We next consider three mechanisms: procedures, inheritance, and class templates.

### Procedures

For exploiting commonality, *procedures* are a powerful mechanism, and probably the one most widely used by programmers today. In SCV analysis, procedures break down as follows:

- ◆ S: a collection of similar code fragments, each to be replaced by a call to some new function F.
- ◆ C: the code common to all fragments in S.
- ◆ V: the “uncommon” code in S. Variabilities are handled by, for example, parameters to F or custom code before or after each call to F.

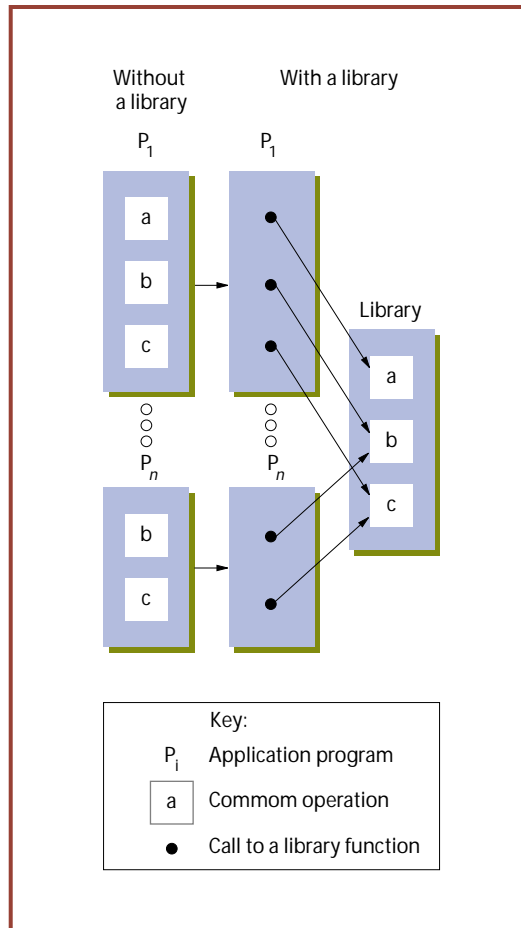
### Inheritance

*Inheritance* provides abstraction similar to procedures, but enlarges the scope to groups of procedures and their associated data structure. In SCV analysis, inheritance breaks down as follows:

- ◆ S: a collection of classes.
- ◆ C: the code common to all classes in S; this code is placed in the base class.
- ◆ V: the “uncommon” code in S; this code is placed in the subclasses.

Simply used, inheritance is only a factoring mechanism to capture common code. The more powerful use of inheritance is for object-oriented programming, where the base class captures common type semantics that are inherited by derived classes. These semantics are expressed as member functions and their parameter and return types called a *signature*. Derived classes each supply their own code implementation. Here the SCV analysis is:

- ◆ S: a collection of member functions organized into a class hierarchy.
- ◆ C: the signature common to all classes in S; this signature is supported by all classes, either expressly, or by inheriting a base class function.
- ◆ V: the “uncommon” code in S; this code is placed in the subclasses.



**Figure 2.** A simplified before and after scenario for a set of programs  $P_1, P_2, \dots, P_n$ . Programs are on the left; common operations are a, b, and c.

### Parametric polymorphism

Inheritance is most commonly used with inclusion polymorphism, a language feature that automatically calls out the variabilities at runtime. Parametric polymorphism, including class and function templates, provides what is possibly the most direct and straightforward representation of commonalities and variabilities. Code and data structure can be written in terms of explicit parameters of variation. For example, templates can represent a “stack of T of maximum size N,” where T might be `int`, `float`, or some user-defined type, and N might be a cardinal expression. With respect to the class templates, which are the most common form of parametric polymorphism, the SCV analysis is:

- ◆ S: a set of classes implementing the same operations but on different types.
- ◆ C: the template code; identical across classes except for one or more types.
- ◆ V: the template actual parameters, that is, the different types.



## LESSONS FROM THE LIBRARIES

Two modern reuse libraries teach important lessons about SCV analysis as it relates to designers' skill and the impact of marketing and political issues.

### STL

The Standard Template Library, now an ANSI standard, provides C++ programmers a variety of operations on sets and sequences. STL supplies seven container classes and approximately 80 algorithms, and pays considerable attention to performance. Iterators are used to avoid the "algorithm explosion problem" common to other libraries, which require one set of algorithms for each container. In STL, each container provides a standardized iterator interface, based on overloaded versions of the C++ operators used to access conventional arrays. Because algorithms access the containers only through this interface, a single set of algorithms accesses all containers and arrays. Further, STL algorithms such as `copy` can "mix and match" copying. They can, for example, copy from one set to another or from a vector to a set. Finally, if STL users develop a class that provides the standard iterator interface, then all STL algorithms can be applied to the new class.

The STL design teaches an important lesson in SCV analysis: sometimes a skillful designer can exploit commonality that is not even apparent to others and can accommodate variability in ways others never considered. This skill can provide enormous leverage.

### Java AWT

The Java Abstract Window Toolkit is an ambitious attempt at cross-platform portability, aiming to supply modern GUI features on platforms such as Macintosh, Unix, and Windows. In Java AWT, the scope consists of the features offered by one or more of the popular GUI libraries, but factors out features offered by all (or most) of those libraries.

The AWT library does not offer remarkable features; indeed, many existing libraries offer more. Nonetheless, AWT does achieve impressive portability, though this is already being threatened by variability. For example, Microsoft has developed their own Java GUI library to exploit features of the Windows APIs not available through the AWT. In SCV terms, Microsoft has reduced the library's scope to features offered by the Windows APIs, preferring functionality to portability.

The AWT design teaches two important lessons about SCV. First, a large experience base is typically required for success. The Java AWT was designed with detailed knowledge of many preceding libraries. Second, SCV efforts are sensitive not just to technical concerns, but to marketing and political concerns as well.

### Reuse libraries

Software libraries have been in widespread use since the 1950s, when developers observed that certain operations were repeated across programs and could be reused to increase productivity and reliability. In SCV analysis, reuse libraries break down as follows:

- ◆ S is the set of all operations in the programs

being considered;

- ◆ C is the operations appearing in all (or many) of the programs; and
- ◆ V is the remaining code in the programs.

Figure 2 illustrates this before/after scenario graphically for a set of programs  $P_1, P_2, \dots, P_n$ . The programs are shown on the left, with common operations a, b, and c. The impact of a reuse library is shown on the right, with the code for each operation replaced by a call to a library function. In practice, however, the situation is not so simple. Typically, shared operations, such as b in  $P_1$  and  $P_n$ , do not appear identically in each program. The variations between each instance of an operation must be accommodated, for example by adding parameters to the library function or by placing code in each program before or after the function call. Many choices are possible, making the design space large.

Although a big payoff is possible by factoring out common code into libraries, achieving this payoff requires skill, experience, and considerable trial and error. There are numerous examples of successful software libraries, including early I/O libraries, math libraries, and the C standard libraries. Two more recent examples, the C++ Standard Template Library and the Java Abstract Window Toolkit (AWT), are described in the boxed text, "Lessons from the Libraries," on this page.

### Software product lines

To create successful systems, companies must predict how the marketplace will evolve, including future changes in customer requirements, competitor offerings, and the underlying technology. A company's ability to identify product-line variability and commonality affects its marketplace share and research and development costs. In today's competitive marketplace, failing to consider these factors can lead to company failure. In SCV analysis, software product lines break down as follows:

- ◆ S is the product line, driven by the market and constrained by the available technology;
- ◆ C is the characteristics common to all products in the line; and
- ◆ V is the variations among those products.

Using our earlier example, we apply SCV analysis to similar hardware devices used in a product line:

- ◆ S is the set of devices used in the product line;
- ◆ C is the abstract interface used to access the device; and
- ◆ V is the implementation required for a particular device.

Device variation is a common example of SCV analysis as applied early in the product development cycle—in this case, during requirements analysis. This early application of SCV is directly traceable to later decisions in design and implementation; without it, such decisions can go unmade until the implementation stage, when the people making decisions are less familiar with their implications and impact. The more accurately designers can predict SCV, and the earlier they can do so, the greater the advantage to the product line.

Designing software product lines<sup>2</sup> requires that designers think about SCV early in the development cycle. Such thinking frequently points the way to large-scale reuse throughout the product line and to automated generation of family members.

## The FAST Approach

SCV analysis can be incorporated into a software development process in various ways. Our FAST approach to domain engineering uses SCV analysis to identify, formalize, and document commonalities and variabilities.<sup>3</sup> FAST uses the results of this process to first create a language for specifying domain members, and then generate members from these specifications.

The SCV analysis we use in the FAST process focuses on program families and produces a *commonality analysis document*, which is a record of the family's terminology, commonalities, and variabilities, and the key issues that arose during the analysis. Figure 3 shows the structure of a FAST commonality analysis document.

### Terminology

Most software development methodologies now suggest that developers equip themselves with a dictionary of standard terms to make communication among developers easier and more precise. Also, because the terms are standard, they represent ideas that are common to the development and are therefore a fruitful source of abstractions. We thus made developing a dictionary of terms part of the FAST SCV analysis process.

### Commonalities and variabilities

Identifying common aspects of the family is a central aspect of SCV analysis in FAST. Accordingly, the analysis contains a list of assumptions that are true for all family members. Variabilities define the

**Introduction.** Describes the purpose of the analysis, which typically includes

1. Defining the requirements of a particular family
2. Providing the basis for capabilities such as
  - ◆ family member specification,
  - ◆ family member code and documentation generation, and
  - ◆ family member composition (based on a components set designed for use in many family members).

**Overview.** Briefly describes the domain and its relationship to other domains.

**Dictionary of Terms.** Provides a standard set of key technical terms used in discussions about and descriptions of the domain.

**Commonalities.** Provides a structured list of assumptions that are true for all family members.

**Variabilities.** Provides a structured list of assumptions about how family members can vary.

**Parameters of Variation.** Specifies the value range and binding time for each variability.

**Issues.** Provides a record of the alternatives considered for key issues that arose in analysis of the family.

**Figure 3.** The FAST commonality analysis document describes the purpose of the analysis and highlights key issues that arose while it was underway.

family's scope by predicting which decisions about family members are likely to change over the family's lifetime. A commonality analysis document thus contains a list of variabilities, with a range of values for each. These value ranges are the *parameters of variation*.

Fixing a value for a parameter of variation specifies a family subset. For example, a variability for the shapes family is that different shapes have different numbers of sides. Selecting "four" for the number of sides rules out circles and triangles as subfamily members and admits squares.

In addition to specifying the range of values for each variability, the analysis also specifies the time at which the value is fixed; that is, it specifies the binding time for the decision. Typical binding times include runtime, system-build time, and system-specification time. Fixing a decision early can lead to more efficient software, in terms of both size and performance. For example, if we want to generate a calculator for the shapes family that computes area,



perimeter, and other metrics for family members, then we might be able to fix various program parameters, such as the formula for calculating the perimeter, at compile time.

### Patterns and SCV

Software designers and programmers are increasingly using patterns to document fundamental structural design models that lack strong semantics of commonality and variation. Design pattern literature<sup>4</sup> is the most popular offshoot of patterns in contemporary software design and often reflects such considerations. For example, the Adapter pattern<sup>4</sup> makes it possible to hide isolated differences in the way a service is offered. Adapter hides differences that violate the predominate commonality in a class in-

**Patterns complement the principles of commonality and variation that underlie such methods as structured design and objects.**

terface. These *negative variabilities*<sup>5</sup> violate commonality, rather than complementing commonality as variabilities should. Though such exceptions aren't the rule, they are common enough to capture as standardized design approaches. As such, they complement stock methods (such as structured design and objects) well. By extension, patterns complement the principles of commonality and variation that underlie such methods.

### FAST versus other approaches

The FAST approach draws on the SCV thinking prevalent in many approaches to domain engineering, but differs in four important ways. First, FAST's SCV analysis is explicit and pervasive in the process and the resulting work products. We set aside time for performing the analysis early in the development process.

Second, we have standardized both the analysis process and the resulting document. As a result, designers can focus their attention on designing the family.

Third, FAST analysis focuses on bounding variability. Both the parameter values and their binding time can significantly affect the feasibility of automatically generating family members; they can also affect their runtime performance. Because these decisions are influential, they must be precisely recorded. This gives all stakeholders an opportunity

to review decisions, and provides designers and implementers with a written record.

Finally, at Lucent, we have extensive experience with SCV analysis and have applied it using the FAST approach in more than 25 domains. The FAST approach is polished and proven. Experienced moderators lead FAST sessions and our analysis teams include the developers responsible for the software in the given domain.

## Lessons Learned

Based on our experience, we have formulated the following basic principles for SCV analysis.

- ◆ *Make S, C, and V explicit.* SCV analysis focuses on a set of decisions with influential consequences. With or without an explicit SCV analysis, the choices will be made; the analysis provides a useful framework for discussing the decisions and their consequences.

- ◆ *Choose S to balance generation costs and family size.* Establishing the scope can be extremely influential and misunderstandings can be damaging. If S is too big, the family has many members but few commonalities, and automatic generation opportunities are limited. If S is too small, there are plenty of commonalities but not enough family members to make automatic generation pay.

- ◆ *Search for C to maximize reuse.* Commonalities are the main source of reuse and thus SCV analysis provides "the cure for the common code." Common code can be written once and shared by all family members. Much of the progress in programming-language design and software methodology has been based on techniques for identifying and exploiting commonality.

- ◆ *Bound the variabilities to minimize production costs.* Variabilities are the main focus of automated software generation. Bounding the variabilities is central to successful automation. At times, brutal restrictions are necessary. The classic example of this is the Ford Motor Company's "any color you like, as long as it is black" approach to automobile paint color in the early days of assembly-line manufacture. The market has since demanded increased variability and the technology has improved to permit it.

- ◆ *View programs as mathematical expressions; use factoring to find commonality.* SCV analysis has





a lot in common with factoring in conventional algebra. A good factorization can be a big win in simplicity, insight, and ease of computation, but it takes effort, skill, and sometimes luck to achieve. For example, it is hard to predict when or if success will be achieved, and factorizations are sensitive to change. However, S, C, and V are often negotiable. This is good in that a developer can sometimes spot a small change to permit a factorization that saves significant effort. On the other hand, customers can mandate a small change that invalidates a successful factorization—and all the work that went into finding and implementing it. Such negotiations can vastly increase design-space size and design difficulty.

◆ *Skill and experience count.* Skillful designers can find and exploit more commonality and accommodate more variability than their less-skilled counterparts. With the design space so large, considerable invention is required. Further, a large experience base is required to successfully identify C and V.

◆ *Periodically revisit each SCV analysis.* While it is important to make S, C, and V explicit, it is equally important to realize that they will change over time (Ford now offers numerous automobile colors).

**S**oftware engineers must produce systems rapidly, but also be sure they are carefully engineered. This is the software engineer's dilemma. Frequently the dilemma arises from past success: a product that is successful in the marketplace blossoms into many versions due to continuing demands for new features, implementation on new platforms, and improvements in the user interface. It is easy for a software development organization to find itself maintaining many versions of the same product with large amounts of similar code created as an afterthought to satisfy new requirements.<sup>6</sup>

SCV analysis gives software engineers a technique that helps them ameliorate the problems of such success. It gives them a systematic way of thinking about and identifying the family that they are creating. It helps to analyze the economics of creating a family. It helps to inform and illuminate a design to clarify structures and patterns that contribute to ease of change and reuse, suggesting ways that a design may fail or succeed as it evolves. It helps identify opportunities for automation to support the creation of family members.

Using SCV analysis at Lucent, we have come to expect decreases in development interval of a factor of 3 to 5, particularly when we apply the FAST approach in a particular domain. The key to success with SCV analysis is that it changes the way software engineers think about software development. Once they are trained to think about SCV in a systematic

way, their viewpoint about software development and its tradeoffs are forever changed. ♦

## REFERENCES

1. D.L. Parnas et al., "The Modular Structure Of Complex Systems," *IEEE Trans. Software Eng.*, Mar. 1985, pp. 408-417.
2. L. Bass et al, *Software Architecture in Practice*, Addison Wesley Longman, Reading, Mass., 1998.
3. N.L. Gupta et al., "Auditdraw: Generating Audits the FAST Way," *Proc. IEEE Int'l Symp. Requirements Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 188-197.
4. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Reading, Mass., 1995.
5. J. Coplien, *Multi-Paradigm Design for C++*, Addison Wesley Longman, Reading, Mass., 1998 (to appear).
6. D. Dikel et al., "Applying Software Product-Line Architecture," *Computer*, Aug. 1997, pp. 49-55.

## About the Authors



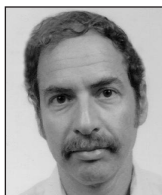
**James O. Coplien** is a principal investigator in the Software Production Research department at Bell Laboratories in Naperville, Illinois. He conducts research in software design patterns, empirical organizational modeling, multiparadigm design, and the object paradigm.

Coplien received a BS in electrical and computer engineering and an MS in computer science from the University of Wisconsin at Madison.



**Daniel Hoffman** is an associate professor of computer science at the University of Victoria, British Columbia, and is currently on sabbatical leave at Bell Laboratories. His research focuses on the industrial application of software documentation, inspection, and testing.

Hoffman received a BA in mathematics from the State University of New York, Binghamton, and MS and PhD degrees in computer science from the University of North Carolina, Chapel Hill.



**David M. Weiss** is currently head of the Software Production Research Department at Lucent Technologies / Bell Laboratories, where he conducts research into methods and processes for improving software production efficiency. Previously, he was director of the Reuse and Measurement

Department of the Software Productivity Consortium. He has also worked in the US Office of Technology Assessment, where he co-authored a technology assessment of the Strategic Defense Initiative. He has also been a visiting scholar at the Wang Institute and a researcher at the Naval Research Laboratory. His research interests are in software engineering, particularly in software development methodologies, software design, software measurement, and, more recently, domain engineering.

Weiss received a BS in mathematics from Union College and an MS and PhD in computer science from the University of Maryland.

Address questions about this article to Coplien, Hoffman, and Weiss at Bell Laboratories, Lucent Technologies, 263 Shuman Blvd., Naperville, IL 60566; {cope,dhoffman}@research.bell-labs.com; Weiss at weiss@lucent.com.