Real-time complexity metrics for Smalltalk methods

by S. L. Burbeck

This paper presents the rationale for ubiquitous and immediate metric feedback on the complexity of Smalltalk methods whenever they are viewed or changed. It also presents seven metrics for Smalltalk methods that can be quickly and easily determined from the code and are suitable for real-time feedback. For each of these metrics, there is a description, an explanation of how it is determined and why it is related to complexity, and suggestions for how to improve code that receives a poor rating for the metric.

Complexity is the relentless enemy of software engineering. Civil engineers have a saying that "rust never sleeps." The comparable maxim in software engineering ought to be that "complexity never sleeps." It frustrates our efforts to express clear requirements, to create clean designs, and to produce understandable, maintainable implementations. Certainly some, perhaps much, of this complexity is inherent in the application domain. Additional complexity can be attributed to limitations in the tools and languages with which we implement software. Yet all too much complexity is inadvertently introduced by the very software professionals whose job it is to minimize it.

Rarely does a professional designer or programmer deliberately add gratuitous complexity to a system. Complexity creeps into software systems unbidden—a little here, a little there—as a result of poorly informed or hasty choices, small misunderstandings, poor communications, and subtle failures of foresight. Much of this unbidden complexity would be avoided if we could spot it as it creeps into our systems.

The traditional policy of obtaining complexity metrics at major milestones in a software development project (or worse yet, at the end of the project) does not help us to detect creeping complexity. The only practical choice to be made at major milestones is to accept or reject the work done thus far. Rework to reduce complexity usually seems too difficult or expensive because the system embodies a bewildering set of interlocking assumptions and trade-offs. Once complexity has gotten out of control, it takes control!

What we need are *real-time metrics:* metrics that are computed afresh and presented without noticeable delay each time a change is made. Real-time metrics become the complexity equivalent of a smoke alarm—a complexity detection process that is ever vigilant and that warns us at the first sign of excess complexity, rather than one that notifies us after the fact that the software is hopelessly complex.

Real-time metrics are one example of the kind of constant software quality management proposed by Adams and Burbeck in 1992.[1,2] They argued that short cycle-time quality feedback, which has proven its effectiveness in manufacturing, is equally useful in software development. They also point out that quality feedback is not just for managers. Programmers need feedback on software quality as much as

0018-8670/96/$5.00 © 1996 IBM

do managers, and, unlike managers, programmers can make immediate direct use of such feedback. After all, minimizing complexity should be the responsibility of those who best understand the true cost of complexity: the professional software developers. To discharge that responsibility, developers need feedback about the effects of their work. Metric feedback is most useful at the time each addition or modification to the system is made because the alternatives and trade-offs involved in the change are fresh in the mind of the developer. Immediate feedback allows developers to continuously and iteratively manage the complexity of a system as it is built. Experience with such a system shows that professional developers welcome such feedback and use it to improve their work.

Issues raised by real-time metric feedback do not arise or are less important when metrics are divorced from the minute-by-minute business of software development. Such issues include:

- Which metrics should be measured? As with all forms of performance measurement, you must choose metrics carefully.[3] As the saying goes: "Be careful what you wish for. You might get it." If we wish developers to pay attention to real-time metrics as they work, it behooves us to ensure that our metrics measure complexity in a meaningful way. This requires both a theoretical understanding of complexity measurement and some empirical study of the reasonableness of the metrics in real-world applications.
- Which metrics can be measured? Software complexity takes many forms, not all of which can be measured meaningfully. Of those measurements that are meaningful, not all can be computed easily enough or quickly enough for use in real-time metrics.
- How should metric feedback be presented to the developers? Tabular reports such as those available from traditional metrics tools would quickly become annoying if they appeared every time a change was made to the system. Ubiquitous real-time feedback should be unobtrusive yet informative. The volume of feedback information should be very small when complexity is within acceptable limits, but more detail should be available when it is needed or desired.
- How are developers expected to reduce complexity when metrics indicate the need? Traditional metrics reports invite a leisurely assessment of the options for reducing complexity. In contrast, the typically rapid pace of software development, and hence of the real-time metric feedback, encourages the developer to immediately rectify a problem. Experienced developers will usually know how to do so, but inexperienced developers could easily be at a loss. Thus, it is important to provide immediately available suggestions for how improvements can be made.

This paper examines these issues in the context of Smalltalk, one of the two most popular object-oriented programming languages. The second and third sections discuss what we mean by complexity and how it is manifested in Smalltalk systems. The fourth section reviews the key issues that affect our ability to measure complexity in Smalltalk systems. The fifth section discusses the reasons for and the effectiveness of real-time feedback of metric information. The sixth section presents seven specific metrics for measuring the complexity of Smalltalk methods. These metrics have proven to be suitable for real-time feedback. Each metric is discussed in terms of its relationship to complexity, and ways to improve methods that the metric indicates are too complex. Finally, the chosen metrics are applied to approximately 35 000 methods in the three major dialects of Smalltalk. The results indicate that it is reasonable to expect developers to implement very large systems with relatively few violations of the metric guidelines. The last section concludes with a discussion of some of the effects observed in day-to-day use of the tools.

What is complexity?

When we say that a system is complex, we mean, at the very least, that it has a large number of parts. But we mean more than that. A beach has a vast number of grains of sand, yet we usually do not consider sand to be complex. Complex systems have many different *kinds* of parts, and the many parts connect to, or interact with, one another in many ways. Thus, two characteristics of complex systems are *numerosity* (both of elements and kinds of element) and *interconnectedness*. In the case of an object-oriented (OO) system these notions can be made more concrete. We consider an OO system to be complex when it consists of many objects (parts) of many classes (kinds of part) that collaborate (interact) with one another in many ways.

Numerosity and interconnectedness are the most visible aspects of complexity, but they do not fully account for the difficulty we have managing complexity in software engineering. Two other aspects of complexity must also be taken into account. One—

cognitive complexity—has to do with human ability to understand a system. The other—*adaptive complexity*—has to do with how systems change or evolve over time.

Cognitive complexity. Software is a creation of the human mind, so software complexity must in part be a cognitive issue. Cognitive complexity is that which hinders human understanding of how the software is designed and constructed and how it functions at run time. Numerosity and interconnectedness play a role in cognitive complexity because people have limited abilities to understand and remember the details about and the relationships between many distinct entities. As we say, "The devil is in the details." Nonlinearities also play a role. People tend to reason inaccurately about nonlinear processes—even simple ones such as exponential or combinatoric growth. Modern software systems routinely involve such a large number of elements interconnected in so many nonlinear ways that human abilities are inadequate for the task of completely understanding these systems.

Details come in various forms that challenge human cognitive abilities differently. There are persons who cannot remember a 12-digit number but have no trouble remembering the names of dozens, if not hundreds, of friends and acquaintances who are enmeshed in a complex web of personal and business relationships. We are aided in dealing with the multiplicity of people and relationships by our ability to classify much of the detail into categories with similar and familiar behavior. For example, we understand much about people in terms of their roles and responsibilities, job titles, positions in organization charts, status as employee or customer, and so forth.

Object-oriented analysis and design tends to give rise to models that mirror much of the familiar understandable domain relationships. When an OO design is fresh, many of the classes, their responsibilities, and the messages they understand reflect familiar aspects of the problem domain. This similarity helps those working with the system to better understand its function and reduces the cognitive complexity of the system. However, the correspondence between the OO system and the problem domain requires a carefully constructed balance of trade-offs among the desired function of the program, domain requirements, platform constraints, and strengths or weaknesses of the language and class library. Over time, changing requirements disrupt these careful trade-offs, thereby causing changes to the system that stretch the fidelity of the models beyond the breaking point. The corresponding growth of cognitive complexity is slow in the early stages of this process and increases rapidly as the models lose touch with the domain. When that has occurred, judicious refactoring can restore an understandable correspondence between the model and the new-

Software is a creation of the human mind, so software complexity must in part be a cognitive issue.

behavior of the system. Such refactoring reduces cognitive complexity without necessarily reducing the number of classes or their interactions.

Adaptive complexity. Although software begins as an abstract creation in the human mind, once the idea leaves our minds it takes concrete forms—both a source code form and an executable form—that have lives of their own. Software executes in a computer that is indifferent to numerosity and interconnectedness (not to mention cognitive complexity). As it does so, it reveals behavior not foreseen by its developers. Source code also has a life of its own. For example, a class written by one person with one intent suggests other uses to other team members. Finally, when the software is released to end users, it becomes enmeshed in the business or social problem domain for which it provides a solution. Users of the software discover unforeseen ways in which it can be used or could be used if it were just changed slightly. This notion that software has a life of its own reflects the fact that software is one element of an evolving dynamic system, a system that involves the software, the hardware on which it runs, the people who develop it, and the people who use it.

Adaptive complexity refers to the way in which the numerosity and the nonlinear interconnectedness of software affect the evolution of the system. These issues are the subject of the relatively new science of complex adaptive systems. (For an introductory review, see Waldrop[4] or Nicolis and Prigogine.[5]) Researchers in that field view complexity as a poorly understood but nonetheless *objective* property of

adaptive systems. Software is but one of many such evolutionary systems.

Complex adaptive systems are ones in which the number, the nature, the interconnections, or the interactions of the elements in the system are changing over time in ways that affect and are affected by the system's own complex structure, i.e., they feed back upon themselves. Numerosity and interconnectedness play a role in adaptive complexity because the more elements and interconnections there are, the more possibilities there are for feedback. Our intuitions about such systems lead us to expect them to become increasingly chaotic as the number of feedback loops grows. Yet the phenomenon that complexity theorists find common to a wide variety of complex adaptive systems is self-organizing emergent behavior. That is, complex adaptive systems behave in a counter-intuitive manner: order emerges out of chaos.

Emergent behavior from complexity is ubiquitous in the world around us at every physical scale and in many kinds of systems. Molecules emerge from complex interactions between the outer electrons of atoms. Ecologies emerge from complex interactions between various species, geographic features, and weather. Social organizations (groups, clubs, businesses, etc.) emerge from complex interactions between people. Hurricanes emerge from complex interactions between warm ocean currents, atmospheric humidity and temperature gradients, and winds. Monopolies and cartels emerge from complex interactions within disordered economic markets. And highly ordered self-perpetuating patterns emerge from random initial states in cellular automata such as John Conway's game of "Life."[6-8] The emergence of each of these macro-level patterns is thought to be an inherent and inevitable result of the complexity of the substrate level.

Computing systems are complex adaptive systems in two quite different realms: their execution environment and their development environment. Executing programs involve many interconnected elements that adapt to the execution environment: memory usage, compute cycles, file system space, input from users, and perhaps other programs (especially in networked client-server environments). The macro behavior of a running program emerges from these interacting elements. To as large a degree as possible, the behavior that emerges from a program should be that which is expected by the developers. But in complex systems, emergent behavior is seldom if ever completely predictable. Unexpected pernicious emergent behavior at run time is called a *bug*. We strive to reduce unnecessary complexity in part to reduce unwanted emergent behavior.

Computer programs exist in human-readable form separately from the form in which they execute. The many interacting elements of this realm are source code constructs, design elements, development tools and environments, and human developers. Throughout the life cycle of a system, designers and programmers add, remove, and change design and source code elements to adapt the system to changing requirements or changing understanding of existing requirements or existing system behavior, or to both. These additions and changes depend on the preexisting state of the system. Therein lies the feedback that fuels a complex adaptive system. Some elements engender others like themselves, e.g., they are used as patterns or conventions or invite "cut-and-paste" reuse. Some invite specialization by subclassing (inheritance reuse) or reuse in collaborations with new objects. And some resist change or reuse because their cognitive complexity makes them difficult to understand or change. Resistant elements may cause suboptimal changes to be made elsewhere. These suboptimal changes tend to increase the complexity of the elements in which they land. Complex areas that cannot be avoided tend to become even more complex because changes to them are more likely to be made without understanding the cleanest way to solve the problem. In general, repeated changes tend to increase complexity and reduce understandability. This sort of self-reinforcing creeping complexity eventually renders the system so brittle that often it must be abandoned. Real-time metric feedback, as proposed here, aims at reducing creeping complexity during development and maintenance by making the effect changes have on complexity more visible.

Complexity in Smalltalk systems

Smalltalk systems are structured at multiple levels of organization: the web of interacting instances, the collection of interconnected and interdependent classes (which are related by a tree of inheritance—"is-a"—relationships, a web of attribute—"has-a"—relationships, and a web of collaborations), and the parallel and partially independent tree of metaclasses (i.e., the class objects themselves that provide the class behavior as opposed to the instance behavior of the class). Often additional organized structures, such as groups of collaborating abstract classes that

form frameworks, are embedded within and across these other structures. Those who design, build, test, and maintain the system must explicitly understand and deal with each of these levels of organization. Each level hinders or facilitates understanding of the system in different ways and therefore contributes its own sort of cognitive complexity. As the system grows and changes during its initial construction and later evolves in maintenance phases, each level affects and is affected by this evolutionary process differently; therefore, each level contributes differently to the adaptive complexity of the system.

Methods. Individual methods define the atomic collaborations of the system. Their existence in a given class defines which messages can be sent to objects of that class, and the code within them defines the precise series of collaborations that implement the computation. Each method is a nexus of interconnections (i.e., collaborations) between objects.

The number of objects involved in a method and the number of collaborations specified in the method clearly affect the complexity of the method. Methods most commonly contribute to complexity by doing too much or doing it too procedurally. In addition, a method may contribute disproportionally to cognitive complexity if its behavior is awkwardly chosen or poorly named.

Overly long and complex methods tend to resist change and reuse.[9] A disciplined, experienced Smalltalk developer will break up, refactor, and clarify long complex methods. But the longer and more complex a method is, the more difficult the task tends to be and, hence, the more tempting it is to avoid. If the necessary experience or the discipline, or both, to simplify long or complex methods is lacking, such methods tend to get longer and more complex as new requirements force changes. Thus, complexity begets more complexity. This is the essence of adaptive complexity.

Classes and metaclasses. Individual classes contribute cognitive complexity if they are poorly matched to the domain (i.e., model it badly), are poorly named, or define poorly named or poorly factored methods so that their purpose is obscure. Or a class can attempt to do too much or too little. Classes contribute to overall numerosity and interconnectedness by defining attributes (e.g., instance variables) and methods. Unlike the case of individual methods, smaller classes are not necessarily better. Classes with little behavior may be entirely appropriate to the domain, or they may simply be thinly disguised data structures. In the latter case, they seem simple only because they are manipulated by other (usually overweight) classes that are thinly disguised procedural programs. The apparent simplicity of such lightweight data structure classes is illusory if the behavior that must be understood to reuse or change them resides in other large, overly procedural classes that are difficult to understand, reuse, or change.

The characteristic way in which classes grow more complex is by accumulating methods that do not quite fit with the previous responsibilities of the class. However, increasing the number of methods in a class does not necessarily add complexity. For example, the number of methods increases when a long, overly complex method is broken into two or more simpler methods, yet the result may be a simpler class. The complexity of the class increases when new *kinds* of behavior are added.

Classes themselves are objects that provide class variables and class methods. The majority of classes define no class variables and implement no class behavior. For that reason, programmers (especially novices) may find class behavior confusing when it is present. Class variables can add complexity because they are shared by all instances of the class and therefore can provide interconnection between otherwise isolated objects. Class methods do not, in general, increase connectedness because they are invoked only by messages to the class object (and its subclasses), not to instances of the class. The most typical class behavior is instance creation and initialization. Cognitive complexity can arise if the initialization is not straightforward. Examples of other types of class behavior are: maintenance of instance uniqueness (e.g., *Symbol* or *Character*) and delegation of instance creation to other classes (usually the newly created object is an instance of some appropriate subclass). In very rare cases, class behavior modifies the class itself. The class determines the behavior of all instances, and that behavior may be changed programmatically at run time. To do so, however, dramatically increases both the cognitive and the adaptive complexity of the system.

Inheritance. Inheritance relationships between classes affect complexity in ways that may not be apparent from an examination of the individual classes. A careful examination of the inheritance tree might show that behavior is misplaced. A couple of classes might duplicate behavior that could be shared if a new abstract superclass were created from which they

can both inherit. Or a class may do less than it could, thereby forcing its subclasses to implement more behavior than necessary. In both cases the lack of behavior (or the lack of behavior in the "right" place) has the potential for promoting complexity as new subclasses are added. Experienced developers may well spot the opportunity for refactoring in these cases. If not, the flaws in the hierarchy grow as new classes are added. Thus, the flaws represent adaptive complexity.

The cognitive complexity of a framework depends on how well matched it is to the needs of its subclasses.

A more subtle misuse of inheritance occurs when a class inherits behavior that cannot be used, presumably in order to inherit other behavior that is desired. This type of inheritance adds substantially to complexity. The developer who chooses such improper inheritance may clearly understand which inherited messages must not be used. Subsequent developers have a difficult task deducing that information.

Collaboration. Two related collaboration webs are involved in an OO application: the web of collaborating classes and the web of specific collaborations between instances that accomplish a computation. In the Booch method of OO analysis and design,[10] the former is described with class diagrams and the latter with object diagrams and interaction diagrams. Collaboration complexity has to do with how extensive and tangled these webs may be and how "naturally" they model the interactions in the problem domain.

The webs determined by the attributes (i.e., instance variables and class variables) of the classes are generally subwebs of the collaboration webs because an object usually has collaborators that are not attributes (e.g., they are passed to the object as message arguments). These subwebs of "containment" contribute complexity that is distinct from that of the collaboration web whenever webs of objects must be exported or imported from external databases, e.g., in client/server environments. OO databases can manage persistent storage of webs of objects with arbitrary references to one another, although not always without performance implications. But the most common case in commercial OO systems is that in which webs of objects must be stored in and retrieved from relational databases. The constraints imposed by the requirements of relational models add to the cognitive complexity of a system when they conflict with principles of good OO design.

Frameworks. Frameworks are webs of abstract classes that are meant to be extended by subclassing to fit the specific requirements of the application. What differentiates frameworks from other abstract classes is that concrete subclasses are intended to specialize a specific set of abstract collaborations between the inheritance trees that descend from each of the abstract classes in the basic framework. In large measure, frameworks exist to provide important inheritable collaborations. Their impact on complexity therefore involves issues of both inheritance and collaboration.

The cognitive complexity of a framework depends on how well matched it is to the needs of its subclasses. A good framework that is easy to understand can anchor a large set of subclasses. Such a match tends to lower the overall cognitive complexity of the set of subclasses in the framework because they share a stereotypical collaboration. However, it may increase adaptive complexity because a change that affects one of the abstract collaborations can ripple in unforeseen ways to all the subclasses that inherit from the framework.

Interactions between levels. The complexity within each of these levels clearly contributes to the overall complexity of a Smalltalk system. Just as importantly, the relationship between the levels also contributes to the complexity. Since changes to the system usually affect more than one level, the effect of a change on overall complexity can be difficult to assess. In some cases complexity within one level can be reduced without impacting other structures. Poorly written methods, for instance, can often be made simpler with no changes outside the method itself. Occasionally, we may be able to refactor a system in a way that simplifies more than one level at once—perhaps one or more classes are seen to be superfluous and disappear entirely, while code that relied on those classes becomes simpler as well. More often, however, refactoring reduces complexity at one level by judiciously adding complexity to another.

A new class may be added to allow simplification of methods or collaborations elsewhere, or an abstract class may be added to simplify many of the classes that inherit from the new abstract class. Either case raises the complexity of the class hierarchy. Addition of a new framework may increase complexity at many levels in exchange for a slower rate of growth in complexity as future specializations of the framework are added.

Understanding these issues well enough to measure the effects of changes that move complexity from one level to another is, at best, a distant possibility. For the foreseeable future, we must trust the explicit redistribution of complexity to the judgment and experience of skilled people. The only plausible near-term goal is to measure complexity within these levels.

Measuring complexity

Our experience with everyday measurement of the physical world tempts us to assume that once we understand a particular notion of complexity, it is a straightforward matter to measure it. That assumption stems from our experience with measurements of simple physical properties such as length or mass. We are aware of potential problems with measurement accuracy, but we take for granted the meaningfulness of notions such as "average length" or "twice the mass." Even in the physical world, though, measurement is not always so simple. Hardness is a readily measurable property of physical objects, and it is meaningful to say that one substance is harder than another. Color, too, is measurable. Yet the notions of "twice as hard" or "average color" are meaningless. When we step away from the physical world, measurement often becomes even more problematic. Consider the well-known difficulties of measuring human intelligence or product quality. The problem is that the underlying empirical properties of color, hardness, intelligence, or quality do not behave compatibly with all of the properties of numbers.

The theory of measurement has long been the subject of rather deep theoretical study that aims to rigorously characterize the properties of empirical structures required to support meaningful numeric manipulation.[11] The empirical structures of interest here are software structures such as methods, classes, inheritance hierarchies, and the like. We wish to map these structures into numerical representations so that we can discuss and reason about one aspect of software, its complexity, in isolation from the many other aspects of the software. The difficulty is that numbers can be assigned to empirical structures in infinitely many ways. Some are useful, and others are not. The theory of measurement provides a framework within which to choose among these numerical representations. Before we leap into measuring software complexity, it is worth considering the issue of just what we can expect to learn from such measurements.

Metrics and meaningfulness. Both the properties of the empirical structures and the way in which we choose to map them into numbers determine how we can use the resulting measurements. For an attribute of software (e.g., complexity) to be measured in a meaningful way, it must exhibit properties that can be mapped consistently into analogous properties of the number system.[12]

To meaningfully compare different chunks of software in terms of which is more complex, the measure must map chunks of software into numbers so that the ordering relationship of the "natural complexity" of the software maps appropriately into the natural ordering of numbers (i.e., $x <= y$). Such mapping requires that we be able to say which of two software artifacts is more complex than the other. More formally, for any two artifacts A and B, we must be able to say either that artifact A is at least as complex as artifact B, or that artifact B is at least as complex as artifact A. And if both statements are true, the two artifacts must be of equal complexity. The natural order of complexity must also be transitive: if artifact A is at least as complex as artifact B and artifact B is at least as complex as artifact C, then it must be the case that artifact A is at least as complex as artifact C. An empirical structure that obeys these necessary ordering relationships can be mapped into an ordinal scale. That is, a number can be chosen for each artifact such that the natural order of the numbers agrees with the natural order of the complexity of the structures.

It is another deceptively large step from meaningfully answering the question of which artifact is "more complex" to answering the question of "how much more complex?" The first question simply requires that the mapping not violate the numeric ordering relation. The second question, in essence, requires that the mapping preserve the much stronger relationship of numeric addition. To preserve this relationship we need a clear notion of what it means to combine, or concatenate, two chunks of software into a single chunk for which it can be assumed that:

- The complexity of the concatenation of any two artifacts is at least as large as the complexity of either artifact alone (monotonicity).
- The complexity of the concatenation of any two artifacts is independent of the order in which they are concatenated (commutativity).
- The complexity of the concatenation of any three artifacts is independent of which pair is concatenated first (associativity).

These properties ensure that the operation of concatenating two software artifacts maps into the operation of adding their complexity measures. If the empirical structure supports an ordinal scale and also supports additive concatenation, it becomes an *interval* scale. Interval scales support computation of the familiar statistics that rely on addition, such as means, standard deviations, and correlations. Ordinal scales do not.

To make statements such as "artifact A is twice as complex as artifact B," the notion of zero must be meaningful. Zero implies that there can be software artifacts with no complexity. It also implies that the concatenation of artifact A with artifact B has the same complexity as artifact A, if and only if artifact B has no complexity. The presence of an identifiable zero point transforms an interval scale into a *ratio* scale.

The above qualitative statements about complexity are examples of a set of properties that support meaningful measurement. They provide enough of a theoretical foundation to decide, for any particular notion of complexity, what sort of measurement scale can be supported. Others may be more suitable in some cases.[13-15]

In the field of software measurement, Weyuker[16] was one of the first to propose "desirable properties of complexity measures." She discusses nine properties that she thought software complexity metrics ought to have. But she only proposed them as a starting point for further study. She made no claims that these properties form a consistent axiom system that is either necessary or sufficient as a foundation for new complexity measures. As it turns out, they are not sufficient and have been shown to be internally inconsistent.[17] Some recent papers present a somewhat more rigorous discussion of the theoretical issues of the measurement of complexity.[18-20]

Meaningful Smalltalk metrics. The many levels of complexity in OO software present challenges for those who seek to create comprehensive complexity metrics. These levels are fundamentally different from one another. Some do not support even a linear ordering relationship. Others support ordering but not concatenation operations.

Within individual methods, orderings can be established in many sensible ways. It does not stretch credibility too much to assume that concatenation amounts to appending code from one method to that of a second method (even though that may be semantically meaningless in most cases). This kind of concatenation is analogous to the measurement of length where concatenation amounts to placing two physical objects end-to-end for the purpose of measuring their combined length. Given this sort of concatenation, the method-level metrics proposed later in this paper satisfy the ordering and concatenation requirements of a ratio scale.

The foundation for complexity of a class is not as firm. Some notions of class complexity support plausible orderings: the number of instance variables (or class variables) defined or inherited, or both, the number of methods defined or inherited, or both, and so forth.[18] But these orderings depend on whether we take into account only what is defined by the class or also include what is inherited by the class. We routinely and casually talk about the complexity of a class in terms of what it defines despite the fact that a class encompasses everything it inherits as well. It is tempting to believe that a class that defines one instance variable and a couple of methods is less complex, independent of its place in the hierarchy, than one that defines more state or behavior. This belief seems plausible from the perspective of cognitive complexity if we are willing to assume that one already understands its superclasses. From the perspective of adaptive complexity this seems much less plausible since the new class interacts with other classes with all its behavior—inherited as well as defined. Consider the following example of two actual classes in IBM Smalltalk:

- Class A defines no instance or class variables, eight instance methods, and one class method.
- Class B defines no instance or class variables, seven instance methods, and two class methods.

On the face of it, neither class seems very complex, and there is little to choose from in deciding which one is more complex than the other. (Note that we are ignoring the complexity of the methods themselves. In this case neither class has especially complex methods.) Inheritance changes the picture

rather dramatically. Class A is *Array*, which inherits from *Object*, *Collection*, and *SequenceableCollection*. Class B is *Boolean*, which inherits directly from *Object*. *Array* inherits a large body of relatively complicated behavior from its superclasses, most of which is used by arrays. *Boolean* inherits a number of methods from *Object* (more than most programmers realize), but very little of the inherited behavior is ever used. Moreover, inheritance plays an additional role in this case: *Array* is a concrete class that stands on its own, whereas *Boolean* is an abstract class. The behavior *Boolean* implements is meaningless apart from the behavior of its subclasses: *True* and *False*. There is a case of subtractive inheritance in *Boolean* as well: it overrides the class *new* method to disable it. When proposing orderings between classes, we must take issues of inheritance into account.

The incremental and iterative development style of Smalltalk development is well-suited to real-time method-level metrics.

Inheritance complicates the notion of concatenating classes as well. Concatenation of two classes that share the same superclass (i.e., sibling classes) could be taken to be the creation of a new class with all the variables and methods defined in either original class. This concatenation is somewhat analogous to pouring two liquids into a common container for the purpose of measuring volume. However, a metric that is restricted to a family of sibling classes is of little use. The situation is less straightforward if one of the two classes descends from the other and overrides methods in its superclass. In that case concatenation may result in the removal (via override) of behavior as well as the addition of new behavior. Concatenation then has aspects of both addition and subtraction.

Establishing an ordering or concatenation operation for other aspects of OO software is even more problematic. We lack obvious candidates for the proper ordering or sensible concatenation of webs of collaborating objects, class-metaclass relationships, inheritance trees, or frameworks. We therefore do not and cannot have a theoretically sound way to measure all of the contributors to the complexity of an OO system. Hence, the very notion of measuring overall complexity is inherently meaningless (as others have noted on different grounds[19,21]). The only levels that seem promising at this time are the method and class levels, and the class level may well support no more than ordinal metrics. For the foreseeable future, we must be content with the goal of constructing reasonable measures of the complexity within these tractable levels of organization. This outcome echoes what Fenton[19] proposes: "... the most promising approach is to identify specific attributes of complexity and measure these separately."

Real-time measurement and feedback

Even a "perfect" complexity metric—one that has a clear intuitive connection to complexity, theoretical validity, empirical validation, and perfect accuracy—cannot by itself reduce the complexity of an application. Consider the analogy of dieting and weight loss. The measurement of physical weight presents no theoretical and few practical difficulties. Weight measures are certainly empirically valid and can be as precise as one desires. But as millions of overweight dieters can attest, the ability to accurately measure their weight does not much help in weight reduction. A pound gained or lost is the cumulative effect of many events and conditions (calories eaten, calories burned in exercise, metabolic rate, etc.) that span days. So too with creeping complexity. Feedback at the time each decision is made is best rather than feedback about accumulated decisions that are difficult to undo. The accumulation of reductions in local complexity reduces global complexity.

The incremental and iterative development style of Smalltalk development is well-suited to real-time method-level metrics. Smalltalk code browsers display code one method at a time. When methods are added or changed, each is compiled before moving on to the next. Since methods are typically small, each addition or change is done in a few minutes (or sometimes a few seconds). This short period provides the opportunity to display feedback on the complexity of new or changed methods very soon after the code is written. The same complexity assessment can be applied as a developer views each existing method. Since Smalltalk developers typically spend much more time browsing and reading code than they do writing code, having real-time metric feedback as they browse provides an ever-present sense of the

complexity of a wide-ranging sample of the code they are reusing.

A Smalltalk development environment with real-time metrics was first demonstrated at OOPSLA'92 by Knowledge Systems Corporation. Whenever the developer browsed or changed a method, a simple indication of the complexity and readability of the method automatically appeared in a metric feedback pane in the browser. Another system with metrics available upon request was prototyped and described by Barnes and Swim.[22] That system does not have real-time feedback, but it has a "Quality!" menu item on the primary browser that allows the developer to easily obtain a metric report for the selected class or method, or both.

Requirements for real-time metrics. If the kind of feedback and its manner of presentation are not well-chosen, real-time metrics could become overbearing and irritating. In appearance and in content, real-time metrics must help rather than distract, constrain, or overwhelm the developer with unasked-for information.

The following principles were used to guide us in providing real-time metric feedback:

- Information should be presented for several metrics—no one metric tells the whole story.
- Metrics must be rapidly computed—delays are annoying.
- Metrics should be presented in a manner that is ever-present yet unobtrusive so that developers are always, if subliminally, reminded of their responsibility to manage complexity.
- Metrics must be presented in a manner that is immediately interpretable. No thought should be wasted on metrics unless thought is needed.
- The developer should be concerned with the degree of complexity only when it exceeds a threshold.
- Metrics should be able to explain themselves. If the coarse initial feedback indicates a possible problem, the developer should be able to easily obtain more detailed information.

Above all, feedback should be advisory. Smalltalk coding is fine-grained design (i.e., detailed design of collaborations), and design is a quintessentially human activity. No metric system can reliably second-guess an experienced developer. The designer or programmer must be able to make use of or ignore complexity feedback according to its value in the circumstances of the moment. There are circumstances, such as working on a rapid prototype, when the developer consciously and properly ignores complexity. There are other circumstances (e.g., when browsing methods that have been automatically generated by tools) where the metric feedback should simply be ignored.

A system that meets these criteria has been developed in the IBM North America Object Foundry. The system works as follows. Whenever a method is displayed or changed in any of the browsers, seven metrics are computed for the method. The computation is quick, seldom taking more than a fraction of a second. Unless the method is very large (and therefore far too complex), there is no noticeable delay as the user browses or changes code. The value of each metric is compared to two thresholds. One marks the upper bound of routinely acceptable complexity for that metric, and a second larger threshold marks the point at which the metric signals excessive complexity. This comparison divides the values for each metric into three categories that provide broad guidelines within which the programmer can exercise a great deal of freedom.

For the purpose of visual feedback, the three categories so defined are denoted by colors: *green* if the count is below the warning threshold, *yellow* if it is between the warning threshold and the unacceptable threshold, and *red* if the count exceeds the unacceptable threshold. The use of color satisfies the requirement that feedback be immediately interpretable yet unobtrusive. The visual feedback is displayed in a small graphic button that has been added to each of the standard Smalltalk browsers. This button contains a rectangular region that displays a colored subregion for each metric. The user may click this button to obtain a new window that presents more detailed information about the metrics for the method. In addition to the colored regions on the button, a face icon acts as a summary indicator that smiles if all metrics are green, looks somewhat quizzical if one or more are yellow, and scowls if any are red. This icon serves as a unified summary of the metrics and provides redundant feedback for color-blind developers.

On request, the system can present "help" information for each metric that explains the common mistakes that lead to a poor metric together with suggested ways to simplify the code. Also, a "quality" menu added to the browser provides various choices for browsing or obtaining a formatted report on all

"red" or "yellow" methods in a group of methods (e.g., all "red" methods in a class).

Defining the thresholds. The choice of thresholds between the categories is inherently somewhat arbitrary. The thresholds presented in the next section were evaluated in three ways. First, the chosen thresholds have been reviewed by more than a dozen very experienced Smalltalk developers. Legitimate differences of stylistic opinion generate disagreements with the exact choices for thresholds. Some argue that one or another threshold should be more strict on the grounds that many methods will receive a green evaluation even though they could easily be written more simply. Others argue for more lenience on the grounds that certain situations might excuse usage that exceeds the threshold. However, there is no consistent opinion that any one threshold is inappropriate. Perhaps in the future, "smarter" metrics may be able to take more context into account so that the thresholds can be stricter in the typical case and more lenient when lenience is warranted.

A second assessment of the thresholds is based on an analysis of nearly 35 000 commercial Smalltalk methods. The results are reported in the next section. These results indicate that the thresholds are reasonable and do not place undue constraints on typical commercial-quality code. The analysis covered the following bodies of code:

- 7768 methods in the Digitalk Inc. VOS/2** version 2.0 image (includes WindowBuilder** code from Object Share Systems, Inc.)
- 15 198 methods in the IBM Smalltalk version 2.0 image (includes ENVY/Developer** code from Object Technology International)
- 11 901 methods in the VisualWorks** version 4.1 image from ParcPlace Systems, Inc.

This body of Smalltalk code represents a wide variety of programming styles. It contains code written by dozens of programmers from at least five different companies. Some of the methods in VisualWorks were written by the originators of Smalltalk at Xerox PARC in the 1970s. Many methods in the base classes (e.g., collections, magnitudes, and streams) appear in all three images but are implemented differently in each. The analyzed code implements a wide variety of behavior: from graphical user interface code to operating system and file system interface code, from numeric algorithms to text formatting algorithms, and from code compilation to version management. The many programmers who wrote the code exhibit quite different styles and abilities. And some of the methods (often very long ones) were automatically generated by window layout tools. Despite these many differences in age, authorship, purpose, and polish, the metric analysis shows that, for most metrics, well over 90 percent of the methods are "green." Moreover, despite the very different origins of the three versions of Smalltalk, all three images show a remarkable similarity in their metrics.

A third approach for assessing the acceptability of the thresholds is to examine their effect in actual use. The real-time metric tool has been used in internal IBM development projects and in consulting engagements with IBM customers. It is also in routine use in training new Smalltalk developers. In addition to the analysis of commercial Smalltalk methods, we report results from an analysis of 3395 methods written by experienced IBM programmers using real-time metric feedback. Anecdotal evidence, confirmed by this analysis, shows that the thresholds are low enough to improve programming practices—even those of experienced developers—yet they are not overly constraining.

Since the chosen thresholds are admittedly somewhat arbitrary and experienced programmers occasionally have legitimate arguments for setting them differently, the question arises as to whether thresholds should be adjustable by the user. We advise against it. Perhaps no harm would come from allowing programmers to set thresholds more strictly. But adjustable thresholds are usually requested by programmers who wish to make the thresholds more lenient. These programmers should be reminded that the goal is not just to pass the metrics thresholds; it is to reduce complexity. Arguing for more lenient thresholds is arguing for the general acceptability of more complex code. Moreover, unless the different thresholds are adopted by all members of the development and test teams, a developer who increased the thresholds would fool no one but himself. If a good case can be made for exceeding a threshold in a particular method, the method need not be changed. Making a case for accepting a method with any red metrics should be difficult. But the yellow threshold is intended to be merely cautionary. Some methods, perhaps a few percent, are better left with yellow metrics. The decision to accept such methods should, however, be an explicit one that the programmer is willing to defend in a peer code review.

Method-level metrics

A large and growing body of literature on OO metrics is at the class level (see Chidamber and Kemmerer[18] for a good review). In comparison, there are very few discussions of method-level metrics. This situation is odd given that class-level metrics are conceptually much more difficult to motivate and understand than method-level metrics (see the earlier subsection on meaningful Smalltalk metrics).

Barnes and Swim[22] address method-level metrics for the OO language "Actor."[23] They measure McCabe cyclomatic complexity,[24] number of lines of code, number of local variables, number of messages to self, and number of messages to other objects.

Lorenz and Kidd[25] discuss five method-level metrics and also discuss thresholds that are similar to our cautionary (yellow) thresholds. They propose counting the number of message sends (threshold 9), number of statements (threshold 7), lines of executable code (no recommended threshold), and strings of message sends (no recommended threshold). They prefer to count message sends or statements rather than lines of code on the grounds that the number of lines of code may be affected more by stylistic differences. They also propose a "complexity" metric (threshold 65), which is a weighted sum of occurrences of various kinds of constructs found in Smalltalk methods, e.g., primitive calls, assignments, nested expressions, method arguments, and temporary variables. This metric is difficult to justify and interpret because it is based on an unsupported assumption that the effect of its components on complexity is additive and because the weights are assigned arbitrarily.

We have chosen to provide real-time feedback on the following seven metrics for each method. Each is an easily computed indicator of some aspect of method complexity.

- Number of lines of code
- Number of blocks
- Number of temporary variables and method arguments
- Number of parenthesized expressions
- Number of explicit returns
- Number of external assignments
- Number of cascaded messages

They are all ratio scales, each of which is intuitively related to method complexity or readability. No one of them can claim to encompass the notion of method complexity. Taken together, however, they form a sieve that catches most excessive structural complexity. None of them addresses the sort of cognitive complexity that results from poorly chosen names for methods and variables. That sort of complexity must still be judged by human code reviewers.

The following discussion includes, for each individual metric, a description of the metric, a brief justification of its relationship to complexity, the threshold values used to bound the green/yellow/red categories, a list of common situations that might lead to excessive complexity, suggested ways to change offending methods to reduce their complexity, and data on the distribution of measurements for Digitalk's VOS/2 version 2.0, IBM's Smalltalk version 2.0, and ParcPlace's VisualWorks version 4.1.

The text presented below in each of the "description," "justification," and "suggested improvements" subsections is available on line from the metrics tool in essentially the same form as it appears here.

Number of lines. Two decades of Smalltalk experience tell us that small is beautiful. Methods should be small, clearly written implementations of a single-purpose behavior, i.e., a method should do one thing and one thing well.[9] The metric used here counts all lines, including comments, with the exception that multiple-line comments are counted as a single line. This method of counting avoids discouraging good comments. The longer a method, the more likely it is to need a multiline explanatory comment. We wish to encourage the programmer to shorten the method, not the comment.

Description and justification. The more lines a method has, the longer it takes a reader to understand it, especially if the reader must scroll the text because it does not fit within a typical browser text pane. This length can inhibit the rapid understanding of the code necessary for quick browsing, debugging, and reuse.

Number of Lines	*Effect on Method Complexity*
x<=14	Green: OK, minimal if any impact
15<=x<=26	Yellow: Method may need refactoring
x>26	Red: Too long, refactor

Suggested improvements. The most common reasons for a method being overly long include:

1. The method contains leftover "first draft" code.

2. The method contains too much vertical white space (i.e., blank lines within the code).
3. The method contains extra comments in the code.

Long methods often signify "first draft" code. When code is first written for a class or a method, the details of the code are being discovered as coding progresses. First-time discovery is usually a messy process, the details of which tend to obscure possible simplifications. Often, similar behavior appears in multiple methods in the class or even in multiple sections of code within the same method. A second (or even third) pass is often needed to identify the common behavior, create new supporting methods or classes that embody that common behavior, and then recode the first draft methods to make use of these new constructions.

Used judiciously, white space can make code more readable and understandable. However, insertion of blank lines between sections of code often implies that those sections ought to be separate methods.

Well-written small methods do not require comments other than a single clearly written method comment. If a programmer finds the need for other comments, the method is probably too complex and should be broken into multiple methods.

Analysis of lines of code in commercial systems. Of the 35 000 methods analyzed, about 11 percent have more than 14 lines of code, and about 3.5 percent have more than 26 lines (Table 1). Thus, even without metric feedback, most methods written by experienced developers satisfy the metric criteria. Yet there remains substantial room for improvement on this metric. Analysis of 3395 methods written by IBM programmers with the benefit of real-time metric feedback shows that it can have a substantial beneficial effect: 6 percent of those methods have more than 14 lines of code, and only 1.2 percent have more than 26 lines.

Some small stylistic differences between the three commercial Smalltalk systems appear in the distribution of number of lines of code (Figure 1). Among small methods—those of six lines or less in length—VOS/2 methods tend to be shorter. For methods larger than seven lines, there is little difference between the three systems. For all three, the large majority of methods are seven lines or less.

Number of blocks. In this subsection we discuss the number of blocks as an indicator of method complexity.

Description and justification. The number of control structures in a given piece of code has long been used as a measure of the overall complexity of the code. The number of block constructs in a method can usually be equated with the number of control structures in the method.

Number of Blocks	*Effect on Method Complexity*
x<=4	Green: OK, minimal if any impact
5<=x<=7	Yellow: Method may need refactoring
x>7	Red: Too complex, refactor

Suggested improvements. The most common reasons why methods may contain too many blocks are:

1. The method is trying to do many different things.
2. The method implements a case statement.
3. The method implements a decision tree.
4. The method traverses a deeply nested object structure.

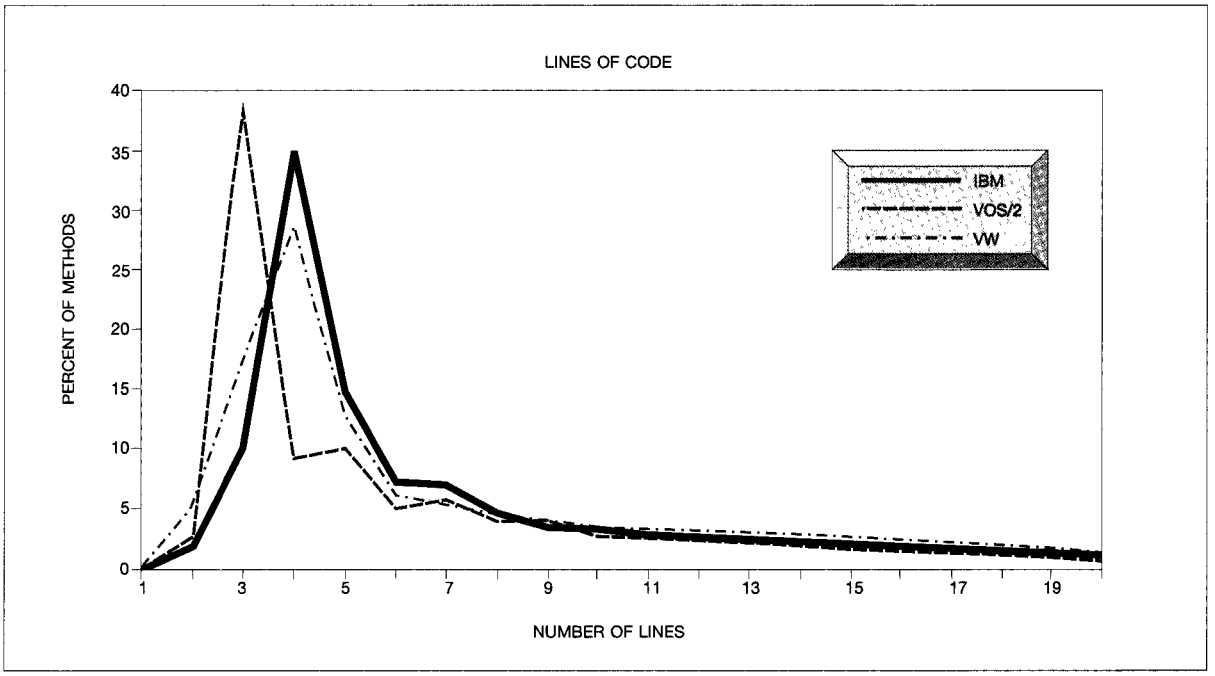
If the method has several isolated islands of code, each of which contains its own independent control structures (blocks), the method is usually trying to do too many things. A good rule of thumb in this case is that a method should do one thing well. Each island of code in the method should be considered a candidate for a separate new method.

If the method implements a case statement in which the cases are distinguished by the kind (i.e., class) of object held by a variable, consider refactoring the behavior using polymorphism. That is, convert the block to be executed for each case into a method in the candidate class. If the cases reflect common states of the receiver, consider refactoring your design to use the *State Object* design pattern.[26]

If the method implements a decision tree, break up the decisions into separate methods. Then the original method can act as an aggregator for these micro decisions without containing all the decision code. This will also make it easier for clients of this object to reuse the finer-grained behavior in ways that would be impossible with a single decision tree method.

If the method traverses a deeply nested object structure, consider moving the structural navigation code into new methods in the objects that make up the structure. Often, this will best be accomplished by applying the *Composite* design pattern.[26] This approach will simplify the method as well as add valuable behavior to the objects being traversed.

Figure 1 Distribution of number of lines of code



Analysis of blocks in commercial systems. Of the 35 000 commercial methods analyzed, about 7 percent have more than 4 blocks, and about 3.4 percent have more than 7 blocks (Table 2). As with the lines of code metric, there remains substantial room for improvement on this metric. Metric feedback had an even stronger effect in this case. Of the 3395 methods written with the benefit of real-time metrics, only 1 percent of methods have more than 4 blocks, and only 0.2 percent have more than 7 blocks.

The distribution of number of blocks (Figure 2) shows remarkably little difference between the three commercial systems. It also shows the degree to which polymorphism obviates the need for many control structures. Two thirds of the methods have no control structures at all. Another fifth of the methods have one or two, typically a single "ifTrue:" block or an "ifTrue:ifFalse:" pair.

Number of temporary variables and arguments. Another indicator of method complexity is the number of temporary variables and arguments.

Description and justification. Arguments and temporary variables typically hold objects to which messages are sent; i.e., they hold collaborators that provide behavior for the method. Having many collaborators implies too much complexity.

Table 1 Summary statistics for number of lines of code

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	87.5	89.0	90.8
Percent Yellow	8.3	7.4	6.5
Percent Red	4.2	3.6	2.7
Largest Value	331	753	111
Mean	8.13	7.89	7.06

Number of TempsAndArgs	*Effect on Method Complexity*
x<=4	Green: OK, minimal if any impact
5<=x<=8	Yellow: Method may need refactoring
x>8	Red: Too complex, refactor

Suggested improvements. The two most common reasons why methods use too many temporary variables or arguments are:

1. The method is trying to do many different things.

Figure 2 Distribution of number of blocks

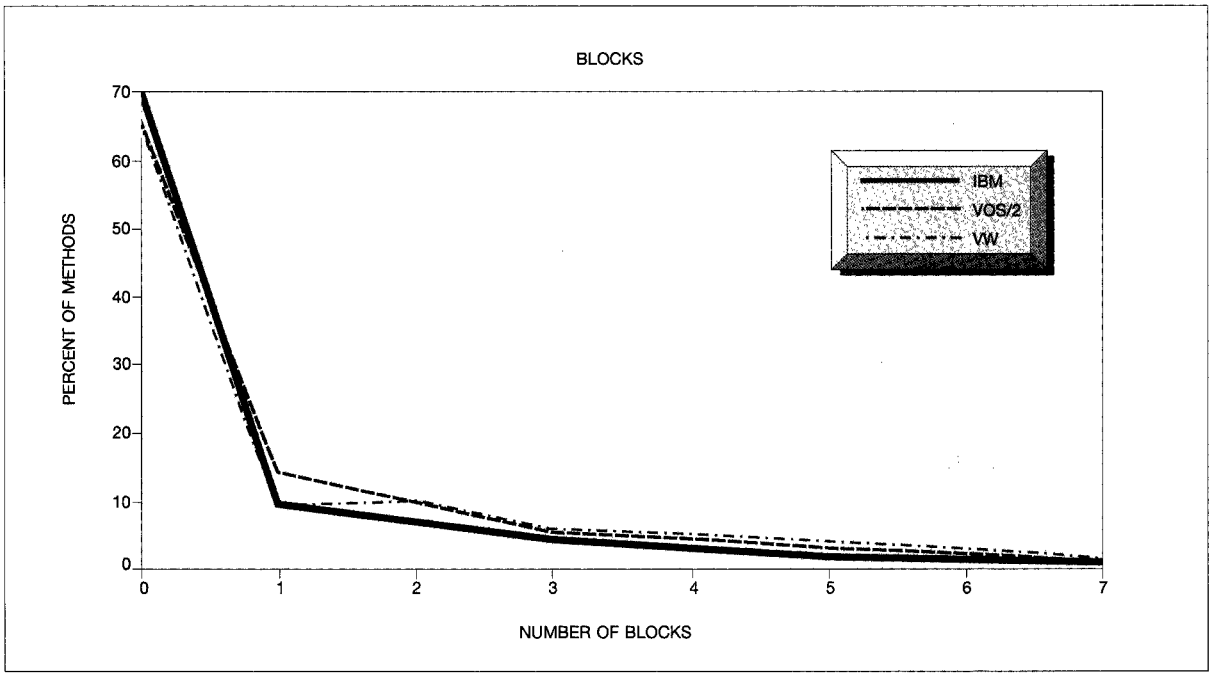


Table 2 Summary statistics for number of blocks

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	93.6	92.5	92.1
Percent Yellow	4.0	3.8	4.5
Percent Red	2.5	3.7	3.4
Largest Value	39	56	38
Mean	1.12	1.15	1.24

2. Several arguments are parts of what should be a single object.

Too many collaborators often indicate that the method is trying to do too many things; break up the method into smaller methods.

Too many arguments often means that a new object should be designed that encapsulates the arguments; i.e., create a single collaborator from a group of collaborators. The new object then delegates responsibilities to members of the group.

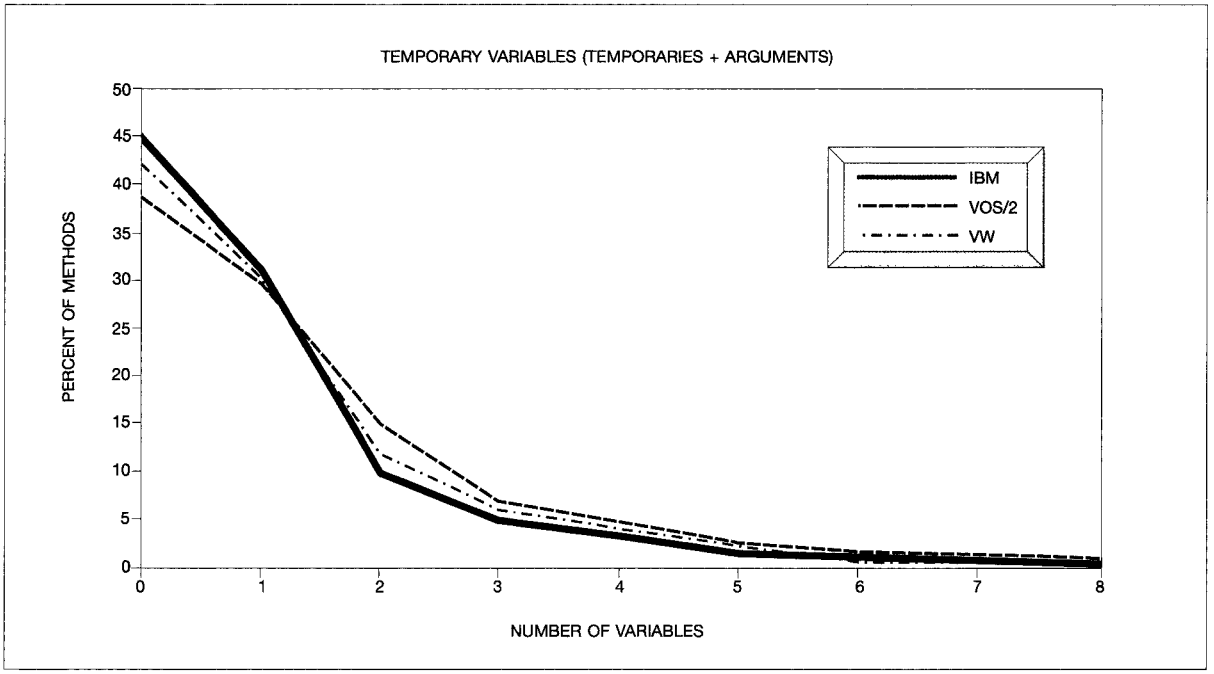
Analysis of temporary variables and arguments in commercial systems. Of the 35 000 commercial methods analyzed, about 5.5 percent have more than 4 temporary variables, and about 1.2 percent have more than 8 (Table 3). With the benefit of real-time metric feedback, we observed that 0.4 percent of methods have more than 4 temporary variables, and only 0.03 percent have more than 8.

The distribution of number of temporary variables and arguments (Figure 3) shows small systematic differences between the three commercial systems. VOS/2 programmers write fewer methods with no temporary variables or arguments and more methods with two than do IBM programmers. The programmers of the VisualWorks system use a style that is between that of the other two systems.

Number of parenthesized expressions. We now discuss the number of parenthesized expressions as an indicator of method complexity.

Description and justification. The number of parentheses can be used as a rough measure of the nestedness of a piece of code and is thus an attribute of the overall complexity of the code.

Figure 3 Distribution of number of temporary variables and arguments



Number of Parenthesized Expr.	*Effect on Method Complexity*
x<=5	Green: OK, minimal if any impact
6<=x<=10	Yellow: Method may need refactoring
x>10	Red: Too complex, refactor

Suggested improvements. The reasons a method may contain too many parenthesized expressions include:

1. Parentheses have been used unnecessarily.
2. The results of parenthesized expressions are receiving other messages.

In some circumstances parentheses not required by syntax may make code more readable. For instance, long arithmetic expressions may look more familiar with parentheses that are not strictly required by the syntax. In most cases, however, excess parentheses reflect the programmer's confusion or insecurity about Smalltalk parsing rules.

If parentheses are used for excessive nesting of expressions, assign intermediate results to temporary variables or break the method into smaller methods.

Table 3 Summary statistics for number of temporary variables and arguments

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	94.1	95.1	94.1
Percent Yellow	4.9	4.1	4.3
Percent Red	1.0	0.8	1.6
Largest Value	21	34	37
Mean	1.38	1.15	1.34

Analysis of parenthesized expressions in commercial systems. Of the 35 000 commercial methods analyzed, about 3 percent have more than 5 parenthesized expressions, and about 1 percent have more than 10 parenthesized expressions (Table 4). In part because there is little room for improvement in these data, results with real-time metric feedback were little different: We observed that with feedback, 2.3 percent of methods have more than 5 parenthesized expressions, and only 0.85 percent have more than 10. This metric will have its largest effect on less-experienced programmers who tend to use unneeded parentheses.

Figure 4 Distribution of number of parenthesized expressions

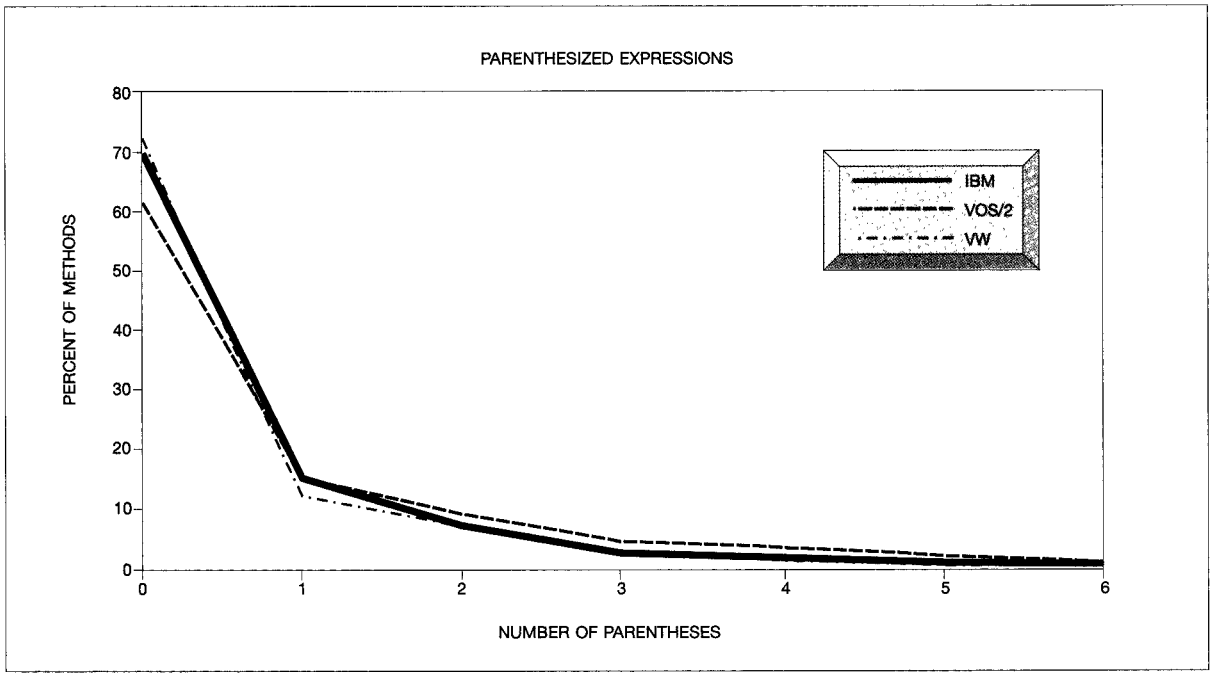


Table 4 Summary statistics for number of parenthesized expressions

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	95.0	97.8	97.2
Percent Yellow	2.9	1.6	2.1
Percent Red	2.0	0.6	0.7
Largest Value	113	182	39
Mean	1.2	0.66	0.70

The distribution of number of parenthesized expressions (Figure 4) as well as the summary data show a small but systematic tendency for VOS/2 programmers to use more parenthesized expressions than the others.

Number of explicit returns. In this subsection, we discuss the number of explicit returns metric.

Description and justification. In addition to the often implicit return at the end of each method, any block may end with an explicit return from the method. This possibility complicates the reader's task in determining what object the method returns and in which circumstances. Most methods should and do have a single return, although in some cases the method is more readable and understandable if more than one return is used. If more than a couple are needed, the method may need refactoring.

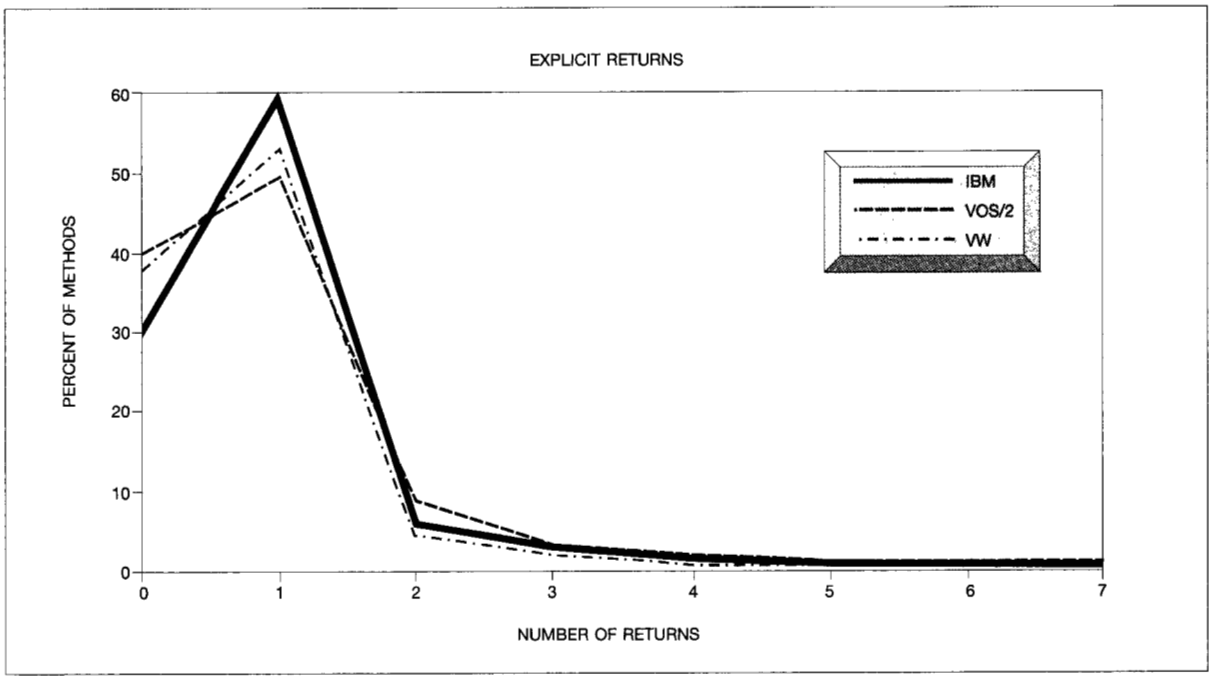
Number of Returns	*Effect on Method Complexity*
x<=2	Green: OK, minimal if any impact
3<=x<=4	Yellow: Method may need refactoring
x>4	Red: Too complex, refactor

Suggested improvements. The two most common reasons that methods may contain too many explicit returns are:

1. The method implements a case statement.
2. The method implements a decision tree.

The method may implement a case statement, each of which computes its own return value. If the cases are distinguished by the kind (i.e., class) of object held by a variable, consider refactoring the behavior using polymorphism. That is, move the return behavior into methods in the possible candidate classes and return the result of sending one message to the

Figure 5 Distribution of number of explicit returns



variable object. If the cases reflect common states of the receiver, consider refactoring the design to use the *State Object* design pattern.[26]

If the returns come from different branches of a decision tree, often the code can be simplified and the number of returns reduced by taking advantage of the fact that the result of an "ifTrue:ifFalse:" message is the value of whichever block is executed. A single return at the beginning of the statement replaces one in each of the branches. If the number of returns is simply a result of a large number of decision branches, break up the decisions into separate methods, each of which returns the appropriate object. Then the original method can act as an aggregator for these micro decisions without containing all the decision code. This will also make it easier for clients of this object to reuse the finer-grained behavior in ways that would be impossible with a single decision tree method.

Analysis of explicit returns in commercial systems. Note that these statistics are for explicit returns only. No attempt was made to account for implicit returns of *self* at the end of the method.

Table 5 Summary statistics for number of explicit returns

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	96.9	96.0	96.3
Percent Yellow	2.5	2.9	2.8
Percent Red	0.6	1.0	0.9
Largest Value	13	17	21
Mean	0.77	0.88	0.79

Of the 35 000 commercial methods analyzed, about 3.7 percent have more than 2 explicit returns, and about 0.9 percent have more than 4 explicit returns (Table 5). With the benefit of real-time metric feedback, we observed that 0.8 percent of methods have more than 2 explicit returns, and only 0.06 percent have more than 4 explicit returns.

The distribution of number of explicit returns (Figure 5) shows that the programmers of IBM Smalltalk tend to have one explicit return more often than do the others. An informal examination of code sug-

Figure 6 Distribution of number of external assignments

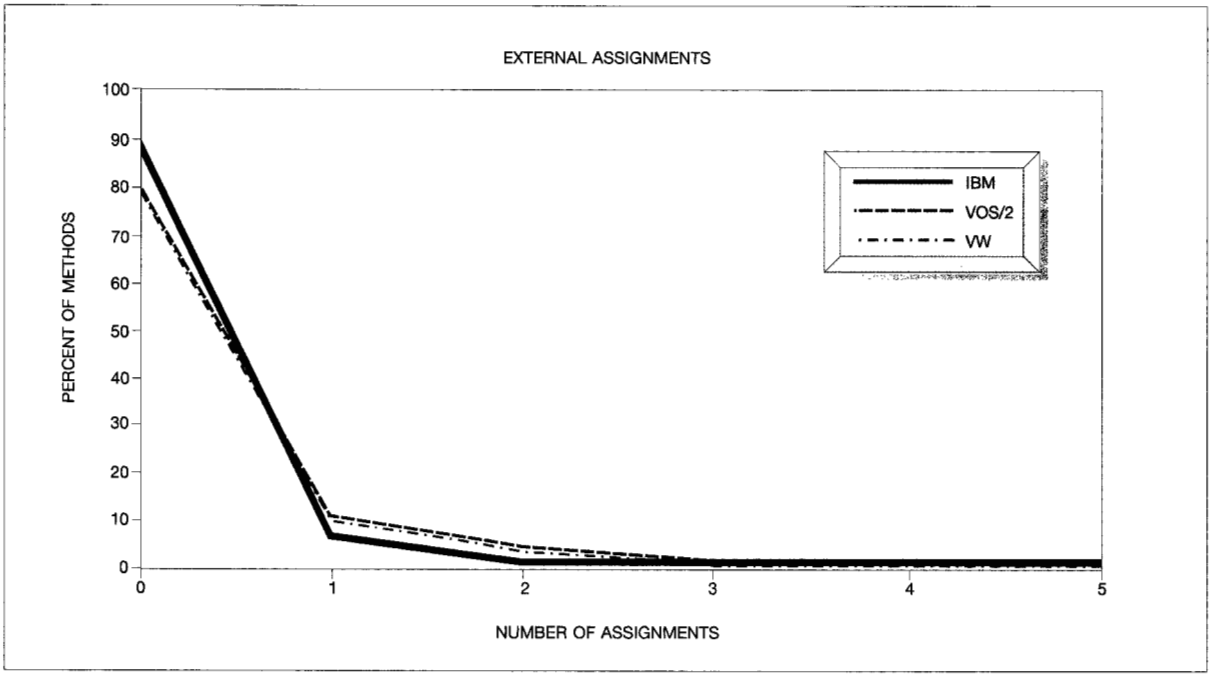


Table 6 Summary statistics for number of external assignments

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	95.5	97.8	93.8
Percent Yellow	3.4	1.6	4.3
Percent Red	1.1	0.6	1.8
Largest Value	26	21	35
Mean	0.26	0.15	0.49

gests that this result is due to a tendency in IBM Smalltalk to explicitly return *self* rather than to rely on the implicit return of *self*.

Number of external assignments. Another metric of method complexity is the number of external assignments.

Description and justification. This metric is a code complexity standard based on the number of external assignments in a method (that is, assignments to variables other than temporary variables). Assignments to variables other than the method's own temporary variables may create linkages that are difficult to understand. If so, they add to the overall complexity of the class or the system. The scope of the variables being set is also important. The larger the scope, the more impact a change to the variable may have, and the more difficult it is to understand. Instance variables affect only one instance, class variables may affect all instances of the class, and globals may affect objects anywhere in the system.

Number of Assignments	*Effect on Method Complexity*
x<=1	Green: OK, minimal if any impact
2<=x<=3	Yellow: Method may need refactoring
x>4	Red: Too complex, refactor

Suggested improvements. The reasons a method may contain too many external assignments include:

1. The method changes the state of many different variables.
2. The method implements a complex algorithm.
3. The method implements a decision tree for setting variables.

If many variables are being set, the variables may represent relatively independent aspects of the domain model that should be managed by different methods, or perhaps even different classes. In an initialization method, all variables legitimately may be initialized at once. Otherwise, independent aspects usually should be changed in separate "setter" methods. Also, when the variables of a group are naturally managed together, consider creating a new object that manages that group.

If the method is implementing a complex algorithm that requires many changes to variables, taking the trouble to think of ways to break the complex method into two or more simpler methods will pay dividends later in understandability and maintainability.

If variables are being set differently in different branches of a decision tree, break up the decisions into separate methods, each of which sets the variables appropriately. Then the original method can act as an aggregator for these micro decisions without containing all of the decision code. This will also make it easier for clients of this object to reuse the finer-grained behavior in ways that would be impossible with a single decision tree method.

Analysis of external assignments in commercial systems. Of the 35 000 commercial methods analyzed, about 4.3 percent have more than 1 external assignment, and about 1.2 percent have more than 4 external assignments (Table 6). With the benefit of real-time metric feedback, we observed that 0.1 percent of methods have more than 1 external assignment, and only 0.03 percent have more than 4 external assignments.

Both the summary data and the distribution of number of external assignments (Figure 6) show the use of more external assignments in VOS/2 and VisualWorks than in IBM Smalltalk.

Number of cascaded messages. The last metric we discuss is the number of cascaded messages.

Description and justification. Cascaded messages, or cascades, are sequential messages sent to the same receiver, separated by semicolons. Excess cascaded messages indicate highly procedural code. Note that in some cases a series of messages to one receiver is the simplest and most expressive way to implement the behavior of the method. In those cases, a series of cascaded messages is often more readable and understandable than alternative coding techniques.

Number of Cascades	*Effect on Method Complexity*
x<=8	Green: OK, minimal if any impact
9<=x<=16	Yellow: Method may need refactoring
x>16	Red: Too complex, refactor

Suggested improvements. There are three primary reasons why methods may contain too many cascades. These reasons are:

1. The method is trying to specify too many fine-grained behaviors.
2. The method is initializing many instance variables.
3. The method is appending a pattern of text to a stream.

In many cases a long list of fine-grained behaviors can be shortened by noticing groups of behaviors that are naturally meaningful in the domain and therefore are likely to be reused in other methods. Refactor by adding a method to the object receiving the cascade for each natural group of behaviors; then use these methods to replace the groups in the original method.

One style of initializing objects is to use a cascaded series of setter messages. If an object needs to initialize more than eight instance variables, the method complexity simply reflects a class that is perhaps too complex. Consider simplifying the class.

Adding text to a stream (e.g., in a "printOn:" method) often involves many cascaded messages combining "nextPutAll:" with "cr". In many cases the method could be shortened and made less complicated and more readable by creating and using new methods, each of which appends a meaningful portion of the output.

Analysis of cascades in commercial systems. Of the 35 000 commercial methods analyzed, only 1 percent have more than 8 cascades, and about 0.4 percent have more than 16 cascades (Table 7). Here again there is little room for improvement on this metric with experienced Smalltalk programmers. Results with the benefit of real-time metric feedback are essentially the same. This is another case where the effect of the metrics will be larger on less-experienced programmers.

Both the summary data and the distribution of number of cascades (Figure 7) show that VisualWorks

Figure 7 Distribution of number of cascaded messages

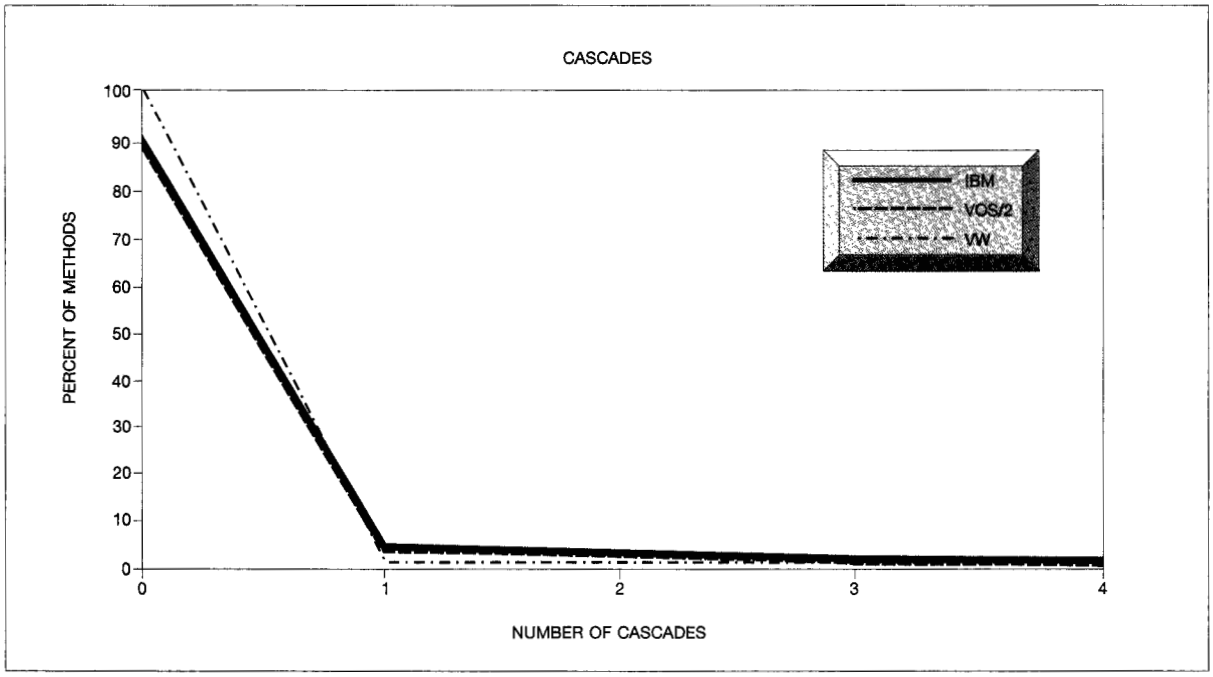


Table 7 Summary statistics for number of cascaded messages

	Digitalk VOS/2 2.0	IBM Smalltalk 2.0	Visual-Works 4.1
Percent Green	98.5	98.9	99.8
Percent Yellow	0.7	0.4	0.1
Percent Red	0.8	0.5	0.1
Largest Value	346	255	60
Mean	0.52	0.39	0.11

programmers use cascades more sparingly than do the others.

Conclusions

Experience report. The real-time metrics tools have been in use since April 1995 within the IBM OO Consulting Practice and since August 1995 in the Smalltalk training course in IBM Education and Training. It is too soon to draw conclusions about the long-term effect of real-time metrics on creeping complexity. But some short-term effects are clear.

We have found that experienced developers readily accept the metrics and easily make small adjustments to their coding habits so that "yellow methods" are uncommon and "red methods" very rare. Experienced Smalltalk developers already tend to write small, low-complexity methods that are easy to read. The metrics proposed here are in accordance with those practices, so most often the metric feedback on the developers' code will be all green. In that case, the metric feedback is a positive reinforcement to their good habits. In the infrequent cases where the metrics are not green, experienced developers typically modify their code immediately. The result is a small but consistent improvement in the quality of their code. The accumulation of these small improvements can have a substantial impact on a large project.

In some situations, e.g., working rapidly on a prototype, experienced developers deliberately accept overly complex code. When they later revisit and clean up the code, the developers use the metrics to quickly find the problem areas. They click through the methods in the browser, glancing at the color of

the metric feedback for each one, or they use the batch search options provided with the metrics tools.

Real-time metric feedback has a much larger impact on inexperienced developers who tend not to follow the good design and coding conventions of experienced developers. In some cases they may not have accepted and integrated the view that small is beautiful. In other cases they may still write complex "C-like" procedural code because they have not fully made the transition to the OO paradigm. For these people, the initial experience of the metric feedback is not as congenial. They typically see yellow or red feedback, or both, on a substantial portion of their methods. Not surprisingly, their first reaction to the metric feedback tends to be to challenge the metrics rather than to modify their code. If they work with more experienced developers who can give them guidance, this resistance soon subsides, and they quickly learn how to simplify their methods. Without such guidance the transition can be frustrating. Although the more detailed metric report with its on-line self-explanatory help information provides useful guidance, the on-line help is most effective as a reminder about applying techniques learned elsewhere. It is not intended to be the sole source of guidance.

The use of real-time metrics tools in a team setting has beneficial effects on team dynamics and practices. Because the metric feedback is visible to other members of the team and to the project manager, most programmers, out of professional pride, devote extra effort to avoid yellow feedback in their code and go to great lengths to avoid red feedback. In team situations, this visibility creates a subtle shift from pride in speed of development to pride in code quality. Since a higher-quality work product reduces time lost to rework and bug fixing, the end result tends to be of a higher quality without slowing the pace of development. Real-time metrics also affect formal and informal cooperative work practices. The formal process of code reviews is both more effective and less burdensome because reviewers focus more on substantive review and less on style review. The self-explanatory metrics also provide a way to integrate new Smalltalk developers, especially inexperienced ones, into an ongoing team. Seasoned team members often can simply remind junior members to "keep the lights green" rather than critique their code in detail. In this way, the start-up burden that newcomers place on the rest of the team is reduced. It also tends to reduce clashes over stylistic differences and individual preferences.

Next steps. The work so far only addresses complexity at the method level. For a class-level metric to be suitable for real-time feedback, we must be able to compute the metric quickly, be able to specify trustworthy thresholds, and be able to suggest heuristics for improving classes that show excessive complexity. Efforts are in progress to define and validate class-level metrics suitable for real-time feedback. Efforts are also underway to develop real-time metrics tools for C++. In general, the issues of complexity in C++ virtual functions are similar to those within Smalltalk methods. Even more similarity may exist at the class level. Because C++ is inherently a much more complex language than Smalltalk, it has additional issues that contribute to complexity: explicit memory management (i.e., constructors and destructors), pointers, multiple inheritance, embedded objects, and non-OO constructs mixed in with OO constructs. C++ development environments are also not as well integrated as those in Smalltalk, so presentation of feedback may be more difficult. However, there is no fundamental reason why real-time feedback could not be as effective in reducing the complexity of C++ code as it is with Smalltalk. The power of real-time metrics is in emphasizing the need to reduce complexity and in creating a tighter synergy between the development system and the developer. The details of the metrics or the language do not matter as much as the details of the feedback loop: how quick it is, how unobtrusive it is, and how easy it is for the developer to use the feedback to improve the code.

Summary. Complexity is as mysterious as it is costly to software development organizations. The overall complexity of a software system can neither be comprehensively defined nor measured. Yet complexity does not spring, fully formed, into otherwise simple software. It grows within the software with the unwitting help, or at least the acquiescence, of the people who build and maintain the software. This growth can be tamed, if at all, only by constant effort on the part of all concerned: software developers, testers, maintainers, and project managers.

We argue that the best way to reduce the growth of complexity is to use metrics and tools that focus directly on the incremental addition of complexity. Real-time metric feedback makes the incremental addition of complexity more visible to all concerned and thereby helps them to reduce its rate of growth. This paper presents metrics for Smalltalk methods that can be computed quickly and presented unobtrusively. These metrics provide useful information

about method-level complexity that encourages developers to modify their behavior in desirable ways. Initial experience with the real-time metrics tools shows them to be well accepted. Developers adopt the tools willingly, if not enthusiastically.

Acknowledgments

The author would like to acknowledge Sam Adams (IBM North America Object Foundry) for many hours of discussion on the topic of software complexity in general and real-time metrics in particular. He helped to choose the appropriate real-time metrics and their cutoffs, and he developed most of the user interface and the metric evaluation framework for the initial prototype of the tool. Steve Graham (IBM North America Object Foundry) contributed many helpful comments, polished the tool for distribution, and ported it to IBM Smalltalk.

**Trademark or registered trademark of Digitalk Inc., Object Share Systems, Inc., Object Technology International, or ParcPlace Systems, Inc. (Digitalk and ParcPlace recently merged to form ParcPlace-Digitalk.)

Cited references

1. S. S. Adams and S. L. Burbeck, "Software Assets by Design," *Object Magazine* **2**, No. 4 (November-December 1992).
2. S. S. Adams, "Return on Investment: Constant Quality Management," *Hotline on Object Technology* **4**, No. 1, 4–8 (November 1992).
3. T. Bollinger, "What Can Happen When Metrics Make the Call," *IEEE Software* **12**, No. 1, 15 (January 1995).
4. M. M. Waldrop, *Complexity*, Simon & Schuster, New York (1992).
5. G. Nicolis and I. Prigogine, *Exploring Complexity: An Introduction*, W. H. Freeman, New York (1989).
6. M. Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'," *Scientific American* **223**, No. 4, 120–123 (October 1970).
7. J. Conway, E. Berlekamp, and R. Guy, *Winning Ways*, Academic Press, Inc., New York (1982).
8. S. Wolfram, "Cellular Automata as Models of Complexity," *Nature* **311**, No. 5985, 419–424 (October 1984).
9. R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* **1**, No. 2, 22–30, 35 (June/July 1988).
10. G. Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, The Benjamin/Cummings Publishing Co., Redwood City, CA (1994).
11. D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky, *Foundations of Measurement, Vol. 1*, Academic Press, Inc., New York (1971).
12. R. D. Luce, "Dimensionally Invariant Laws Correspond to Meaningful Qualitative Relations," *Philosophy of Science* **45**, 1–16 (1978).
13. J.-C. Falmagne and L. Narens, "Scales and Meaningfulness of Quantitative Laws," *Synthese* **55**, 287–325 (1983).
14. R. D. Luce and L. Narens, "Classification of Concatenation Measurement Structures According to Scale Type," *Journal of Mathematical Psychology* **29**, 1–72 (1985).
15. L. Narens, *Abstract Measurement Theory*, The MIT Press, Cambridge, MA (1985).
16. E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering* **14**, No. 9, 1357–1365 (1988).
17. H. Zuse, *Software Complexity: Measures and Methods*, DeGruyter, Amsterdam (1991).
18. S. R. Chidamber and C. F. Kemmerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* **20**, No. 6, 476–493 (June 1994).
19. N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering* **20**, No. 3, 199–206 (March 1994).
20. D. A. Gustafson and B. Prasad, "Properties of Software Measures," in *Formal Aspects of Measurement*, T. Denvir et al., Editors, Springer-Verlag, Inc., New York (1991).
21. J. Tian and M. V. Zelkowitz, "A Formal Program Complexity Model and Its Application," *Journal of Systems Software* **17**, No. 3, 253–266 (March 1992).
22. M. G. Barnes and B. R. Swim, "Inheriting Software Metrics," *Journal of Object-Oriented Programming* **6**, No. 7, 27–34 (November-December 1993).
23. C. Duff, "Designing an Efficient Language," *Byte* **11**, No. 8, 211–224 (August 1986).
24. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2**, No. 4, 308–320 (December 1976).
25. M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall Object-Oriented Series, Prentice Hall, Englewood Cliffs, NJ (1994).
26. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).

Accepted for publication January 16, 1996.

Stephen L. Burbeck *IBM North America, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: sburbeck@vnet.ibm.com)*. Dr. Burbeck is a senior consultant in the IBM North America Object Foundry group. He received his Ph.D. from the University of California at Irvine in mathematical psychology in 1979. In 1980 he became Director of Data Processing and Statistics at the Linus Pauling Institute of Science and Medicine and in 1985 was one of the founders of the company that marketed the first Smalltalk-80™ for the IBM PC-AT®. During that period he participated on the executive committee for OOPSLA-86, the first conference on object-oriented programming systems, languages, and applications. He moved to Apple Computer Corporation in 1988, where he was product manager for Apple's MacApp™ and MacSmalltalk™. In 1990 he moved to North Carolina to become Vice President of Knowledge Systems Corporation. He joined IBM in January of 1995. Dr. Burbeck adopted Smalltalk as the language of choice in 1985. Since then he has been involved with Smalltalk and object-oriented analysis and design in the capacity of programmer, designer, manager, teacher/mentor, and consultant. He has authored and coauthored papers on mathematical sociology, mathematical psychology, biochemistry/biophysics, and object technology.

Reprint Order No. G321-5602.