Cornering the Chimera

Cornering the Chimera

R. GEOFF DROMEY, *Australian Software Quality Research Institute*

"It is better not to proceed at all, than to proceed without method."
— Descartes

Concrete and useful suggestions about what constitutes quality software have always been elusive. I suggest a framework for the construction and use of practical, testable quality models for requirements, design, and implementation. Such information may be used directly to build, compare, and assess better quality software products.

Over the past decade, the term "software quality" has been widely used. In many instances the term has been loosely used in relation to process and product. This has created considerable confusion and diverted the industry from its primary goal — improving the quality of the products of the various phases of software development.

It helps to get clear at the outset that some very elusive notions — like "quality," "goodness," and "fitness-for-purpose" — are *experiential*. That is, people make a judgment, depending on their particular needs or perspective, that something they use, encounter, or examine is "good" or has "quality." Exactly what tangible properties engender such a response is something quite different. In our quest to improve software quality, we must devote much more attention to this area.

Another source of semantic confusion stems from the oft-heard advice that "quality should be built into software." This very misleading statement distracts from the real issue — how to build software that manifests high-level quality attributes. We cannot build high-level quality attributes like reliability or maintainability into software. What we can do is identify and build in a consistent, harmonious, and complete set of product properties (such as modules without side effects) that result in manifestations of reliability and maintainability. We must also link these tangible product properties to high-level quality attributes.

0740-7459/96/$05.00 © 1996 IEEE

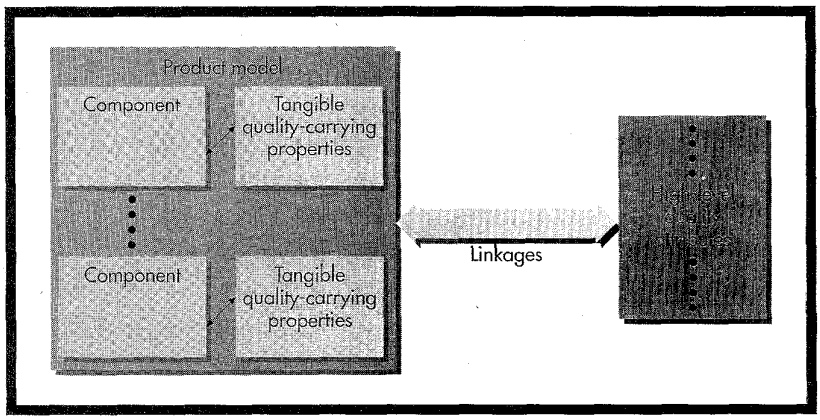


Figure 1. *Elements of a product quality model.*

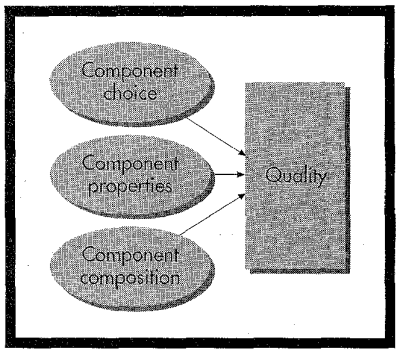


Figure 2. *Factors that determine product quality.*

While some attention has been paid to high-level quality attributes, little has been devoted to the systematic study of tangible product properties and their influence on high-level quality attributes. Today the dominant modus operandi for software development is heavily process-oriented. This rests on the widely held belief that you need a quality process to produce a quality product. The flaw in this approach is that the emphasis on process usually comes at the expense of constructing, refining, and using adequate *product* quality models. Instead, developers frequently rely on public or internal software-engineering standards that are not always helpful.[1]

The fundamental axiom of software product quality is: *a product's tangible internal characteristics or properties determine its external quality attributes.* Developers must build these internal properties into a product in order for it to exhibit the desired external quality attributes. A product quality model, therefore, must comprehensively identify the tangible (measurable and/or assessable) internal product characteristics that have the most significant effect on external quality attributes. And it must establish the dependencies between the two. Until we use better product quality models, we will not derive the full advantages of process assessment and improvement. When we are clearer about what we are trying to achieve in terms of product quality, it should be much easier to tune our processes accordingly.

Our understanding of what constitutes quality software has certainly not exhibited an unfluctuating advance toward greater truth. We can make real advances only when we understand the work of others, find it wanting, and try to overcome its weaknesses while building on its strengths.

An ultimate theory of software quality, like the chimera of the ancient Greeks, is a mythical beast of hybrid character and fanciful conception.[2] We are obliged, however, to strive to make progress, even though we realize that progress often brings a new set of problems.

There have been a few attempts to tackle the problem of product quality systematically and comprehensively.[3-6] In general, however, efforts have stalled because of

- the perceived scale of the problem,
- the diversity of quality defects in software, and
- the difficulty of factoring high-level quality attributes down to tangible properties.

We can overcome these barriers by adopting the strategy of always proceeding from the tangible and measurable to the less tangible, higher level quality attributes.

QUALITY MODEL FRAMEWORK

What sort of framework can link tangible product properties to intangible quality attributes? In tackling this problem, we must face two issues:

- many product properties appear to influence the quality of software, and
- apart from some empirical and anecdotal evidence,[7] there is little formal basis for establishing which product properties affect which high-level quality attributes.

To address these issues, we need a generic quality model and a process to build such models for different software products. Figure 1 shows the three principal elements of a generic quality model: product properties that influence quality, a set of high-level quality attributes, and a means of linking them.

Product model. Products are composed of components. Some components are atomic; others are composed of simpler components. For example, the components of a software implementation are variables, expressions, statements, and so on. Given this conceptual model, product quality is largely determined by the

- choice of components that make up the product and how they are implemented,
- tangible properties of the individual components, and
- tangible properties associated with component composition.

Figure 2 illustrates these quality requirements.

Inherent *rules-of-form* govern the use of each component type. For example, a variable must be initialized before it can be used. And *rules-of-composition* govern the way components are used in the context of other components. Violating any of these rules affects product quality. Some violations are severe enough to affect functionality, so the product cannot perform as intended. Less severe violations might not affect functionality, but will affect other high-level quality attributes such

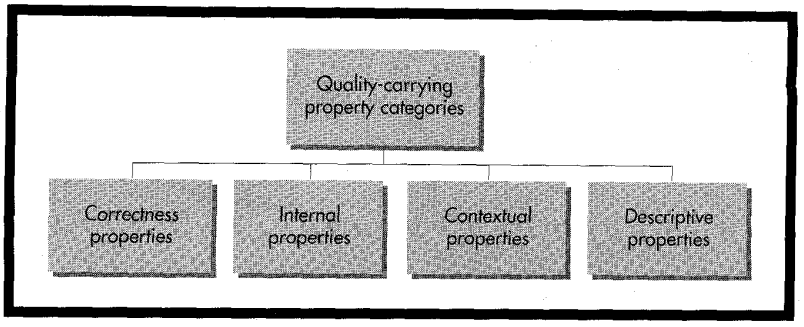


Figure 3. *Product properties that affect quality.*

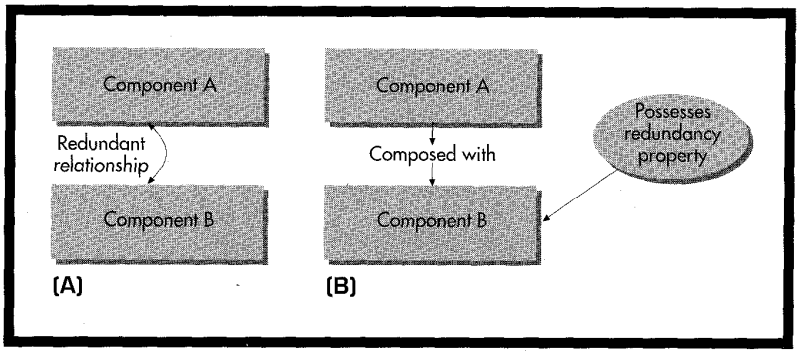


Figure 4. *Contextual properties, expressed according to* ***(A)*** *a relation between components and* ***(B)*** *an external property of a component.*

as efficiency and maintainability.

The possible sources of violation are:

- using the wrong component in a given context,
- improperly implementing a component, and
- misusing a component in relation to other components.

Now let's take a more careful look at the nature and range of properties that may be associated with components. Simply identifying large numbers of potential properties does not provide a good basis for constructing a quality model. Figure 3 shows a better basis for classifying the tangible quality-carrying properties of components.

Correctness properties. Things can go wrong either with the way components are deployed directly or in context. Some properties are so significant that if they are violated the product will not perform as intended. Such correctness properties deserve separate consideration and so are classified separately. Correctness properties may be *internal* — associated with individual components — or *contextual* — associated with the way components are used in context.

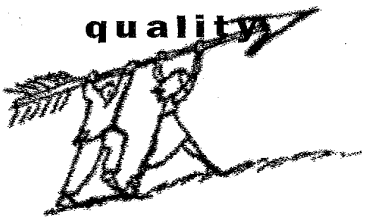
Internal properties. Every component has a *normal form* that defines its internal "truth."[8] For example, the body of a loop must always ensure progress toward termination. A component's normal form should not be violated, regardless of the context. Internal properties measure how well a component has been deployed according to its intended use or implementation requirements or how well it has been composed.

Contextual properties. How components are composed influences product quality, but how can we associate properties with the large number of possible relationships? We need to characterize *relational quality* without getting involved in a combinatorial explosion. One practical way to do this is to avoid focusing on the relations and instead to associate contextual properties with individual components. Figure 4 illustrates the difference in approach: Suppose that when component A and component B are composed, there is inherent redundancy in their relationship. Figure 4a describes the redundancy in terms of the relationship between A and B, but Figure 4b assigns the contextual property of redundancy to component B. In this way, contextual properties deal with the *external* influences by and on the use of a component. This approach significantly simplifies the task of dealing with quality problems that stem from the composition of large numbers of different types of components. Contextual properties sometimes involve correctness, but many times they involve less severe problems that affect high-level quality attributes like maintainability.

Descriptive properties. To be useful, a software product must be easy to understand and use for its intended purpose. These descriptive properties apply to requirements, designs, implementations, and user interfaces. For example, program variables should have descriptive names. Sometimes, descriptive properties play the important role of specifying functionality and constraints or identifying contexts, either formally or informally.

Quality attributes. Software quality is often discussed in terms of high-level attributes like functionality, reliability, and maintainability. Ideally, a set of high-level quality attributes should be complete, compatible, and nonoverlapping. But each high-level attribute usually depends on several product properties that are certainly not mutually exclusive in their effect on high-level quality. For example, various forms of redundancy affect both efficiency and maintainability. There is nothing we can do about this. Instead, we must ensure that the links between product properties and high-level attributes are clearly established. The important thing is to focus on those high-level attributes that describe the *priority* needs for the software. Priorities can vary from product to product and project to project.

Finally, if we accept the notion that a quality process is needed to produce a quality product, then we should demand that the product be developed



by a mature, well-defined process. In other words, we must attach a high-level attribute, "process-mature," to each product's quality model. This is a practical way to link the process to product quality.

Links. In trying to establish the links between tangible product properties and intangible quality attributes, we come up against the age-old cause-and-effect problem. Unfortunately, there is little we can do — logically or formally — to establish these links. And it is a tedious, daunting task to try to empirically verify every link for every product property. Instead, I suggest we establish links for the four product properties already identified: correctness, internal, contextual, and descriptive. On the basis of the evidence and arguments, you can judge for yourself if you accept any broader links that might be proposed.

The goal is to remove a programmer's license to apply bad practices.

Consider the correctness property and ISO-9126.[9] All correctness properties imply that a component, either directly or in context, is not implemented or does not function as intended or designed. When this is true, neither the functionality nor the reliability of the product can be guaranteed.

In other words, a prerequisite for a system to behave reliably and satisfy its functional requirements is that the tangible correctness properties of all its components are satisfied. Therefore, the correctness properties influence the quality attributes *functionality* and *reliability*.

Following this approach, you can classify product properties according to the conceptual product-model classifications I have proposed. I do not have enough space here to develop and justify all the links I will mention.

Model construction. Constructing and refining a product quality model is straightforward. It involves five steps:

1. Identify a set of high-level quality attributes for the product.
2. Identify the product components.
3. Identify and classify the most significant, tangible, quality-carrying properties for each component.
4. Propose a set of axioms for linking product properties to quality attributes.
5. Evaluate the model, identify its weaknesses, and either refine it or scrap it and start again.

In this way you can construct testable, assessable, and refineable quality models for the key products of software development: the requirements specification, the design, and the implementation. Each of these models could be the focus of a complete article; here I can describe only the key aspects of each. My intent is to illustrate *how* to develop such models. You will see that the emphasis changes as we move from one type of product to another. And, although it seems logical to consider requirements first, then design, and then implementation, I think it will be clearer if I deviate from this sequence.

IMPLEMENTATION QUALITY MODEL

The implementation is where all the poor decisions and quality problems from earlier development phases ultimately come home to roost. Not to mention the quality problems that are born in the implementation phase itself. As Nicklaus Wirth so aptly put it, "in programming, the devil hides in the detail." Here I focus on some of the detail — both the simple and more complex properties — that affect the integrity of individual components and their use in context.

In this undertaking I cannot ignore the influence of the programming language. A language can influence software quality

- *negatively*, by permitting bad practices and/or by preventing good practices;
- *neutrally*, by permitting both good and bad practices; and
- *positively*, by preventing bad practices and enforcing good practices.

Most popular languages fall into the first two categories. As a consequence, they place a heavy burden on the programmer to produce good quality software. This is hardly ideal. Programming is hard enough without asking programmers to shoulder so much responsibility. A language that influenced quality positively could potentially reduce quality defects by a tremendous amount. It is not necessary to take away the programmer's freedom; rather, the goal is to remove a programmer's license to apply bad practices.

There are three ways to do this: design better languages and compilers, implement more rigorous inspections, or build better static analyzers that implement programming standards that define effective quality models. The quality model for implementations I describe here supports these three suggested strategies.

Identify quality attributes. An effective way to identify a set of high-level quality attributes for software implementations is to ask, "what are the most important uses of this implementation?" ISO-9126 provides a good response to this question, in the form of the first six attributes and their subattributes listed in Table 1. To these, we add the attributes process-mature and reusability. As you might expect, the quality attributes in Table 1 are broad and intangible and so are of little help in building software with such attributes. To make progress, we must

TABLE 1
IMPLEMENTATION: HIGH-LEVEL QUALITY ATTRIBUTES

Attributes	Subattributes
Functionality	Suitability, accuracy, interoperability, compliance security
Reliability	Maturity, fault-tolerance, recoverability
Efficiency	Time behavior, resource behavior
Usability	Understandability, learnability, operability
Maintainability	Analyzability, changeability, stability, testability
Portability	Adaptability, installability, conformance, replaceability
Reusability	Machine-independent, separable, configurable
Process-mature	Client-oriented, well-defined, assured, effective

use the quality-model framework and proceed to identify tangible product properties that will result in these intangible quality attributes.

Identify components. It is relatively straightforward to identify an appropriate set of product components for an implementation because the language's grammar identifies them. Implementation components fit broadly into two categories: those that describe computations and those that describe data. In the first category are things like loops, if-statements, guards, assignments, and expressions. In the second category are things like variables, constants, and types. I have published a detailed list elsewhere.[10]

Identify quality-carrying properties. Arriving at a set of quality-carrying properties for each component is an empirical process. It can, however, be guided by asking a series of questions like, "Is there any property associated with this component that affects correctness?"

Consider variables, for example. Variables, fundamental components of every imperative language, possess a small, well-defined set of quality-carrying properties, shown in Figure 5. Clearly, if a variable is not assigned before it is used or if it is not of the appropriate precision, correctness could be affected. Using a variable for more than one purpose in a given module certainly threatens correctness and makes a module less descriptive. Declaring then failing to use a variable and using a global variable in a module both represent contextual problems that affect quality. Finally, failing to give a variable a descriptive name or failing to document the purpose of a variable affects its descriptive properties. I can't guarantee that this list is exhaustive, but experienced practitioners have failed to uncover additional properties.

To ensure that variables have no negative impact on quality when you implement programs, you should therefore ensure that all of these tangible component properties are satisfied.

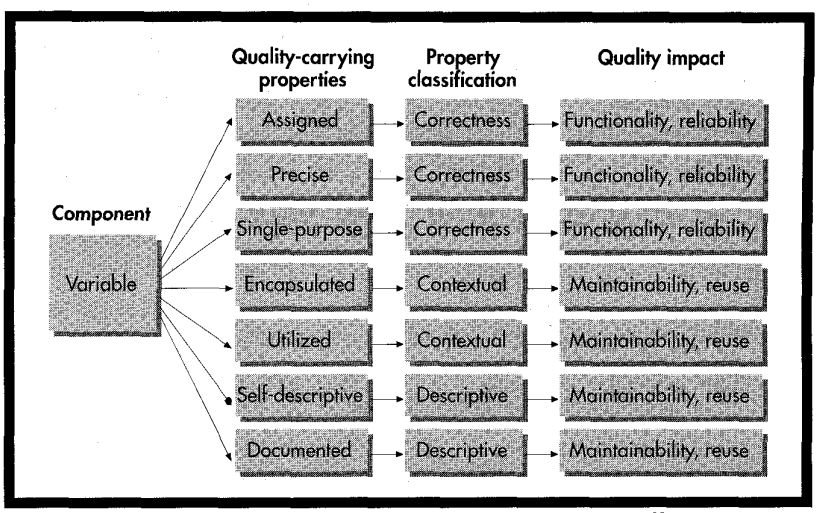


Figure 5. *Product properties of a variable component and their effect on quality.*

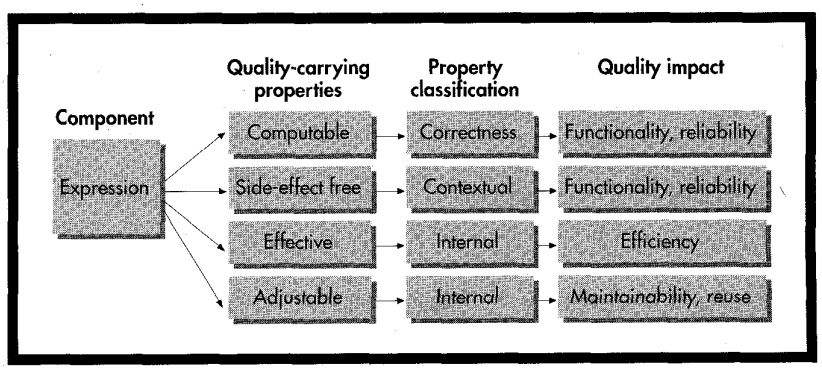


Figure 6. *Product properties of an expression component and their effect on quality.*

In a similar way, you can associate a set of quality-carrying properties with *each* of the other components used by a particular language. Expressions, like variables, are fundamental components of all imperative programming languages. They also possess a small, well-defined set of quality-carrying properties. Figure 6 shows these properties.

What these properties tell us is that expressions should not have a structure that risks division by zero, or taking the square root of a negative number,

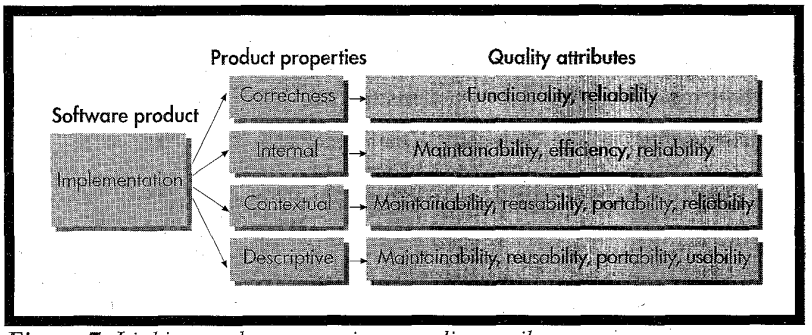


Figure 7. Linking product properties to quality attributes.

TABLE 2
REQUIREMENTS: HIGH-LEVEL QUALITY ATTRIBUTES

Attributes	Subattributes
Accurate	Conformant, functional, valid, constrained
Understandable	Motivated, coherent, self-contained
Implementable	Achievable, viable, total-solution
Adaptable	Modifiable, extensible, reusable
Process-mature	Client-oriented, well-defined, assured, effective

and so on. An expression should have no side-effects (compare the use of expressions in C), contain no unnecessary arithmetic or logical computations, and contain defined constants rather than mystery numbers.

The result of this exercise is something tangible that programmers and others can use to produce quality software. Violation of any of these properties represents a *quality defect* that can be detected.

How do you know when you have identified all the relevant properties for variables? The answer is simple: You don't! However, this is not as bad as it seems. When you identify the need for a new quality-carrying property, you simply add it to the model, thereby refining and strengthening it.

Link properties to attributes. Now we must link product properties to high-level quality attributes. Earlier, I reasoned that correctness properties affect the quality attributes' functionality and reliability. Using the same reasoning, I can generate the links in Figure 7. For example, redundancy in the body of loops affects both maintainability and efficiency and, to a lesser extent, reliability.

Linking product properties to the quality attributes is neither unique nor absolute. I claim only that this is a sufficient set that provides some (but not complete) guidance on assessing the high-level quality impact of violations of various tangible product properties. These links do, however, let us deal with both high- and low-level design issues, consistency and completeness issues, and the characterization of functionality. In choosing the links, I deliberately tried to minimize overlap.

REQUIREMENTS QUALITY MODEL

Much has been written about the difficulties in obtaining and formulating requirements. I won't summarize those issues here. Instead, I describe key aspects of a quality model for requirements, focusing mainly on requirements rather than a complete requirements specification.

Identify quality attributes. To identify high-level quality attributes of a requirements specification, ask "What do I want to do with this specification?" Principally, you want to

- *use* it to describe a problem's requirements;
- *use* it as the basis for design;
- *use* it as the instrument of contract and common understanding among the client, the users, and the developer;
- *change* it to meet new or modified requirements; and
- *reuse* or *adapt* it to help solve another problem.

To use it as a specification, the client must be able to understand it and have confidence that it describes what he requires. To use it as the basis for design, the designer must be able to understand it and have confidence that it contains all the relevant information. Because requirements often change over the life of a project, it must accommodate change. So the resulting quality model must meet various needs of different parties.

These might appear to be the same needs identified for an implementation, but there is quite a difference in emphasis. ISO-9126, for example, buries the attribute understandability in maintainability, as the subattribute analyzability. Yet understandability is much more important in requirements than, say, functionality or reliability. I suggest the quality attributes in Table 2 do a better job of capturing the high-level quality needs and expectations of requirements than ISO-9126.

Identify components. I think the root cause of many problems with requirements is confusion over exactly what requirements are and what form they should take. I also think that the form you choose can have a significant effect on the quality of your requirements. Requirements start out as an idea about a perceived need. How users express this need varies greatly, depending on the scale of the problem. Requirements must evolve from this idea — which is sometimes very vague and informal — to something that is tangible, accurate, controllable, verifiable, and implementable.

There are many opinions about how best to specify requirements, and clearly no representation is appropriate for every system. I am not going to join the debate about whether requirements should be formal or not, or whether they should be specified in Z

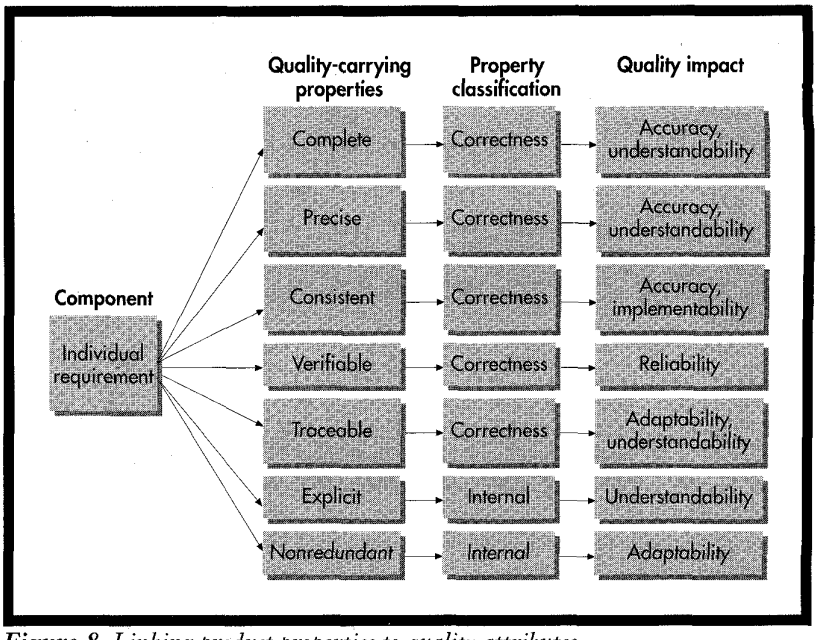


Figure 8. *Linking product properties to quality attributes.*

or VDM. I am interested in the fundamental underlying form that requirements should exhibit, regardless of representation. To write good requirements, you must be able to see beyond representations to the underlying form.

At the highest level there are only two underlying types of requirements: functional and nonfunctional. Both may be most simply specified using just *variables* and *constraints*.

Functional requirements. It is very tempting to specify functional requirements with the phrase, "we want a system that does . . ." This may be appropriate for users, but it is a trap that developers should not fall into. There are no good tools that can accurately express actions or transformations. So expressing requirements this way risks misinterpretation, ambiguity, omission, and inconsistency.

A better way is to place constraints on outputs or express relations between inputs and outputs. After all, it is the result a system produces, not the function it performs, that ultimately is important.

At its most basic level, a function accepts a set of inputs and uses or transforms them to produce a set of outputs. To specify functionality, first identify all the desired outputs and all the inputs needed to produce them. One very simple representation is:

```
Output_Variable_List
 :=  Function_Name
  (Input_Variable_List)
```

This captures the dataflow and may be directly translated into a dataflow diagram.

Next, specify all the constraints on the inputs, all the constraints on the outputs, and the relations between the inputs and the outputs. Constraints may be either simple or complex. They may either be a *property* of an individual variable or they may express *relations* among sets of variables. A relation identifies a condition that can be either true or false. A simple constraint involves one or more variables and a relational operator (for example, a < b). Logical connectives (and, or, not) and quantifiers (for all) are used to build more complex constraints. The important thing is to identify all variables and characterize relations accurately, using connectives and quantifiers.

Another good way to express the highest level of functionality is to design the user interface as part of the requirements process. A user interface and user manual give users the clearest picture of how their requests will be realized. An architect would not think of going ahead with a building until the owner is happy with the perspective plans. Perhaps there is a lesson here for software engineers.

Nonfunctional requirements. Nonfunctional requirements can vary widely. Again, it is important to turn these requirements into something tangible. Like functional requirements, nonfunctional requirements must be clear-cut and completely verifiable. A nonfunctional requirement is a constraint, such as "the software must conform to the Company Programming Standard XXX." In this case, the variables are software and programming standard and the relation is conform. The constraint can be judged to be either true or false. What is important is that the variables and the relations are well-defined.

The discussion so far suggests the following components:

- requirements set,
- individual requirements,
- constraints,
- variables,
- constants, and
- relations.

Identify quality-carrying properties. Establishing the quality-carrying properties of requirements components is similar to establishing them for implementation components. Individual requirements are the key components of a requirements specification. In assigning a set of quality-carrying properties to a requirement, you must be careful to avoid assigning properties that rightly belong to the components of a requirement (such as the variables and constraints). Most of the quality-carrying properties of a requirement relate to relationships among the requirement's components. The first property we will consider relates to the form of requirements: All the inputs and outputs associated with a requirement must be included. If they are not, the requirement is incomplete.

A second quality-carrying property of a requirement is that the set of constraints associated with it must be compatible and therefore consistent. In order for a requirement to accurately

TABLE 3
DESIGN: HIGH-LEVEL QUALITY ATTRIBUTES

Attributes	Subattributes
Accurate	Conformant, functional, valid, constrained
Effective	Resource-efficient, rational
Understandable	Motivated, coherent, self-contained
Adaptable	Modifiable, extensible, reusable
Process-mature	Effective, well-defined, assured

capture what is required it must also be explicit, precise, and nonredundant.

Finally, it is important that we can easily trace a requirement back to the user requirements and verify whether or not each requirement is satisfied. For a requirement to be verifiable it must be possible to use a defined procedure to establish whether or not it is satisfied.

Figure 8 shows the quality-carrying properties of individual requirements.

Link properties to attributes. Using reasoning similar to that which I used for the implementation example, I generated the links in Figure 8.

DESIGN QUALITY MODEL

Programmers' desire to begin coding as soon as possible has meant that design has not traditionally received the attention it deserves. As a consequence, our understanding of how to do design, what the real purpose of design is, what constitutes good design, and what to measure to assess design quality is still relatively immature. Constructing a quality model for design is therefore less straightforward. The existence of so many different design methods suggests that there is not even any general consensus about what the *components* of a design are and what is an appropriate *representation*. In order to construct a generic quality model for design, we must try to address some of these fundamental questions.

To create a design, ideally we begin with a set of functional and a set of nonfunctional requirements that describe *what* behaviors and *what* characteristics the implemented system must exhibit. The design must show *how* each of these requirements is to be realized in the context of the overall system.

To do this for functional requirements, there are really only two alternatives: *decomposition* based on the identification of smaller, simpler functional components and *composition* based on object functionality. A mix of both is often the most appropriate for creating loosely coupled systems. In using these design strategies we must establish the relationships among the functional requirements in the system, the most desirable situation always being to keep functional requirements as nearly independent as possible.

Taking all this into account, a sufficiently detailed list of what a design does includes:

- satisfy the requirements and be easily traceable back to those requirements,
- provide a basis for implementation and facilitate this transition,
- provide a framework for implementing functionality correctly and effectively encapsulating data,
- control complexity of functionality and data at all levels,
- lead to the production of quality software,
- express computations using components,
- manage the dataflow among components,
- incorporate components from other applications (reuse),
- be easy to modify, extend, and verify,
- localize the effects of change and runtime problems,
- produce software that is easy to understand at several levels,
- cut problems at their joints and thereby satisfy the principle of correspondence,[11] and
- recognize and solve specific problems using known architectures.

Identify quality attributes. Neither the ISO-9126 quality-attribute set nor our set for requirements is appropriate for design. However, there are similarities. A design must accurately satisfy requirements and be understandable and adaptable. And a design should be developed using a mature process. A design's quality, however, is distinguished by its effectiveness in solving the problem at hand. An appropriate set of quality attributes is shown in Table 3.

Identify components. The choice of components for design is not clear-cut. There are a great variety of tools and graphical techniques to represent high-level and detailed designs. But all these tools and techniques do the same thing: they identify, characterize, express, encapsulate, and compose functionality and data. When all the syntactic sugar is stripped away, the fundamental building blocks and glue of a design are:

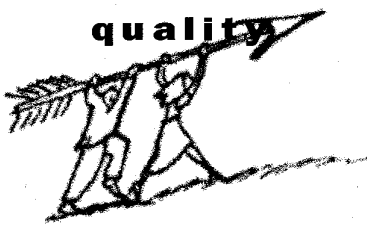
- modules (of various types),
- variables (inputs and outputs),
- pre- and postconditions, and
- various means of composition.

Of these, modules — the actual components you choose to use — have the most effect on design quality. Therefore, I will focus here on modules, but first I want to note that the *source* and *sink* properties of variables are critical quality-carrying properties in understanding how a complex system fits together.

In the case of modules, the great challenge in constructing a quality model is to decide, irrespective of representation,

- what types of modules the system needs,
- what properties each type of module should exhibit,
- what components should compose each type of module, and
- how these components should be composed.

In a single word, the fundamental issue is *architecture*. Just as the study and practice of architecture has made a great contribution to the quality of buildings, so too will the study and



practice of software architecture make an important contribution to the quality of software design. We are only now entering the era of software architecture.[12]

We must explore ways to substantially simplify the architecture of software and the accompanying development process. Conventional software is complex because its form is seemingly amorphous and because it is built from a juxtaposition and intertwining of many structures.

Is it really necessary to jumble together so many different structures? If we could reduce the variety of structures, we would have a much better chance of reducing software complexity and at the same time simplifying and clarifying the design process. Fundamentally, there are four ways to reduce complexity:

- reduce the variety of structures that may be composed,
- simplify the way structures may be composed,
- introduce a layering architecture that limits the types of structures that may be composed in a given layer, and
- introduce coupling and cohesion criteria, to separate computations that have no dependence at the level at which they are performed.

All four of these strategies have been successfully exploited by microelectronic engineering to control and reduce the complexity of VLSI designs. Microelectronics, in fact, has successfully used six broadly categorized levels of abstraction (architectural, behavioral, functional, gate, circuit, and layout) to control design complexity and structure. This success suggests that we should seriously consider layering in software design.

By layering, I have in mind something quite different from traditional top-down design. In my model, objects, programs, processes, and systems are all layered. The constructive principle is, *components constructed at one level may only be used at the next level up*. This principle also means that a component cannot be used to construct other components in its own layer. I suggest the following layered components be used as the basic tools for constructing designs. As in requirements, all the module types that follow have the same underlying form: a list of outputs, a name, and a list of inputs, such as:

```
Out1, Out2, . . ., Outm
  := ModuleName
   (In1, In2, . . ., Inn)
```

Base layer. Each abstract data type consists of a set of primary functions in the base layer that are not separately compilable. Each of these functions uses no other primary function of the abstract data type. Assignments and other state-changing statements are composed using SSIR (sequence, selection, iteration, and recursion).

Program layer. A program here is quite different from a conventional program. A program is composed solely of functions from the base layer; it contains absolutely no assignments. All state changes are achieved by function output. All functions are composed using SSIR. Programs are separately compilable and executable and are well-suited for expressing the high-level aspects of a design.

Removing assignments from programs, which we call *computation hiding*, greatly simplifies their form. And, because it is not possible to declare any new type in a program, reuse is encouraged.

Process layer. If all the functionality associated with an individual object cannot be captured using functions and programs, a process layer is added. Processes are composed solely from the individual object's programs using SSIR. Again, because there are no assignments, processes have a very simple form that is well-suited for expressing high-level aspects of a design. It is also possible to have higher level processes.

Systems layer. It is sometimes necessary to reactively compose programs of an individual object. Systems compose reactive programs and processes. Systems are specified in a purely declarative way. All the designer needs to do is declare all system variables, identify which variables are input to each program, and identify which variables are output from each program. The default is that each program executes when all the programs that supply its inputs have terminated. A finite-state machine that looks after the I/O dependencies may be used to control system execution.[13] The system model is powerful enough to accommodate various architectures such as pipes, filters, and client-server.

For example, part of a system specification is

If we could reduce the variety of structures, we would have a much better chance of reducing software complexity.

```
b, d, e, f := ReactiveSystem( . . . )
 .  .  .
 a, b := prog1(x, y, z),
 c := prog2(u, v),
 d, e := prog3(a, c),
 f := prog4(w)
 .  .  .
end_ReactiveSystem
```

Program outputs are on the left side of the assignment operator. In this case, `prog3`, which depends on `a` and `c` for input, can execute only when `prog1` and `prog2` have terminated; `prog4` starts executing concurrently with `prog1` and `prog2`. *All functions, programs, processes, and systems use the same form to specify inputs and outputs.*

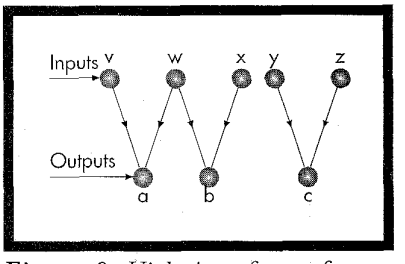


Figure 9. *Violation of transformational cohesion.*

Composite modules. Programs, processes, and systems can have more than one object as input. Such components are not encapsulated within an object. However, you can use the same layering principles as for object programs. Because it is not possible to declare new types other than in abstract data types (in the base layer), the layered architecture enforces and facilitates reuse from the lowest to the highest levels. Above the primary system and process layers, you might have secondary and tertiary layers, each constructed on exactly the same architectural principles. This architecture, coupled with strict rules for module cohesion based on I/O dependence, is a very powerful way to control the complexity of software at all levels. This framework is also well-suited for supporting design in a transparent way from the highest level down to the detailed implementation based on objects.

Identify quality-carrying properties. Modules are the highest level components in a software system. As such, they are always built from simpler components. Consequently, many of the quality issues that arise have to do with the quality of the components rather than the module itself. Having said that, modules do possess three key generic quality-carrying properties:

- cohesion (internal),
- coupling (contextual), and
- layering (internal).

We have already considered layering. Cohesion and coupling are widely known and often referred to, but they are rarely defined or used in a way that can make an important contribution to design quality. If defined constructively, these properties can play a key role in controlling complexity.

Cohesion. The concept of cohesion is often loosely or implicitly defined and therefore poorly understood. A module is made more complex when it includes two or more independent functions. A function's I/O dependencies provide a basis for defining cohesion. Two output variables are cohesive if they depend on at least one common input. A set of outputs is maximally cohesive if the addition of any other output to the set does not extend the set of inputs on which the existing outputs depend and there is no other output that depends on any of the input set. A function whose outputs all belong to a single cohesive set satisfies the principle of transformational cohesion.

This concept is easiest to illustrate with a directed bipartite graph. Consider the module:

```
a, b, c := Name(v, w, x, y, z)
  . . .
  a := v + w;
  b := w*x;
  c := y/z

end_Name
```

Figure 9 illustrates the situation: Although all the outputs depend on all the inputs, the output pair (a,b) has no dependence on the input pair (y,z). Hence the inputs y and z and the output c should be in a separate function. If you need more than one bipartite graph to describe a function's I/O dependencies, then it is not transformationally cohesive and should be split.

Another useful principle is *separation of concerns*, which requires that modules

- have no side effects,
- employ no global variables,
- have at least one output variable, and
- contain only variables that are each used for a single purpose.

Coupling. Modern languages and programming practices have reduced the problem of coupling. The issue here is to ensure that modules do not use global variables or exhibit side effects. Data coupling fulfills these requirements.[14] A module is data coupled when the only information passed to it is via a parameter list and none of this information is control information. It is best if the language enforces these requirements.

Link properties to attributes. These components and properties provide a reference against which you can assess the design quality of any individual module at any level in a system, including the top level.

Cohesion has a major effect on a design's effectiveness, understandability, and adaptability. It measures how well complexity has been controlled, how well a solution has been composed, and, by default, the integrity of the functionality that has been isolated in a given module (important for reuse and adaptability). It is relatively straightforward to implement an analyzer to detect cohesion problems like those described here and to generate corresponding metrics.

Coupling affects the design's understandability and adaptability.

This conceptual framework is sufficient to support a purely object-oriented approach to design but it also recognizes loosely coupled components and subsystems that may be composed either conventionally or reactively. Many designs fit naturally into these broader, richer architectural frameworks.

To assess an overall design, you should assess modules at various levels for their cohesion, their coupling, and the extent to which they exploit layering to control complexity. For modules at the lowest levels it is also necessary to examine other measures

of control-structure complexity[15] to assess detailed design quality.

For example, suppose you have a typical C module that consists of a considerable number of assignments, a number of procedure and function calls, and a number of global variables, all composed using sequence, selection, and iteration. From the mix of components alone, you can tell immediately that there has been no systematic attempt at layering, at systematic functional decomposition, or at composing object functionality. The presence of global variables also indicates coupling problems. This module's implementation-level flaws will show up in any corresponding design document, most likely in the form of serious gaps in the description of the design.

Such a module has probably been hacked together without any serious attempt at design. With a set of coupling, cohesion, and layering metrics, you can quickly characterize its design quality and begin to convert it into a form that conforms to the ideal suggested.

Useful suggestions about quality, when they are brought to our attention, usually strike us at once as familiar and revelatory. We see them as sensible, reflecting what we have felt but perhaps not expressed. What I have proposed here is not identical to quality, nor is it a substitute for what people experience as quality. What I have strived to do is create a number of things that I hope get close to capturing and conveying the same idea.

The proposed quality models demonstrate, by way of example, that it is possible to create a framework that you can use in a practical way, both to build better products and to assess and assure their quality. Over time, the details will evolve and be refined. In trying to understand ideas like quality and its relationship to other knowledge, we sometimes end up with false explanations, substitutions, and proposals. These can have a detrimental effect on the state of practice. On the subject of software quality, we must therefore always employ careful thinking and constant vigilance to avoid casual confusion.

I do not pretend to have presented a satisfactory "solution" here, but I certainly have learned from the experience and have attained an undiminished appetite to join others in trying to herd the mythical beast of software quality into a tighter corner. ◆

REFERENCES

1. S.L. Pfleeger, N. Fenton, and S. Page, "Evaluating Software Engineering Standards," *Computer*, Sept. 1994, pp. 71-79.
2. E.F. Carritt, *The Theory of Beauty*, Metheun, London, 1962.
3. B.W. Boehm et al., *Characteristics of Software Quality*, North-Holland, New York, 1978.
4. B. Kitchenham, "Towards a Constructive Quality Model," *Software Eng. J.*, July 1987, pp. 105-112.
5. B.W. Kernighan and P.J. Plaugher, *The Elements of Programming Style*, McGraw-Hill, New York, 1974.
6. M. Deutsch and R. Willis, *Software Quality Engineering*, Prentice-Hall, Englewood Cliffs, N.J., 1988.
7. M.M. Pickard and B.D. Carter, "Maintainability: What Is It and How Do We Measure It?" *Software Eng. Notes*, July 1993, pp. A36-39.
8. S. Pan and R.G. Dromey, "Beyond Structured Programming," *Proc. Int'l Conf. on Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1996, to appear.
9. *Software Product Evaluation - Quality Characteristics and Guidelines for Their Use*, ISO/IEC Standard, ISO-9126, Int'l Organization for Standardization, Geneva, 1991.
10. R.G. Dromey, "A Model for Software Product Quality," *IEEE Trans. Software Eng.*, Feb. 1995, pp. 146-162.
11. M. Jackson, *Principles of Program Design*, Academic Press, London, 1976.
12. D. Garlan and M. Shaw, "An Introduction to Software Architecture," Tech. Report CMU/SEI-94-TR-21, Software Eng. Institute, Carnegie Mellon, Pittsburgh,1994.
13. T. Lightfoot, *A Computational Model for Reactive Systems*, Griffith University, honours thesis, 1995.
14. G. Myers, *Software Reliability: Principles and Practices*, John Wiley & Sons, New York, 1976.
15. S. Pan and R.G. Dromey, "Reengineering Loops," *Computer J.*, to appear.

R. Geoff Dromey is the Foundation Professor at the School of Computing and Information Technology at Griffith University. He is the founder of the Software Quality Institute, through which he has worked closely with industrial, national, and international standards bodies and governments. He has worked at Stanford University, the Australian National University, and Wollongong University. His current research interests are applying formal and empirical methods to improve software quality and software-development productivity. He has authored or coauthored two books and more than 50 papers. He serves on the editorial boards of four journals.

Dromey received a PhD in chemical physics from LaTrobe University. He is a member of the IEEE and ACM.

Address questions about this article to Dromey at Australian Software Quality Research Institute, Faculty of Science and Technology, Nathan Campus, Kessels Rd. Brisbane, Griffith University, Queensland 4111 Australia; g.dromey@cit.gu.edu.au