A Model for Software Product Quality

A Model for Software Product Quality

R. Geoff Dromey

Abstract— **A model for software product quality is defined. It has been formulated by associating a set of quality-carrying properties with each of the structural forms that are used to define the statements and statement components of a programming language. These quality-carrying properties are in turn linked to the high-level quality attributes of the International Standard for Software Product Evaluation ISO-9126. The model supports building quality into software, definition of language-specific coding standards, systematically classifying quality defects, and the development of automated code auditors for detecting defects in software.**

Index Terms—**Software quality, product evaluation, ISO-9126, code auditing, quality defect classification, quality model, quality attributes, software characteristics, maintainability, quality-carrying properties.**

I. Introduction

SIGNIFICANT gains in the quality of software will not take place until there is a comprehensive model of software product quality available. Several different models of software product quality have been proposed [1]–[6]. While these models offer interesting insights into various aspects of software quality they have not been strong enough to stimulate significant gains in the quality of software or to gain wide acceptance.

Most recently the international standard ISO-9126 Software Product Evaluation Characteristics (1991) [7] has been put forward as a high-level framework for characterizing software product quality. This standard appears to have drawn considerably on the model originally proposed by Boehm *et al.* [1]. While this standard can provide high-level guidance it does not go nearly far enough to support building quality into software.

What must be recognized in any attempt to build a quality model is that software does not directly manifest quality attributes. Instead, it exhibits product characteristics that *imply* or contribute to quality attributes and other characteristics (product defects) that detract from the quality attributes of a product. Most models of software quality fail to deal with the *product characteristics* side of the problem adequately and they also fail to make the direct links between quality attributes and corresponding product characteristics. We will address these two issues. Our focus will be on the primary software product, the code or implementation. However, the framework we will provide may be equally well applied to other components of software products such as requirements specifications and user-interfaces. To support this claim we will also sketch part of a quality model for a requirements specification.

There is a wealth of knowledge about software quality available. The greatest challenge in proposing any model for software product quality is to find a framework that can accommodate this knowledge in a way that is constructive, refinable, and intellectually manageable. The prime requirement of any such model is that it makes clear and direct links between high-level *quality attributes* and explicit *product characteristics* at all levels. Beyond this the model must provide:

- systematic guidance for building quality into software,
- a means to systematically identify/classify software characteristics and quality defects, and
- a structure that is understandable at a number of levels, refinable and adaptable

A. Framework for a Model of Software Product Quality

A common approach to formulating a model for software product quality is to first identify a small set of high-level quality attributes and then, in a top-down fashion decompose these attributes into sets of subordinate attributes. The Software Product Evaluation Standard, ISO-9126 is typical of this approach. For example, it decomposes maintainability into the four attributes *analyzability, stability, testability* and *modifiability*. While this provides some indication of what maintainability is about the subordinate terms are still very vague and of little assistance in building quality into software. Seeking even further direct decomposition of such vague attributes is not the best way forward. Instead, it is better to employ a model that places only a *single level* (a set of quality-carrying properties) between the high-level quality attributes and the components of a product. For complex applications like software we will show that such an approach is both simpler and much more powerful. It allows us to approach the task of building a model for software product quality in a systematic and structured way by proceeding from the tangible to the intangible. This is a practical strategy for dealing with concepts as elusive and complex as quality.

Elsewhere we have described a generic model along these lines that supports building quality into products and processes [8]. This quality model consists of three primary entities: *a set of components, a set of quality-carrying properties of components, and a set of high-level quality attributes*. There are at most six binary relations among these entities. The following diagram illustrates the potential relations that must be considered to build quality into designs.

For building quality into a product or process only four of these relations are important (i.e., the ones with solid arrowheads). The model supports the examination of quality

Manuscript received January 1, 1994; revised October 10, 1994. Recommended by R. Jeffery.

R. G. Dromey is with Software Quality Institute, Griffith University, Nathan, Brisbane QLD 4111, Australia.

IEEE Log Number 9407725.

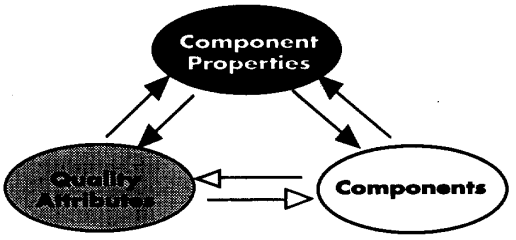


Fig. 1. Generic quality model.

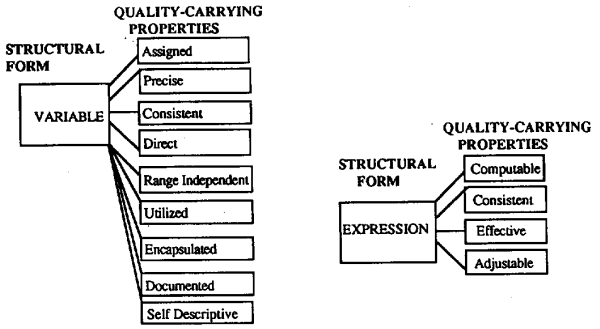


Fig. 2. Quality-carrying properties of variables and expressions.

from two important perspectives. Firstly building in quality from the *bottom-up*. That is, for each component we can identify which quality-carrying properties are important to satisfy and which high-level quality attributes each of these properties impacts. In defining this perspective the quality-carrying properties serve as the intermediaries that link entities to high-level quality attributes. It is also possible to employ this model to look at building in quality from the *top-down*. That is, for each high-level quality attribute we can identify which quality-carrying properties imply that attribute and which product entities possess particular quality-carrying properties. Applying this scheme we can define the scope of the task of building each high-level quality attribute into software products. In adapting this model to software we will replace the term component by *structural form* and we will focus upon the four primary constructive directed relations that may be used to assist building quality into software. The first two of these relations are:

- structural form → quality-carrying property relation
- quality-carrying property → quality attribute relation

Together these two relations allow us to view the task of building high-level quality attributes into software from the bottom-up by ensuring that particular product properties are satisfied. This perspective is most useful to those (that is, the programmers) with the responsibility of implementing quality software. In fact a programming standard can be usefully structured along these lines. The other two relations that are useful are:

- quality attribute → quality-carrying property relation
- quality-carrying property → structural form relation

These two relations allow us to view the task of building high-level quality attributes into software from the top-down by identifying which properties need to be satisfied for each structural form in order to build in a given high-level quality attribute. This perspective is most useful to designers who have the responsibility for specifying and factoring high-level quality attributes into the design of software. Only when we have access to both these perspectives are we in a position to understand what must be done to build quality into software. This bottom-up/top-down model can provide the concrete advice that is so vitally important to implement the process of building quality into software.

In formulating a model for software product quality based upon the generic model we have just sketched we will exploit the property that programs are constructed only using structural forms (that is, the statement types and the statement components of the implementation language). A basic characteristic of *structural forms* is that they possess, or alternatively, they may each be assigned a set of *quality-carrying properties*. The form-property model can facilitate building quality into software, detecting and classifying quality defects in software and the creation of a framework that is refinable and understandable at a number of levels. A focus on quality defects also makes a positive, if indirect, contribution to building quality into software by telling us what not to do. In considering software the two principal categories of structural forms are:

- computational forms—that describe processes
- representational forms—that describe data

We will first use two simple examples to illustrate the process of applying the form-property model and then proceed to describe the model in more detail.

B. Examples

Consider the fundamental structural forms: *variables* and *expressions*. To be free of quality defects they should possess a number of defined properties. For the moment we will simply list the *quality-carrying* properties of variables and expressions.

To *build-quality-into* variables and expressions when we implement programs we should therefore ensure that all of the above properties are satisfied. In a similar way we can associate a set of quality-carrying properties with each of the other structural forms used in programs for a particular programming language. The syntax of a language identifies all its structural forms.

Violation of any of the quality-carrying properties of a structural form results in a *quality defect* which affects its integrity. Our use of the term *defect* here does not necessarily imply that its presence will cause the functionality or reliability of a software system is to be affected. In some circumstances only nonfunctional properties are impacted by quality defects. In other circumstances defects may point to functionality and reliability problems. Some defects that violate the quality-properties of expressions are:

- uncomputable (e.g., divide by zero)
- inconsistent (e.g., contains side-effects)
- ineffective (e.g., contains computational redundancy)
- unadjustable (e.g., contains numbers)

All of the above types of defects detract from the quality of expressions. Therefore to build quality into expressions when we implement programs they should all be avoided. There are other problems that can arise with expressions. For example, an expression may contain an *unassigned* variable. However, because of the precedence rule used in the classification discipline we impose (see below) this defect will be classified as a *variable integrity* defect rather than as an expression integrity defect even though it impacts the computation of the expression. The classification discipline employed to characterize defects fits our normal intuition. It focuses on the *source of the problem* rather than on the *consequences* of the defect. In the previous example there is no defect directly with the expression. Rather the problem resides elsewhere—the variable should have been assigned before being used in the expression. A crucial requirement in developing such a framework for categorizing quality defects is that the process is *repeatable*. That is, two people confronted with the same defect should arrive at the same classification for that defect. The present proposal sets out to achieve repeatability. In addition, with the proposed model many quality defects can be detected by automatic means. Elsewhere we have described a system for doing this [9].

The conceptual groundwork for constructing a model of software product quality has now been outlined. In the remainder of this paper we will seek to flesh out the model by defining the two important quality perspectives for building in quality by specifying their four supporting relations. To do this we will first examine the various structural forms associated with processes and data for the imperative paradigm. The proposed framework can be extended to handle other programming paradigms, 4GLs, user interfaces and other software product components such as software requirements specifications, etc.

II. Structural Forms for Processes and Data

To be entirely accurate it is necessary to focus on the structural forms for a particular language. The reason for this is because we find differences like the assignment being an expression in C but a statement in Pascal. Even with this difference it is possible to adopt a style of programming in C where assignments are only used in statements. Putting these sort of difficulties aside, a common set of structural forms for processes within the imperative paradigm, listed in order from the highest level structures to the lowest level structures (increasing numerical order) is given below. The set of structural forms for data (which are ranked below "expressions") within the imperative paradigm, are also ranked from the highest to the lowest structural level, (again in increasing numerical order) are also listed below.

A. Classification Discipline

In classifying quality defects we should always associate them with the lowest level structural form to which they apply.

As an example, if a defect can be associated with either an assignment or an expression we should classify it as an expression defect because of its lower structural level. We will have more to say about this later. To apply this model to a particular language it will be necessary to make adjustments to accommodate the differences associated with structural forms. Our intent here is not to completely characterize any particular language but rather to describe the whole process in enough detail so that it can be repeated or adapted as necessary for any particular language, programming environment or other application context.

TABLE I
Structural Forms for Processes and Data

Level	PROCESS - Structural Forms	Level	DATA - Structural Forms
Highest		Highest	
1	System (set of programs)	12	Records
2	Library (set of reusable ADTs, functions and procedures)	13	Variables
3	Meta-program (e.g. shell script using program I/O)	14	Constants
4	Program	15	Types
5	User-interface	Lowest	
6	Objects (ADT)		
7	Module (encompasses functions and procedures)		
8	Sequence		
9	Statement - > (note precedence in sub-category)		
9.1	Loop		
9.2	Selection		
9.3	Function/procedure call		
9.4	Assignment		
10	Guard		
11	Expression		
Lowest			

III. High-Level Quality Attributes of Software

The quality of software is most often discussed in terms of high-level attributes such as functionality, reliability and maintainability, etc. Ideally, any such choice of high-level quality attributes should be complete, compatible and nonoverlapping. For software this turns out to be a difficult task. Each high-level quality attribute depends on a number of low-level quality-carrying product characteristics that are certainly not mutually exclusive in their high-level quality impact. For example, various forms of redundancy in software affect both its efficiency and its maintainability. Similarly, correctness properties affect both reliability and functionality. There is not much we can do about this overlap problem. Instead we must ensure that the links between low-level quality-carrying product characteristics and high-level quality attributes are clearly established. In addition, we must satisfy ourselves that the high-level quality attributes we choose adequately describe the high-level needs we have for software.

The International Standard ISO-9126 Software Product Evaluation which is built on six quality attributes (*functionality, reliability, usability, efficiency, maintainability, portability*) represents one such attempt that appears to have gained wide acceptance and consensus. Because of the status of this model as an international standard we have chosen to link our model for software product characteristics to it. There is however one serious omission with this standard. It does not emphasize the *reusability* of software. We conjecture that reusability is an important high-level quality attribute of software, which, because of its impact on productivity and quality, deserves a similar status to the other high-level attributes in ISO-9126.

If we accept this minor augmentation of the ISO-9126 model then we must define what we mean by reusability and factor it into the model. A structural form is *reusable* if it uses standard language features, it contains no machine dependencies and it implements a single, well-defined, encapsulated and precisely specified function whose computations are all fully adjustable and use no global variables or side-

effects. All ranges associated with computations and data structures in a reusable module should have both their lower and upper bounds parameterized. Also no variable should be assigned to a number or any other fixed constant and all constants used should be declared. Some might argue that reuability is already covered by maintainability and portability. However, it is strongly dependent on a distinctive subset of modularity, structural and descriptive properties. This suggests that reusability is deserving of separate recognition. There is also one other strong reason for giving reuse the status of a high-level quality attribute. It will encourage those responsible for software development to pay more serious attention to constructing software that is reusable.

IV. Quality-Carrying Properties of Software

In broadest terms the properties associated with structural forms that impact the quality of software involve two fundamental things: *correctness* and *style*. The correctness properties we will use cover characteristics that impinge on the specification-independent minimum requirements for correctness, irrespective of the problem being solved (that is, *weak correctness* criteria rather than strong formally proved correctness criteria). The style properties cover characteristics associated with both high and low-level design, and the extent to which the software's functionality at all levels is specified, described, characterized and documented. It is therefore convenient to divide the *quality-carrying* properties associated with the structural forms of programs into four basic categories. In order of precedence (for classification purposes) these categories are:

- correctness properties (minimal generic requirements for correctness)
- structural properties (low-level, intramodule design issues)
- modularity properties (high-level, intermodule design issues)
- descriptive properties (various forms of specification/documentation)

The next step, which is probably the most difficult and open to question, is to identify a set of properties that adequately cover these four categories. In presenting a set of properties that do what is required we do not pretend that this is the only or the best set of properties for a particular application. What we do however claim is that a model of this form provides a very useful way to tackle the problem of software product quality systematically and constructively. Over time, we may expect with experience of application, that a more refined and accurate set of properties will emerge. Our criteria for selecting and defining these properties has been based on the requirements that they form an *orthogonal* (nonoverlapping), *consistent*, and *complete* set. Quality defects that are discovered that do not result from a violation of any of these properties will provide the constructive force needed to refine the property model and definitions of the properties. It may, for example, be appropriate to have a set of properties that focus much more intermodular issues.

With this model there may be occasions where we must make the choice of classifying a defect as for instance either a correctness problem or perhaps a modularity problem. In this case the precedence (which we have arbitrarily chosen) suggests the problem should be to classified as a correctness problem. The precedence rule is correctness problems before structural, before modularity, before descriptive problems. Our reasoning in choosing this order is based on our perception of their relative impact on the utility of software.

We should not be discouraged by this situation as this sort of framework is used over and over again in science to build any good and useful empirical model. What we propose is a framework for climbing the ladder of software product quality and thrust our foot only on the first rung—the task remains to climb to the top of the ladder. We will list a set of properties that may be associated with structural forms and then provide definitions for each of these properties. Only by examining the definitions of each in detail will it be possible to judge how successful we have been in characterizing the quality-carrying properties of the structural forms of programs. Some quality-carrying properties are much harder to define and characterize than other properties. To assist with the definition process we will use a variety of devices including both positive and negative examples. For example, take the property *structured* which can apply to a number of structural forms. We can at least partially define the property structured in an indirect way by identifying deviations from being structured. When a deviation from being "structured" occurs it results in a quality defect. We claim that a structural form must be *structured* in order not to contribute negatively to the quality of a software product. Exhibiting any of the deviations such as being "unstructured" or ill-structured prevents a given structural form from having the property of being "structured". The "definitions" we will use are always corrigible and open to refinement and improvement. They do however provide a basis for developing a useful constructive model of software product quality.

There are a number of defects associated with structural forms that are language-specific. For example expressions in C may have side-effects whereas expressions in Pascal do not permit side-effects. We identify the impact of each product defect on the high-level quality attributes of the ISO-9126 Software Product Evaluation standard in each case. In the *quality impact* specification, the intent is that the greatest impact is upon the first listed quality attribute and then successively lesser impacts are on the other listed quality attributes. These decisions are empirical.

The order, and hence precedence, of subproperties, within a category has been chosen based on a judgment of the relative impact of a subproperty on its parent property. This is purely an empirical heuristic decision. However it is not hard to justify to most people that a violation of a computability property is likely to have a much more significant impact on correctness than violation of a consistency property (see the following page)—hence the rank of C1 for "computable" compared with C8 for "consistent".

We will now look at a range of properties that are relevant to structural forms. In each case, deviations from particular properties result in quality defects.

A. Correctness Properties

Correctness properties fall broadly into three categories that deal with computability, completeness and consistency. The particular properties we have selected have been chosen in such a way that any violation of one of these properties could potentially mean, that under some circumstances at least, the software may not exhibit its intended functionality. For example, if the structure of a loop indicates that under some circumstances there is a risk that it may not terminate then this risk threatens correctness and hence functionality and is therefore a quality defect. By contrast a loop may be classified as *progressive* if upon examining its structure we find that for all paths through the loop there is evidence of progress towards termination and it is not possible to by-pass the termination point. These characteristics are a weak statement of the formal requirements for a proof of termination. The correctness properties we will use are therefore:

C1. Computable	Result obeys laws of arithmetic, etc.
C2. Complete	All elements of structural form satisfied
C3. Assigned	Variable given value before use
C4. Precise	Adequate accuracy preserved in computations
C5. Initialized	Assignments to loop variables establish invariant
C6. Progressive	Each branch/iteration decreases variant function
C7. Variant	Loop guard derivable from variant function
C8. Consistent	No improper use or side-effects

Each of these properties which is ranked from highest to lowest precedence will now be defined and discussed in more detail.

C1. Computable: A structural form is computable if it only involves computations that are defined according to the standard theory of computation and are within the limits defined by the program, the programming language and/or the machine. The property applies to all expressions including subscripts.

Applies to: $\Longrightarrow$ expressions
Quality Impact: $\Longrightarrow$ functionality, reliability
Sample Defects: (noncomputable)

- — division by zero or other impossible computation
- — subscript out-of-range
- — writing to an unopened file
- — division by a variable of unknown status
- — square root of a negative number or number of unknown status.

C2. Complete: A structural form exhibits the property of being *complete* when it has all the necessary elements to define and implement the structural form so that it may fulfil its intended role in a way that will not impact reliability or functionality. As well as using a general completeness property we have chosen to identify three other properties which are specializations of completeness, i.e., *assigned, initialized* and *progressive*. These properties are singled out because of their key contributions to correctness of loops and other statements.

Applies to: $\Longrightarrow$ objects, modules, statements
Quality Impact: $\Longrightarrow$ functionality, reliability, usability, maintainability
Sample Defects: (incomplete)

- — if-statements that may abort (language-specific)
- — self-assignment (e.g., $x := x$)
- — unreachable code in a selection mechanism
- — module that generates no output.

C3. Assigned: A variable is *assigned* if it receives a value either by assignment, input, or parameter assignment prior to its use. The property assigned is a specialization of the completeness property that applies specifically to variables and data structures of all types.

Applies to: $\Longrightarrow$ variables
Quality Impact: $\Longrightarrow$ functionality, reliability
Sample Defects: (unassigned)

- — use of a variable in a term or expression that has not been previously assigned a value.

C4. Precise: A variable or constant is imprecisely typed when its precision is not sufficient to meet the required accuracy of the computation.

Applies to: $\Longrightarrow$ variables and constants
Quality Impact: $\Longrightarrow$ functionality, reliability
Sample Defects: (imprecise)

- — use of single precision when a computation demands double precision
- — use of an integer when problem demands only in the range 0..9.

C5. Initialized: A loop structure is *initialized* if all variables in a loop are initialized prior to loop entry and as late as possible prior to loop entry. The initialized property is a specialization of the assigned property that applies to loops. It is therefore a completeness property. Initialization, is central to the correct and efficient functioning of loop structures. It is also an area of a computation that is vulnerable. The most appropriate initialization is that which establishes the loop invariant for a loop. Defects in initialization can arise largely from doing either too much or too little in the initialization step [10]. Initialization defects identify composition problems between the body of a loop and the initializations chosen for the loop variables. They identify higher level structural problems rather than simply the assignment of variables. The problem of a variable not being initialized for use in a loop is a variable integrity defect, rather than on initialization defect.

Applies to: $\Longrightarrow$ Loops
Quality Impact: $\Longrightarrow$ functionality, reliability, maintainability
Sample Defects: (underinitialized, overinitialized, prematurely initialized)

- — For a detailed treatment of initialization defects see [10, Ch. 12].

C6. Progressive: A loop or recursive algorithm is *progressive* if there is clear evidence that the structure makes progress towards termination with each iteration or recursive call and the associated variant function is bounded below by

zero. Recursive calls must have a reachable base case. The progressive property is a completeness property of iterative and recursive constructs.

Applies to: $\Longrightarrow$ modules (recursive), loops
Quality Impact: $\Longrightarrow$ functionality, reliability, maintainability
Sample Defects: (nonprogressive)

— nested loop where outer loop variables are only changed (make progress) in an inner preguarded loop or called function [10, Ch. 12].

C7. Variant: A loop guard (or inductive guard in recursive structures) is *variant* if it defines a relation (the variant condition) that is congruent with, and derivable from, the variant function used to prove termination of the loop [10] (e.g., for loop that has a variant function $j - i - 1$ which is decreased by $i := i + 1$ and/or $j := j - 1$ an appropriate guard that is variant would be $i \neq j - 1$. A variant guard has a form that makes it easy to assess the termination behavior of a loop.

Applies to: $\Longrightarrow$ guards (for loops and recursive structures)
Quality Impact: $\Longrightarrow$ functionality, reliability, maintainability
Sample Defects: (nonvariant)

— loop that uses a boolean variable flag as a loop guard (e.g., *while* not found *do*...) is nonvariant and not derivable from the variant function for the loop.

C8. Consistent: A structural form is consistent if its usage maintains its properties or functionality and if all its elements contribute to and reinforce its overall intent or effect. Side-effects and any other forms of misuse violate the consistency of a structural form.

Applies to: $\Longrightarrow$ modules, statements, guards, expressions variables and records
Quality Impact: $\Longrightarrow$ functionality, reliability, maintainability, reusability, portability, usability.
Sample Defects: (inconsistent)

— using a variable for more than one purpose in a given scope
— modifying a loop variable on exit from a loop
— using a variable as a constant
— changing a variable in an expression (is a side-effect)
— unused input (read(x);...:read(x))
— output of a variable twice without change
— use of variables/constants of different precision/type in a computation.

B. Structural Properties

The structural properties we have used focus upon the way individual statements and statement components are implemented and the way statements and statement blocks are composed, related to one another and utilized. They enforce the requirements of structured programming and demand that there should be no *logical, computational, representational* and *declarative* redundancy or inefficiency of any form either in individual statements or in sequences or in components of statements. There is a requirement that computations should be expressed directly, efficiently, simply and not in an obscure fashion. Redundant testing is sometimes advocated as a means to increase reliability but this is not defensible at the intramodule level. Another requirement is that every structural form that is declared in a program should be utilized. This applies to such diverse entities as variables and modules. While the main focus of structural properties is intramodular some of these properties also apply at higher levels of organization. These structural properties are:

S1. Structured	Single-entry/single-exit
S2. Resolved	Data structure/control structure matching
S3. Homogeneous	Only conjunctive invariants for loops
S4. Effective	No computational redundancy
S5. Nonredundant	No logical redundancy
S6. Direct	Problem-specific representation
S7. Adjustable	Parameterized
S8. Range-independent	Applies to variables (arrays), types, loops
S9. Utilized	To handle representational redundancy

S1. Structured: A structural form exhibits the property of being *structured* if it follows rules of structured programming [11]. That is, there should be only a single point of entry and exit for every control structure. Too many conditions associated with a guard and poor bracketing of an expression also represent deviations from being structured.

Applies to: $\Longrightarrow$ sequences, guards, and expressions
Quality Impact: $\Longrightarrow$ maintainability, reliability, functionality
Defects: (unstructured, ill-structured)

— exit from the middle of a loop
— multiple returns from a function
— loop guard with too many conditions.

S2. Resolved: A structural form is *resolved* if the control structure of the implementation involved matches the structure of the data [12] or the problem [10] in the sense advocated by Jackson (that is, the control structure matches data structure and thereby satisfies the correspondence principle). At all times the strategy seeks to construct loops that *minimize* the number of variables they change.

Applies to: $\Longrightarrow$ loops
Quality Impact: $\Longrightarrow$ maintainability, efficiency
Defects: (unresolved)

— use of a single loop to process a two-dimensional array [ref. 10, Ch. 12].

S3. Homogeneous: An iterative or recursive form is *homogeneous* if it can be described by an invariant where the major predicates assume a conjunctive form (e.g., the invariant must be of the form "A *and* B *and* ..." but A etc. may involve disjunction). An iterative or recursive form is inhomogeneous if it involves an invariant where major predicates must be combined by disjunction.

Applies to: $\Longrightarrow$ loops, modules (recursive)
Quality Impact: $\Longrightarrow$ maintainability
Defects: (inhomogeneous)

— a loop structure with functionality that is not cohesive (see examples in [ref. 10, Ch. 12].

S4. Effective: A structural form exhibits the property of being *effective* when it has all the necessary elements and only the necessary elements to define and implement the structural form. Elements beyond what are necessary and sufficient to specify the process, computation, data structure or user-interface violate the property of effectiveness for the particular structural form. In other words, unnecessary variables or computations or lack of simplification of structures or computations, violates the property of effectiveness. It applies particularly to expressions and assignment statements and other statements but not to conditions. The redundancy resulting from failing to have a resolved control structure (according to Jackson's methodology) is excluded from an "ineffective" classification. It is treated as unresolved (see above). Note we might at first think an assignment in the body of a loop that does not change its value with each iteration could be classified as a sequence (the loop body) defect. However this structure only shows as a defect in the context of the loop and so it is a defect of the loop structure. Any executable statement that does not change the state of a computation is classified as ineffective. Also if the same result can be achieved more simply then a computation is ineffective (e.g., using $\{N \geq 0\}\ i := 0$ do $i \neq N \rightarrow i := i + 1$ od instead of $i := N$ makes the sequence consisting of the loop plus the initialization ineffective)

Applies to: $\Longrightarrow$ expressions, statements
Quality Impact: $\Longrightarrow$ usability, efficiency, maintainability
Defects: (ineffective)

— assignment that establishes an already-established condition
— expression with unnecessary computation (e.g., $y := x + 1 + 1$).

S5. NonRedundant: A structural form exhibits the property of being *nonredundant* when it has all the necessary logical elements and only the necessary elements to define the structural form. Conditions beyond what are necessary and sufficient to specify the process violate the property of nonredundancy for the particular structural form. In other words, unnecessary conditions or lack of logical simplification of computations, violates the property of nonredundancy. This property is distinguished from "effective" in that it involves some form of logical redundancy rather than computational (arithmetic/algebraic) redundancy. *In other words it applies to conditions not assignment statements.* The redundancy resulting from failing to have a resolved control structure (according to Jackson's methodology) which can be a form of high-level logical redundancy is excluded from a "redundant" classification. It is instead given the more specialized classification of being *unresolved.*

Applies to: $\Longrightarrow$ guards
Quality Impact: $\Longrightarrow$ efficiency, maintainability
Defects: (redundant)

— testing a condition that has already been established.

S6. Direct: A computation is expressed *directly* if the abstraction, choice of representation and the structure of the computation are congruent with the original problem being modelled by the computation. An indirect way of framing a computation makes it harder to understand because at least one more level of detail must be considered. When a computation is expressed indirectly there is, from the user's view at least, an inefficiency in the representation. Something additional, that is not present or relevant to the original problem is introduced. The use of boolean flags or numbers to represent other real-world items and clever but obscure computational tricks are all typical of an indirect way of formulating and representing computations. Modern languages through devices like enumerated types make it easy to avoid an indirect style of programming. In early versions of languages like Fortran it was difficult to avoid the use of an indirect style of programming for many applications. Some argue for an indirect way of formulating computations to gain efficiency but this is hard to defend given the power of today's computers.

Applies to: $\Longrightarrow$ statements, expressions, variables, constants, types
Quality Impact: $\Longrightarrow$ maintainability, efficiency
Defects: (indirect)

— use of flags (boolean and others)
— use of numbers to represent colours
— use of clever tricks, (e.g., $(I/J)^*(J/I)$ to initialize identity matrix)
— use of boolean variables to represent conditions.

S7. Adjustable (parameterized): A structural form is *adjustable* if it contains no undeclared constants (apart from 1, 0, or -1) and if the minimum number of single-purpose variables needed to support the computation it performs are used.

The word adjustable has been chosen to specifically deal with parameterization internal to the structure of modules and programs.

Applies to: $\Longrightarrow$ module calls, expressions
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability
Defects: (unadjustable)

— if a structural form contains numbers instead of defined constants.

S8. Range Independent: A structural form is *range-independent* if both its lower and upper bounds are not fixed numeric or character constants. This property applies particularly to array specifications and iterative structures designed to process a segment of elements in an array. Most often arrays and loops assume a fixed lower bound. For example, an array-sort will be written to sort all the N elements in an array $a[1 \ldots N]$. A more widely useful, range-independent, algorithm is one which sorts a segment of an array $a[L \ldots U]$.

Applies to: $\Longrightarrow$ declarations (arrays), loops
Quality Impact: $\Longrightarrow$ reusability, maintainability
Defects: (range-dependent)

— an array type or variable is declared with a fixed lower or upper bound
— an array-processing loop assumes processing starts at 0 or 1.

S9. Utilized: A structural form is *utilized* if it has been defined and then used within its scope. This property applies to

all forms of data structures and modules. Its negation identifies any form of redundancy resulting from declaration as opposed to logical, representational or computational redundancy.

Applies to: $\Longrightarrow$ objects, modules, all forms of declared data
Quality Impact: $\Longrightarrow$ maintainability, efficiency
Defects: (unutilized)

— a variable that has been declared but is not used
— a function is declared but is not used.

C. Modularity Properties

The modularity properties employed largely address the high-level design issues associated with modules and how they interface with the rest of a system. These issues include how a module encapsulates its data, how it is coupled to other modules, how loose its functionality is, how flexible it is and what potential does it have for reuse. These modularity properties are:

M1. Parameterized	All inputs accessed via a parameter list
M2. Loosely coupled	Data coupled
M3. Encapsulated	Uses no global variables
M4. Cohesive	The relationships between the elements of an entity are maximized
M5. Generic	Is independent of the type of its inputs and outputs
M6. Abstract	Sufficiently abstract—is no apparent higher level form.

M1. Parameterized: A module is *parameterized* if it contains as parameters all and only the necessary and sufficient inputs and outputs to characterize a particular well-defined function/procedure.

Applies to: $\Longrightarrow$ modules
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability
Defects: (unparameterized, over-parameterized, ill-parameterized)

— unparameterized, (e.g., module with no parameters)
— over-parameterized (e.g., swap$(i, j, a[i], a[j])$)
— ill-parameterized (e.g., function that modifies input parameters).

M2. Loosely Coupled: A module or a program is *loosely coupled* if all module calls are data-coupled (see Myers [13]) to the calling program/module.

Applies to: $\Longrightarrow$ module calls
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability, reliability
Defects: control-coupled, stamp-coupled, content-coupled, common-coupled, externally coupled

— see Myers [13] for a detailed discussion of these defects.

M3. Encapsulated: The way variables are used can have a significant impact on the modularity and hence self-contained quality of modules, programs and systems. A variable (or constant or type) should be used only within the scope in which it is defined. If it satisfies this property it is said to be *encapsulated.* A module that uses global variables or side-effects violates this property. Consistent with the discipline of identifying quality defects with the lowest-level structural form to which they may be associated, encapsulated is treated as a variable-usage property even though it impacts modularity. To build quality software that is easy to maintain and reuse we should ensure that each module is allowed to access and modify only those data items that are absolutely needed by the module. Other data items should be "hidden" in other appropriate modules.

Applies to: $\Longrightarrow$ variables, constants and types
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability, reliability
Defects: (unencapsulated)

— use of variable in a module that has not been declared in the module's scope.

M4. Cohesive: A structural form is *cohesive* if all its elements are tightly bound to one another and they all contribute to achieving a single objective or function. Statements within a cohesive component should be organized from the least to the most dependent, that is, the last statement, in a sequence depends on all its predecessors (see [14]). Any interleaving of independent statements destroys cohesion. A variable-dependency graph may be used to assess the cohesion of a given sequence of statements [15]. This concept of cohesion applies at more than one level. That is, for three blocks in sequence, the computations in the third block should depend on the computations in the preceding two blocks, and so on. Identifying statements and blocks that could be executed in parallel is a good way of assessing cohesion and independence.

Applies to: $\Longrightarrow$ sequences
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability
Defects: (uncohesive)

— a module with a lot of parameters has low cohesion as it probably implements more than one well-defined function.
— loop with dispersed initialization (see [10]).

M5. Generic: A module is *generic* if its computations are abstracted to a type-parameterized form.

Applies to: $\Longrightarrow$ modules
Quality Impact: $\Longrightarrow$ maintainability, reusability, portability
Defects:

— primitive type-dependant (procedure to swap just integers).

M6. Abstract: An object/module is *sufficiently abstract* if there is no obvious, useful higher level concept that encompasses the structural form.

Applies to: $\Longrightarrow$ Objects
Quality Impact: $\Longrightarrow$ reusability, maintainability
Defects:

— specialized module/object (e.g., declaring a car object class instead of vehicle object class).

D. Descriptive Properties

There are three primary properties that reflect how well software is described. Explicit *comments* may be added to

a program to document how the implementation realizes its desired functionality by manipulating variables with prescribed properties. To precisely characterize the functionality the process can be taken a step further by including *precondition* and *postcondition specifications* for all functions and other significant computations in a program. Also by appropriate choice of identifiers and module names it is possible to make an important contribution to the analyzability and *self-descriptiveness* of programs. The descriptive properties are therefore:

D1. Specified	Preconditions and postconditions provided
D2. Documented	Comments associated with all blocks
D3. Self-descriptive	Identifiers have meaningful names.

D1. Specified: A module or program or other structural form is *specified* if its functionality is described by preconditions and postconditions. A structural form is *fully specified* if all blocks are specified and loops have attached invariants, variants, preconditions and postconditions. The highest level of specification involves the use of a formal specification language. When a structural form is not specified there is always a doubt about its intended functionality. Use of specifications, if done properly, provides the most rigorous form of documentation and description.

Applies to: $\Longrightarrow$ objects, modules, loops, sequences
Quality Impact: $\Longrightarrow$ functionality, maintainability, reliability, usability, portability, reusability
Defects: (under-specified, unspecified, ill-specified)

- — functionality is not described by preconditions and postconditons (under-specified)
- — contains no preconditions or postconditions (unspecified)
- — specification is ambiguous, inaccurate, inconsistent or incomplete (ill-specified)

D2. Documented: A structural form is *documented* if its purpose, strategy, intent and properties are all explicitly and precisely defined within the context of the structural form.

Applies to: $\Longrightarrow$ objects, modules, loops, sequences, module-calls, data structures, variables, constants, types
Quality Impact: $\Longrightarrow$ maintainability, portability, reusability, usability
Defects: (undocumented, under-documented, over-documented, ill-documented).

- — structural form contains no comments (undocumented)
- — insufficient comments are used to describe purpose (under-documented)
- — more comments are used than are needed (over-documented)
- — documentation is misleading or wrong (ill-documented)

D3. Self-Descriptive: A structural form is self-descriptive if its purpose, strategy, intent, or properties are clearly evident from the choice of names for modules and various identifiers are meaningful and congruent with the context of the application.

Applies to: $\Longrightarrow$ objects, modules, module-calls, variables, constants, data structures,
Quality Impact: $\Longrightarrow$ maintainability, portability, reusability, usability
Defects: (undescriptive, over-described, ill-described)

- — name chosen bears no relation to property (undescriptive)
- — name chosen is unnecessarily long (over-described)
- — name is ambiguous, misleading or wrong (ill-described)

E. Refining the Definition of Quality-Carrying Properties

The definitions we have provided for the various quality-carrying properties are by no means comprehensive. Our intent, because of the empirical nature of the model, has been to provide base working definitions that can be refined as necessary in a given application context. There are three options for refining these definitions. One way to do this is to aim for completeness by trying to see if the property is applicable to each possible structural form in its *usage* and *representation* and *context*. In our definitions above we have listed the structural forms to which each property applies but we have not detailed how the property is interpreted for each structural form. The second thing we can do is extend the list of examples showing defects and positive instances of the property. For some properties (involving correctness and logical redundancy and incompleteness, etc) it is possible to provide formal definitions. For example, the concept of strongest postconditions $\mathrm{sp}(P, S)$ [16] may be used to formally prescribe when a statement S, executed in the presence of a precondition P is redundant. That is, the strongest postcondition *after* executing S under P will be equivalent to P if S is redundant. In other words, S does not change the state of the computation. This may be expressed formally by the following equivalence $\mathrm{sp}(P, S) \equiv P$.

V. Model for Software Product Quality

Having defined *a set of structural forms, a set of quality-carrying properties* and *a set of high-level quality attributes* we can proceed to build a constructive model of software product quality by defining the relations among these three sets of entities. The first of these tasks is to identify and associate a set of quality-carrying properties with *each* of the structural forms that may be used in a program (implicitly we have already done this in the previous section). This is the key relation that may be used to support building quality into software. The constructive theorem that supports this task is:

Constructive Theorem: If each of the quality-carrying properties associated with a particular structural form is satisfied when that particular structural form is used in a program, then that structural form will contribute no quality defect to the software.

From this follows the complementary assertion: *if any of the properties associated with a structural form are violated in a program, then each violation will contribute a quality defect to the software.*

Using a model based on these two principles allows us to achieve our two primary goals. It gives us *direct advice* on building quality into software and at the same time it may be used to assist in the systematic classification of quality defects in software. Two important consequences follow if we are willing to accept and adopt this model:

1) *Building Quality into Software:*
The task of building quality into software reduces to systematically ensuring that all the quality-carrying properties associated with each structural form used in a program are satisfied for all applications of that structural form in the program.
2) *Systematic Detection/Classification of defects:*
Detecting quality defects in software reduces to systematically checking whether, for each structural form, in all of its occurrences, any of its quality-carrying properties that imply high-level quality attributes are violated}.

In the previous section we have already identified the structural forms to which each of the quality carrying properties apply. To build quality into software it is far more useful to identify all the quality-carrying properties associated with each structural form. As we will see below, for quick reference, this information may also be neatly summarized in tabular form.

We will now systematically work through the product properties that imply quality attributes for each of the main structural forms in imperative programs. For each structural form its properties will be listed according to the precedence rules set out in Section VII below. In this context it is important to define exactly what we mean by a given property when we associate it with each structural form. For example, *inconsistent* translates into something different when applied to an expression compared to what it means for a function/module. In our presentation here we will not fully develop properties in this way. What we will do instead is give instances for various properties and structural forms. For example, we will provide instances of inconsistency as it applies to modules, expressions and so on.

A. The Relation Between Structural Forms and Quality-Carrying Properties

In the previous sections we have identified a set of quality-carrying properties that can be attached to structural forms in programs. We also identified the structural forms to which each of the properties could be attached. For building quality into software it is important to organize this information so we can see at a glance what quality-carrying properties are associated with each structural form. In what follows we will carry out this organization and finally summarize the results in a table. The highest level forms are not dealt with here because we have chosen not to emphasize system-level quality issues. Examples of quality defects associated with each structural form are given.

1) Object Integrity: An object may be realized by declaration of an abstract data type. Its key high-level quality attributes depend on its, specification, completeness and the level of abstraction employed in choosing the data structure and its operations. The internal quality properties of a module are handled by the quality-carrying properties that are assigned to its data structures/variables and its operations or functions.

a) Quality-Carrying Properties: Complete, utilized, abstract, specified, documented, self-descriptive

Defects:

— incomplete (does not enable access to all components of data)
— over-specialized (the abstraction is not at a high enough level)
— unutilized (declared but not used)
— unspecified (no precondition/postcondition specifications)
— undocumented (no comments stating the functions of object)
— unself-descriptive (poorly chosen name for object).

2) Module Integrity: The term module is used to describe procedures, functions and subroutines, etc. The quality of a module depends on how well its functionality is described, its level of abstraction, its degree of independence and how easy it is to reuse. Its internal quality is covered by a quality-carrying properties that are assigned to the statements from which it is composed.

a) Quality-Carrying Properties: Complete, progressive (recursive modules), consistent, homogeneous, utilized, loosely-coupled, parameterized, generic, abstract, specified, documented, self-descriptive.

Defects:

— incomplete (no apparent input and/or output parameters)
— nonprogressive (not all inductive branches appear to make progress)
— tightly coupled (control information passed to module)
— unparameterized (module defines a fixed computation)
— type-specific (handles data of a predetermined type only)
— unutilized (module declared but not used)
— unspecified (no precondition/postcondition specifications)
— undocumented (no comments stating the purpose of module)
— unself-descriptive (module name poorly chosen).

3) Sequence Integrity: A sequence is used to describe computations formulated using one or more consecutive executable statements in a given block. The quality of a sequence structure depends on its level of cohesion and whether there is any transfer of control out of the sequence. Other quality problems associated with sequences like, for example, redundant assignments in a sequence are handled at the statement or statement component level. That is, a redundant assignment is *ineffective*. This conforms to the principle of always classifying quality defects at their source. It should be noted that all executable statements in a program (if-statements, assignments, loops, etc.) have the quality-carrying property *effective* associated with them. If they do not change the state of computation or the same overall result can be achieved more simply then an executable statement (or sequence of statements) is ineffective.

a) Quality-Carrying Properties: Structured, effective, cohesive, specified, documented

Defects:

- — unstructured (e.g., contains a *goto*, *break*, etc. in sequence (block))
- — ineffective (e.g., $i := 0$; do $i \neq N \rightarrow i := i + 1$ od can be replaced by $i := N$)
- — uncohesive (exist interleaved statements in a sequence that are independent).

4) Loop Integrity: The structural form loop is used to characterize the various forms that implement iteration (e.g., the *while, repeat, for*, etc. loop structures used in Pascal and other imperative languages). The quality of a loop depends on its partial and total correctness properties [8], the way it is composed (including its initialization) and on how well its behavior is described. The quality of a loop is also strongly influenced by whether it has a single point of entry and exit. The property is associated with the loop body (rather than the loop itself) which consists of a sequence of statements that possess the property of being *structured.*

a) Quality-Carrying Properties: Complete, initialized, progressive, consistent, resolved, homogeneous, effective, range-independent, specified, documented

Defects:

- — incomplete (only decreases variant function)
- — nonprogressive (not clear that all branches make progress)
- — inconsistent (a loop that does just one iteration)
- — underinitialized, overinitialized, uninitialized (see [10])
- — ineffective (does not change state of computation)
- — unresolved (hidden loop, if-statement and loop-guard are same)
- — inhomogeneous (see example [10])
- — unspecified (no invariant and variant function specified)
- — undocumented (no comment on the purpose of loop)

5) Selection Integrity: The selection structural form is used to characterize if-statements, case statements and switch statements, etc. How selection statements are implemented can have a significant impact on the quality of programs. The key quality property associated with selection is completeness; that is, whether all cases have been covered and also whether all cases are reachable. Another problem with selection statements is the inconsistency associated with switch statements (in C) which allows control to flow from one selection into another. Other problems associated with selection statements are handled as either guard defects or as defects associated with statements that are guarded by the selection structure. For example, when the execution of the statements in the branch of a selection do not change the state of a computation, the defect is classified as its source, that is, as a problem with the guarded statements rather than with the selection statement itself. Of course if no branches change the state of a computation the selection statement is redundant (that is, ineffective).

a) Quality-Carrying Properties: Complete, consistent, effective

Defects:

- — incomplete (e.g., not all cases covered, or there is an unreachable statement)
- — inconsistent (e.g., fall-through in a C switch statement)
- — ineffective (e.g., statement does not change state of computation).

Note: a guard in an if-statement that tests an established condition is classified as a logical defect of the guard rather than a defect of the if-statement.

6) Module-Call Integrity: In the scope where it is employed, a module-call has the purpose of changing the state of a computation by changing the values of one or more variables. The way parameters are used in a module-call can have an impact on quality. There are two main problems: a given variable may be passed more than once and/or fixed constants (e.g., numbers) are passed as parameters. Like other executable statements, a module call is ineffective if it can be demonstrated that it does not change the state of a computation. Other quality problems associated with module calls can be traced back to problems with the use of statements and variables in the body of the module. Module calls that employ either no input and/or output parameters might be thought to violate the property of completeness. However this structural defect should be traced back to a defect in the design of the module in the first place rather than associating it with the module call.

a) Quality-Carrying Properties: Consistent, effective, adjustable, documented

Defects:

- — inconsistent (e.g., same parameter passed twice)
- — ineffective (e.g., call is computationally redundant)
- — unadjustable (e.g., numbers passed as parameters).

7) Assignment Integrity: Most of the quality-carrying properties associated with assignment statements are not associated with the assignment itself but with its components; its variables and the expression being evaluated. The two quality-carrying properties that remain are whether the statement is redundant or not and whether it is complete. It would be possible to consider both these issues in terms of effectiveness because they involve no state change. However the completeness property deals with whether an assignment has been properly formed which may have direct implications for correctness.

a) Quality-Carrying Properties: Complete, effective

Defects

- — incomplete (statement lacking an additional term, e.g., $x := x$)
- — ineffective (statement does not change state of computation).

8) Guard Integrity: The guard structural form is used to guard state-changing statements in loops and selection statements. Guards are logical constructs and therefore vulnerable to logical redundancies and inefficiencies which may be identified using standard logical equivalence formulas that define simplifications. For loop guards, the ideal form is that they define a relation (the variant condition) that is congruent with the variant function used to prove termination of the loop [10]. Sometimes guards contain numbers or other fixed constants (e.g., *while* $i < 100$ *do*) which make them not adjustable. It may seem appropriate to assign the property adjustable to the guard itself. However following the rule of always associating quality defects with the lowest level structural with

which they are associated, this problem is assigned to the component expression that forms part of the relation. That is, the expression should be adjustable ("100" is not) rather than the guard. Guards should be appropriately structured using parentheses.

a) Quality-Carrying Properties: Variant, structured, nonredundant

Defects:

— redundant (e.g., logical redundancy "*while* ch = space & ch $\neq$ eol *do*")
— nonvariant (e.g., *while* $a[i] \neq x$ *do* ...).

9) Expression Integrity: Expressions are the primary vehicle for implementing computations. The quality-carrying properties associated with expressions relate to computability, side-effects, the presence of any redundancy and the use of fixed constants. Other quality problems associated with expressions relate to their constituent variables and are classified at a lower level accordingly. The appearance of fixed constants in expressions is however definitely an expression defect and not a variable or constant problem.

a) Quality-Carrying Properties: Computable, consistent, structured, effective, direct, adjustable

Defects:

— uncomputable (e.g., divide by zero)
— inconsistent (e.g., contains side-effects)
— inconsistent (e.g., use of variables/constants of different precision)
— ineffective (e.g., contains computational (arithmetic) redundancy)
— unadjustable (e.g., contains numbers or other fixed constants).

10) Record Integrity: A record is just a composite variable. It therefore has associated with it the same quality-carrying properties as variables. These properties are discussed in detail in the next section. An added problem with records is that they may admit the insecurity of only being partially initialized.

11) Variable Integrity: The way variables are used has a significant impact on the quality of programs. A variable possesses correctness, structural, modularity and descriptive properties. In terms of correctness, a variable must always be assigned before it is used, it must be of the appropriate precision and it should only ever be used for a single purpose within a given scope. The only structural obligation for variables is that they be utilized if they are declared. The modularity quality of a variable is that it must only be used (encapsulated) within the scope in which it is declared. Use of a variable at a lower scope, that is, as a global variable, is probably the single most significant thing that detracts from the quality of imperative programs. Assignment to a global variable in such a context results in a side-effect of the lower scope. From an external perspective side-effects are hidden actions which have a severe impact on the analyzability of programs. Another vital quality requirement for variables is that any name chosen should clearly and accurately characterize the property that is ascribed to the variable. It is also wise to strengthen the definition of the intended use of a variable by including a comment that defines its property at the time the variable is declared.

a) Quality-Carrying Properties: Assigned, precise, consistent, encapsulated, direct, range-independent, utilized, documented, self-descriptive

Defects:

— unassigned (e.g., variable not assigned prior to use in expression)
— imprecise (e.g., single precision used when double needed)
— inconsistent (e.g., variable used for more than one purpose in scope)
— unencapsulated (e.g., global variable used in function)
— unutilized (e.g., declared variable not used)
— undocumented (e.g., no comment when variable declared)
— unself-descriptive (e.g., variable 'x' used to store "maximum")
— indirect (e.g., use of a boolean flag).

12) Constant Integrity: A declared constant possesses correctness, structural, modularity, and descriptive properties. In fact it possesses a subset of the quality-carrying properties of variables. It differs from a variable only in that being used for more than one purpose and being assigned are not issues.

a) Quality-Carrying Properties: Precise, encapsulated, direct, utilized, documented, self-descriptive

Defects:

— imprecise (e.g., single precision used when double precision needed)
— unencapsulated (e.g., global constant used in function)
— unutilized (e.g., declared constant not used)
— undocumented (e.g., no comment when constant declared)
— unself-descriptive (e.g., constant 'x' used to store "3.14159265").

13) Type Integrity: The set of quality-carrying properties that apply to types is a contraction of the set of properties that apply to variables and constants. The problems with global use that apply to variables also apply to types. A type should be used if it is declared and it should be both self-descriptive and documented.

a) Quality-Carrying Properties: Encapsulated, direct, utilized, range-independent, documented, self-descriptive

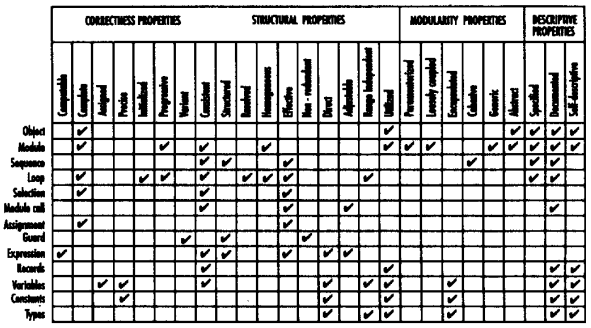
Defects:

— unencapsulated (e.g., global type used in function)
— unutilized (e.g., declared type not used)
— undocumented (e.g., no comment when type declared)
— unself-descriptive (e.g., type 'x' used to represent "jobtype").

A table summarizing the quality-carrying properties associated with each structural form is given above.

This table defines two of the sets of relations of our generic quality model: *the quality-carrying properties associated with each structural form* and *the set of structural forms which exhibit a particular quality-carrying property*. The information presented in this way is useful for two purposes: assisting those that implement software to build in quality to the various structural forms that programs are composed of and to assess whether particular quality-carrying properties have been built into software.

TABLE II
QUALITY-CARRYING PROPERTIES FOR STRUCTURAL FORMS



B. The Relation Between Quality-Carrying Properties and Quality Attributes

Another important way in which we can view product quality-carrying properties is to directly relate them to the high-level quality attributes that are used to characterize the quality of software. For this purpose we will use the high-level attributes advocated in the International Standard ISO-9126 Software Product Evaluation. We will however add to this list the attribute of reusability as we believe (see Section III) this characteristic is important enough in its own right to deserve such high level status. Reusability clearly depends on low-level design.

The intent of this view is to more explicitly identify the requirements for building each of the high-level quality attributes into software. The product properties we have identified and defined provide direct advice, or rather a specification that must be satisfied to build the desired high-level quality attributes into software. This specification is clearly empirical. When we try to construct a systematic process for deciding which product properties contribute to which high-level quality attribute we run into the same difficulty that the naturalist John Muir had: "when *we try to pick out anything by itself, we find it connected to the entire universe*". For example, it is not hard to make an argument that most, if not all quality-carrying product properties make a contribution to the maintainability of software. Given this situation, the issue has to be "which product properties make the most significant contribution to the maintainability of software?" Making this judgement is clearly an empirical step. The criteria we will use to make this judgement are as follows:

- a minimal subset of quality-carrying properties will be selected in each case
- the properties will be ranked in terms of the assessed importance of their contribution to the quality attribute.

We will now examine each of the major quality attributes for software and associate with each a set of quality-carrying properties. Little work has been done on trying to make such direct links. This is a problem which we consider needs to be thoroughly explored.

1) Functionality: The quality attribute functionality depends heavily on two things, correctness properties and the extent to which the functionality of a system is accurately characterized. A program cannot be correct in its own right; it can only be correct with respect to a specification. In a similar way, it makes sense to talk about the functionality of a program with respect to its specification. The product properties that impact functionality are listed in order of their likely impact on functionality (see also the table that follows).

COMPUTABLE
COMPLETE
ASSIGNED
PRECISE
INITIALIZED
PROGRESSIVE
VARIANT
CONSISTENT
STRUCTURED
ENCAPSULATED
SPECIFIED

The only problem with this is that it does not say which structural forms possess these properties. To use this information effectively we must link it back to the various structural forms of the product which possess these various characteristics. This information is available in Table II, Section V.A. Properties like *structured* and *encapsulated* are included in this list because of the widely held opinion that there is a much higher risk of their being functional defects in software that is neither structured nor modular in form.

2) Reliability: Functionality implies reliability. The reliability of software is therefore largely dependent on the same properties as functionality, that is, the correctness properties of a program. However where differences arise is in relation to completeness. A program can be correct with respect to its specification, and therefore satisfy its functional requirements and yet fail because the inputs do not satisfy the expected precondition. Unstructured code and side-effects represent high-risk factors for reliability. It is claimed that compliance to the structured and encapsulated properties significantly reduce these risks. For this high-level quality attribute and the remaining attributes the relevant properties are summarized in Table III.

3) Usability: Usability is concerned with the quality of the user interface, its design and performance characteristics. In this specification we have chosen not to describe it in detail as it depends on a completely different set of structural forms (buttons, menus, etc.) which have their own set of quality-carrying properties. At this point we will simply list the quality-carrying properties defined for programs which are also relevant to specifying the quality of a user interface (see Table III).

4) Efficiency: The position taken is that computational and logical redundancy are important factors that effect the efficiency of a program. Ensuring that there is a match between program control structure and data structure also makes a contribution to efficiency. These contributions to efficiency are all independent of the algorithms or strategies used in an implementation. In many instances their contribution to efficiency is not likely to be nearly as great as the contribution of the algorithms. For example, using an $O(N \log_2 N)$ instead of an $O(N^2)$ sorting algorithm will have a dramatic impact

on efficiency for large values of N. Unfortunately it will not be practical or even possible to determine the optimal computational complexity for each algorithm in an implementation. We must therefore settle for a much weaker qualitative model for efficiency that is based on excluding various forms of redundancy. The contributing properties are listed in Table III.

5) Maintainability: There is a widely held belief that software which is very easy to maintain is software of high quality [17]. There is a very wide range and a large number of quality-carrying properties that make an important contribution to the maintainability of software. There are two primary concerns for maintaining software: it must be clearly specified and well-documented so the intent of the whole and various fragments of the software is beyond doubt; the software must also be easy to understand. And, for software to be easy to understand, there is a multitude of structural, modularity and descriptiveness factors that need to be satisfied. The most important of these are listed in Table III

6) Portability: A program or system is portable if it requires little or no changes to compile and run it on other systems. The three primary things that affect portability are machine dependencies, compiler dependencies and operating system dependencies. There are two strategies that may be used to minimize portability impact: effective use of parameterization can isolate and minimize machine dependencies, modularization and isolation of compiler/system/language dependencies in a single (or small number of) place can also improve the portability of software. The relevant quality-carrying properties are listed in Table III.

7) Reusability: There are at least two interpretations of reusability. It may used to describe software that is ease to adapt and modify for use in other contexts or, more strictly, it may describe software that has properties that allow it to be used in other application contexts without change. We will apply the latter interpretation here. Using this interpretation, for a module to possess the quality attribute reusability it depends on two things: the functionality of the module must be clearly and precisely described; the module must decoupled and therefore independent of its implementation context. A structural form is *reusable* if it uses standard language features, it contains no machine dependencies, it implements a single well-defined function and all computations are fully adjustable, use no global variables and contain no side-effects. All ranges associated with computations and data structures should have both their lower and upper bounds parameterized. To be completely reusable no variable in a computation should be assigned to a number or any other fixed constant. All constants used should be declared. Type independence also increases reusability. The relevant quality-carrying properties are listed in Table III.

Table III summarizes the relations between high-level quality attributes and the set of quality-carrying properties. The table may be used to answer two questions: *"which quality-carrying properties may be used to satisfy a given high-level quality attribute?"* and *"which high-level quality attributes does a given quality-carrying property impact?"*

The information in this table is useful for assisting the designer to build particular high-level quality attributes into software and for understanding which product properties impact particular high-level quality attributes.

TABLE III
RELATIONSHIP BETWEEN QUALITY ATTRIBUTES AND QUALITY-CARRYING PROPERTIES

	CORRECTNESS PROPERTIES								STRUCTURAL PROPERTIES								MODULARITY PROPERTIES						DESCRIPTIVE PROPERTIES		
	Complete	Complete	Assigned	Precise	Initialized	Progressive	Verified	Consistent	Structured	Standard	Homogeneous	Effective	Non - redundant	Direct	Adaptable	Range Independent	Unified	Parameterized	Loosely coupled	Encapsulated	Cohesive	Generic	Abstract	Specified	Documented	Self-descriptive
Functionality	✔	✔	✔	✔	✔	✔	✔	✔	✔															✔		
Reliability	✔	✔	✔	✔	✔	✔	✔	✔	✔										✔	✔				✔		
Usability		✔						✔				✔												✔	✔	✔
Efficiency										✔		✔	✔	✔			✔									
Maintainability		✔			✔	✔	✔	✔	✔	✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Portability								✔							✔			✔	✔	✔	✔				✔	✔
Reusability								✔							✔	✔		✔	✔	✔	✔	✔		✔	✔	✔

VI. ASSURING QUALITY HAS BEEN BUILT INTO SOFTWARE

To assure, that the sort of quality properties we have described in our proposed model for software product quality are adhered to, some means of inspection are needed. Code inspection has long been recognized as a powerful method for assuring and improving the quality of software. The only problem is that it is very costly and labour-intensive to perform systematically and rigorously on large amounts of software.

To overcome this problem we have developed a powerful and flexible static analysis system (code auditor) [9] which supports and conforms to the model of software product quality that we have described. This system, which is rule-based, allows users to analyze the quality of software from a number of different perspectives. For example, it is possible to assess the high-level quality attribute *maintainability* of the software by running all the rules that have been classified as impacting this high-level quality attribute. To do this, simply involves selecting maintainability in the quality-attributes menu. In a similar way it is possible to assess high-level product characteristics such as *correctness*. It is also possible to select subordinate correctness properties such as *assigned* which checks whether all variables are assigned before being used. Overall the system for running the various rules is very flexible. Implementation of many of the rules in the system involves a detailed and sophisticated static analysis of the program text.

In analyzing software the PASS (Program Analysis and Style System) tool provides a comprehensive report on the quality of C programs. In the summary part of the report it presents such statistics as number-of-quality-defects-per-thousand-lines-of code, the number of maintainability, reliability, etc defects in the entire file. There is also a summary of how many times each defect was found. The main part of this report uses a format not unlike that used by compilers to report syntax errors. The line number and function where each defect occurs is pin-pointed. This system has been successfully employed to assess the quality of a wide range of industry software. The system allows an industry average to be maintained [9].

VII. DISCIPLINE FOR CLASSIFYING QUALITY DEFECTS

As we have seen programs consist of structural forms that describe data and processes. In order to classify and

describe quality defects we need to talk about the properties of structural forms. An alternative approach would have been to use a model where relationships between structural forms were considered but this was rejected because it makes the characterization of defects more difficult. Provided our chosen set of structural forms admits composition (the *sequence* structural form does this) there is no need to speak directly of interrelationships between structural forms. For example suppose, in a sequence of statements, a guard tests a condition that has already been established by prior statements then we say that the guard is "redundant" (it violates the nonredundant usage property of guards). That is, we have assigned a defect to a structural form because of the problem associated with its usage in a particular context.

It was stated earlier that a primary requirement for the proposed model of software product quality was that it should possess a defect classification procedure that was repeatable. That is, two people familiar with the model should arrive at the same classification for any given defect.

The fundamental basis for classification we have employed to achieve repeatable classification involves the use of precedence to establish order. For this to work it is essential to have a fully structured system of precedence rules for classifying defects. Otherwise, we may end up classifying a given defect in an arbitrary number of different ways. One primary and two secondary classification rules are needed to implement system.

A. Primary Classification Rule

Always associate a defect with the lowest level structural form for which it assumes the status of a property.

This rule greatly simplifies the decision process for classification and ensures that defects are characterized in terms of their *origin* rather than in terms of their consequences.

Once the task of deciding which structural form the defect is to be associated with is accomplished the next task it to identify the property of the structural form which is violated. If there is any conflict in making this decision then first intercategory and then, if necessary, intracategory precedence rules may be applied to make the final classification decision.

The high level precedence for the *principal product-property classifications* which dictates the priority order is:

$$\text{correctness} \rightarrow \text{structural} \rightarrow \text{modularity} \rightarrow \text{descriptive}$$

That is, classification of a defect as violating a correctness property takes precedence over a classification violating a structural property, and so on.

In a similar way, within a given principal property, a precedence order also applies. Take, for example, the correctness properties:

C1. Computable
C2. Complete
C3. Assigned
C4. Precise
C5. Initialized
C6. Progressive
C7. Variant
C8. Consistent

Here, if a choice must be made say, between classifying a defect as violating the "complete" property, and the "consistent" property, the former should be given precedence, and so on. Defect classification will now be illustrated by discussion of several examples.

Examples:

1) Unassigned Variable in an Expression of an Assignment Statement:

Defect Classification: We have the choice of associating this defect either with an assignment, with an expression or with a variable:

assignment integrity → incomplete (incorrect)
expression integrity → incomplete (incorrect)
variable integrity → unassigned (correct)

At first glance this defect might seem like an expression integrity problem (e.g., the expression is undefined if one of it's variables is unassigned). However the *source* of the problem is not the expression itself but the variable. The rule that a variable should be assigned before use has been broken.

2) Unreachable Statement in an If-Statement:

Defect Classification: In this case we have the choice between classifying the problem as a defect in the if-statement or as a problem with the statement that is not reachable.

selection integrity → incomplete (correct)

We do not classify a statement as unreachable because this amounts to treating it as a *relation* between structural forms. Instead we classify it as a defect of a structural form at one higher level where it reverts to a *property* of a structural form. We say a statement at a higher level (the if) is incomplete—this is the source of the defect.

3) Modification of a Loop Variable on Exit from a Loop:

Defect Classification: This could be potentially seen as either a defect in the sequence or as a problem with variable usage.

sequence integrity → inconsistent (incorrect)
variable integrity → inconsistent (correct)

Precedence dictates that the source of the problem is variable usage. Changing a loop variable on loop exit means the variable is being used for more than one purpose because the invariant property associated with its use in the loop is destroyed. A formal treatment of this problem is given elsewhere [18].

4) Double Initialization, that is, Initialization of a Variable Prior to Execution of a Loop and then Initialization of the Same Variable Again Prior to Loop Entry: This might potentially be seen as an assignment integrity or a variable integrity problem.

Defect Classification:

assignment integrity → ineffective (incorrect)
variable integrity → inconsistent (correct)

Precedence dictates that the problem be classified as a variable usage problem rather than as a problem with the assignment.

5) Function which Returns no Values but which makes an Assignment to at Least One Variable External to the Function:

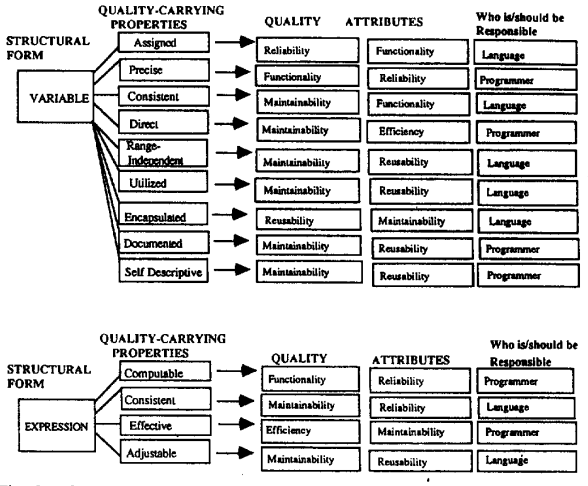


Fig. 3. Quality-carrying properties and programming languages.

Defect Classification: This example was chosen because it indicates the presence of a composite defect. There is inconsistent use of a module—it returns no values. And secondly, the module exhibits a side-effect because it makes an assignment to a global variable. Fixing either of these problems would still leave a remaining problem. So we end up with the following two classifications.

module integrity → inconsistent (correct)
variable integrity → unencapsulated (correct).

VIII. Languages and Software Product Quality

The model for software product quality that we have provided raises a number of important issues about programming language design. Most existing languages leave the responsibility for satisfying the various quality-carrying properties in the hands of the designer/programmer. This places a very heavy burden for software quality on the shoulders of the programmer. Design and code inspections and static analysis tools may be used to assist the programmer in ensuring that various quality-carrying properties are satisfied. These approaches however do not offer the best way to deal with the issue.

A far better way to proceed is to shift the major part of this burden from the programmer to the language designer and the compiler writer. By appropriate choices in the design of languages and compilers many of the quality-carrying properties associated with various structural forms can be satisfied or enforced. This means that the programmers have to change their style of implementation and/or submit their programs to much more rigorous compiler checks which insist that quality requirements are satisfied before a compiler will produce executable code. As examples, Fig. 3 illustrates for variables and expressions where the responsibilities for satisfying various quality-carrying properties can/should reside.

Elsewhere we have shown how a simple yet powerful language may be defined to implement the quality requirements that we have defined [19].

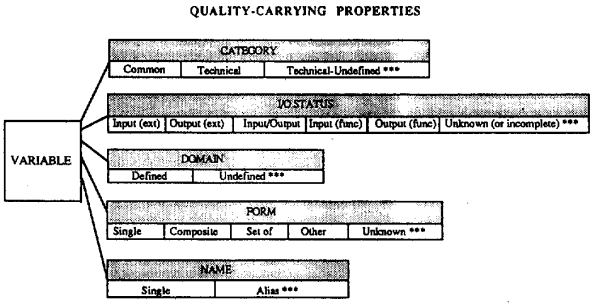


Fig. 4. Quality-carrying properties for a requirements variable.

IX. Application of the Model to Other Software Products

The model we have described is generic. It can be utilized in many other contexts including for other products of software development. Our intent here is not to develop quality models for such applications but rather to establish the feasibility of the approach by sketching a small part of a quality model for a requirements specification. Such specifications consist of a set of required functions each of which has associated with it a set of input and output variables. In addition, there are usually a set of constraints associated with the variables and functions. Relations are used to define these constraints. As with the model we have developed for software implementations we may associate a comprehensive set of quality-carrying properties with each of the components that are used to define individual requirements (e.g., the input and output variables and any constraints and/or properties associated with the function and its variables). In addition, other quality-carrying properties must be defined which apply to subsets and even the complete set of requirements (e.g., matters relating to consistency and completeness). Proceeding in this way we can develop a comprehensive quality model for software requirements. To illustrate the process let us focus once again on variables. As in programs, the way variables are used in a requirements specification has a key impact on its quality. The quality-carrying properties needed for a variable in a specification are somewhat different from those needed in a program. To specify the quality of a variable in a program it is only necessary to deal with properties that are either only present or absent. In a requirements specification things are different. We need to specify quality-carrying properties that can take on a small set of values. A subset of the quality-carrying properties of variables is given below together with the brief explanation of their role and the identification of defect status.

A. Quality-Carrying Properties of Variables in a Requirements Specification

Fig. 4 shows the quality-carrying properties (category, I/O status, domain, form, name) for variables in a requirements specification. A quality defect occurs when a property has a value marked with "***".

1) Category: For the purposes of specifying requirements the two possible values of category are sufficient. Common is

used to characterize variables whose properties are commonly known (e.g., phone-book) and technical describes variables whose properties are context-dependent. There is a tendency in requirements to refer to common variables and not to bother to define them for the purposes of shorthand. While this is a defect in a requirements specification it is not a problem that is hard to overcome. However when a technical variable (e.g., reentry-velocity) is not defined this is a much more serious quality defect that needs rectification.

phone-book → common → undefined → not serious defect
reentry-velocity → technical → undefined → serious defect

Note to *define* common variables we can use an example or cite a reference (e.g., Brisbane telephone book). We may also refer to a certain page in an organization's data model.

2) I/O-Status (source/sink properties): The quality of a requirement's specification depends very much on the source/sink properties of variables. If this is defined/explicit this is important, if it is unknown then it detracts from the quality of a requirement. Each variable should have a *source* and at least one *sink*. If there is more than one source there could be a problem. If this information is not known then it detracts from the quality of the requirement. The source could be EXTERNAL (e.g., user input, an existing database etc.) or the source may be the output of a function/process. A sink might be the input to a function. Variables can also be input to and output from a function—in this case a variable must also have some other source.

3) Domain: The domain of a variable must be defined.

4) Form: From a requirement it should be possible to determine a variable's form. If this is not be possible—it represents a quality defect.

5) Alias: Defect when more than one name is used to refer to a particular variable

Using the approach we have begun to outline here for variables it is possible to build a comprehensive quality model for a requirements specification. A similar approach can also be taken to construct quality models for user interfaces and other software products.

X. Conclusion

The model we have defined and illustrated here provides an explicit process for building quality-carrying properties into software. These properties in turn imply particular quality attributes. In other words we have proposed a model that establishes the link between tangible product characteristics and less tangible quality attributes.

An important advantage of this model is that it can assist in conducting a systematic search for quality defects. The model guides us where to look for defects and also indicates the properties that will need to be violated to create defects. This information provides constructive guidance for building a comprehensive set of defects for any particular language environment.

No claim is made that the model we have proposed is "correct" or that it is the only one that might be employed. The model is empirical and therefore corrigible and open to refinement. Irrespective of disputes or disagreement over the details of the model the framework provided offers a means for, in the longer term, providing direct guidance for building quality into software both from the top-down (during design) and from the bottom-up (during implementation). In addition the model supports assuring the quality of software and systematic classification of quality defects. While the details of the model might need to be changed and refined the framework should provide a sound constructive foundation for achieving a better understanding of software product quality.

References

[1] B. W. Boehm, J. R. Brown, M. Lipow, G. J. MacLeod, and M. J. Merritt, *Characteristics of Software Quality*. New York: Elsevier North-Holland, 1978.
[2] B. Kitchenham, "Towards a constructive quality model," *Software Eng. J.*, pp. 105–112, July 1987.
[3] B. W. Kernighan and P. J. Plaugher, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.
[4] M. Deutsch and R. Willis, *Software Quality Engineering*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[5] T. P. Bowen, "Specification of software quality attributes," Rome Laboratory, New York, Tech. Rep. RADC-TR-85-37, vols. 1–3, 1976.
[6] R. Nance, "Software quality indicators: An holistic approach to measurement," in *Proc. 4th Ann. Software Quality Workshop*, Alexandria Bay, New York, Aug. 1992.
[7] Software Product Evaluation—Quality Characteristics and Guidelines for Their Use, ISO/IEC Standard ISO-9126 (1991).
[8] R. G. Dromey, "A generic model for building quality into products and processes," in preparation.
[9] R. G. Dromey and K. Ryan, *PASS-C: Program Analysis and Style System User Manual*, Software Quality Inst., Griffith Univ., 1993.
[10] R. G. Dromey, "Program Derivation," in *International Series in Computer Sciences*. London, England: Addison-Wesley, 1989.
[11] N. Wirth, "Program development by stepwise refinement," *CACM*, vol. 14, pp. 221–227, 1971.
[12] M. Jackson, *Principles of Program Design*. London, England: Academic, 1975.
[13] G. Myers, *Software Reliability; Principles and Practices*. New York: Wiley, 1976.
[14] R. G. Dromey and A. D. McGettrick, "On specifying software quality," *Software Quality J.*, vol. 1, no. 1, pp. 45–74, 1992.
[15] R. G. Dromey, "A framework for engineering quality software," keynote address, *7th Australian Software Eng. Conf.*, Sydney, Australia, Sept. 1993.
[16] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. New York: Springer-Verlag, 1989.
[17] T. Manns and Coleman, *Software Quality Assurance*. London, England: MacMillan, 1988.
[18] Si Pan and R. G. Dromey, "A formal basis for measuring software product quality," *17th Australian Comput. Sci. Conf.*, Christchurch, NZ, Jan. 1994.
[19] B. Oliver and R. G. Dromey, "SAFE: A programming language for software quality," *1st Int. (Asia-Pacific) Conf. Software Quality and Productivity*, Hong Kong, China, Dec. 1994.

R. Geoff Dromey is the Foundation Professor of the School of Computing and Information Technology at Griffith University. He founded the SQI in 1989.

Through the SQI, he has worked closely for a number of years with industry, national and international standards bodies and government. He has worked at Stanford University, the Australian National University, and Wollongong University before taking up the Chair at Griffith University in Brisbane Australia. His current research interests are in applying formal and empirical methods to improve the quality of software and the productivity of software development.

Dr. Dromey has authored/co-authored two books and over fifty refereed research papers. He is on the Editorial Board of four international journals.