SOFTWARE PRODUCT QUALITY: Theory, Model, and Practice

R. Geoff. Dromey,
Software Quality Institute,
Griffith University,
Nathan, Brisbane, QLD 4111
AUSTRALIA

Abstract

Existing proposals for software product quality have not been underpinned by the sort of empirical theory and supporting models that are found in most scientific endeavours. The present proposal provides a set of axioms and supporting software and quality models needed to construct a comprehensive model for software product quality. This model has been developed using a requirements-design-implementation strategy to ensure that it meets the needs of a number of different interest groups. A significant advantage of the proposed model is that it allows the problem of software product quality to be broken down into intellectually manageable chunks. Other important features of the model are that it has enough structure to characterize software product quality for large and complex systems and it will support practical specification and verification of quality requirements.

Keywords

Software product quality models, quality attributes, ISO-9126, quality-carrying properties, large-scale software projects.

SOFTWARE PRODUCT QUALITY: Theory, Model, and Practice

R. Geoff. Dromey,
Software Quality Institute,
Griffith University,
Nathan, Brisbane, QLD 4111
AUSTRALIA

There is always a first step in a journey of ten thousand miles.

Chinese Proverb

1. Introduction

Proposals for modelling software product quality have had very limited success[1-3]. We suggest there are several clear reasons for this.

- Foremost is the fact that the proposed models have not been underpinned by the sort of empirical theory and supporting models that are found in most scientific endeavours.

- Secondly, these models have neither acknowledged nor properly exploited the fact that software has a set of *behaviours* and *uses* which correspond directly to its high-level quality attributes. What is interesting is that these behaviours and uses map directly to the needs of the different interest groups involved with software.

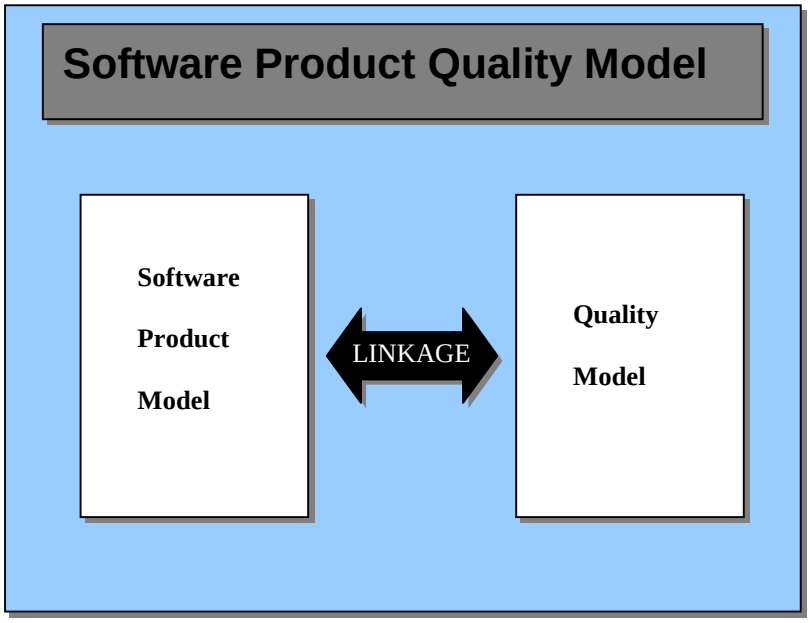
- Thirdly, the definitional and model-construction strategies employed have not been disciplined enough to meet the challenges of the task.

- Finally, there are a very large number of tangible properties of software that are known to positively influence its quality but these properties have never been organised into any sort of systematic framework that could maximise the impact of their cumulative weight.

Our claim is that when these matters are properly attended to it is possible to construct a practical model of software product quality. We will employ a goal-directed *requirements-design-implementation* strategy to develop a model for software product quality that will attend to these matters. This approach has been taken quite deliberately for two reasons: firstly there are a number of different interest groups who have quite distinct software product quality requirements. Our model will need to properly accommodate these requirements; secondly, this strategy is one that people involved with software will find easy to relate to because of its wide application in software development.

The first task in building a software product quality model is to identify what the intended applications of the model are and to address the needs of the different interest groups that will use the model in these different applications. This corresponds to identifying the user requirements of the model.

The second stage in building a model for software product quality is to identify a suitable architecture/design for the model. The following diagram encapsulates the high level architecture of our model.



That is, our obligations are threefold:

- To construct a *software product model*
- To construct a *quality model*
- To link the models for software and quality to construct the model for *software product quality*.

Once we have constructed a framework that defines and supports our software product quality model our final obligation is to define a strategy for implementing this model.

The proposals in this paper represent an extension, a refinement, and an attempt to provide a conceptual framework and empirical theoretical underpinning for earlier proposals on software product quality[1,2,8,9]

2. Terminology and Framework

Before we can make serious progress with the construction of a model for software product quality we need to sort out some use of terminology. This step will allow us to establish what is often called in technical terms a *universe of discourse*. The advantage of doing this is that it allows us to set out how we intend to use a variety of words and how we intend to link these words to various concepts. A lot of problems arise in discussions about software product quality because there is confusion about how various terms are used.

Software is composed of both high and low-level components. Through these components, and through the ways they are composed, software exhibits properties that characterise and distinguish it from other artefacts. Sometimes the words: *characteristics, factors* and *attributes* are used as alternatives for what we will refer to here as properties. We will use these latter terms, but in quite distinct ways. We will distinguish several different kinds of properties which may be either tangible (concrete) or abstract (intangible) and either functional or non-functional properties. Tangible properties must be either measurable, assessable, calculable or detectible by either manual or automated means.

Some of the properties of software are *desirable*. We call these desirable properties, qualities or quality attributes. Quality, or more specifically, a set of quality attributes is the vehicle through which the different interest groups express their needs of software. A goal-directed approach to building a quality model for software is effective for accommodating and balancing the needs of these different interest groups. The set of *desirable properties* or *quality attributes* of software provides an abstract or high-level specification for what we will call software product quality.

We next make the observation that the quality attributes we associate with software correspond either to behaviours or uses. A behaviour is something that the software itself *exhibits* when it executes under the influence of a set of inputs (e.g., *reliability* and *efficiency* are behaviours). From a functional perspective, a behaviour of a software system can be characterised by the set of responses (including outputs and metrics) the system exhibits through the execution and interaction of sets of its functions in response to one or more sets of inputs.

A use is something that different interest groups *do* with or *to* software (e.g., *portability* and *maintainability* are uses). We may also give *uses* a functional interpretation. That is, a use is a *user-performed function* where the software (or software system) is the input, e.g.

output ← User-performed-function(software).

The output varies depending on the use. For the use “maintainability” the output may be a modified version of the input, whereas for the use “learnability” the output could be a set of verifiable statements that reflects the user’s understanding of the system or a set of tasks (plus time to learn) that the user knows how to perform using the software system.

A Software Characteristic is an abstract property (determinable) of software that classifies a set of tangible quality-carrying product properties. It is not a behaviour or a use. *Modularity* is an example of a software or product characteristic. Software characteristics may correspond either to a set of functional entities or a set of non-functional tangible properties. Software characteristics help software to satisfy quality attributes. For example, tangible *machine independent* properties of software components contribute to a software system’s portability. It follows that software characteristics may be used to support the definition of high-level quality attributes.

3.Requirements of A Software Product Quality Model

To formulate the requirements for a software product quality model three issues must be addressed: the different *interest groups* need to be identified; the intended *applications* of the model need to be spelled out; and it is necessary to establish the *quality needs* or *perspectives/views* of the different interest groups.

An appropriate starting point for obtaining a set of requirements is to ask the fundamental question:

Question:
Who has to be satisfied with software?

The response is: *clients and sponsors*, *users* and *developers/maintainers* (*auditors* might also be proposed as an important interest group with quite distinct quality needs but we have chosen not to include them in this exposition). These different interest groups have different, and sometimes competing/conflicting primary needs of software. For a particular interest group to be *satisfied* with software it will need to meet their specific functional and/or non-functional requirements. Together, these requirements all fall under the heading of quality requirements.

The next fundamental question on the path to obtaining a set of requirements is to sort out about the uses of our intended model.

Question:
Why do we want to define, characterise and build a model for software product quality in the first place?

Being clear at the outset about the use or uses of a definition, a theory, or a model, can make the job of constructing that definition, model or theory a lot easier. Such an appraisal gives us a concrete goal to aim for and a set of requirements to satisfy.

We suggest that the primary applications of such a model are:

- to use such information to assess and prescribe or specify the quality requirements of software products,
- to understand what is entailed in constructing software with specified levels of quality,
- to enhance the behaviours and facilitate the uses of software,
- to use such material to train/educate people how to produce quality software,
- to provide a framework that can be extended to meet particular domain/application-specific quality needs,
- to have a model that can be understood at a number of levels by the different interest groups involved with software,
- to serve as a vehicle for advancing our knowledge of software product quality

In providing a set of requirements for a quality model it is also instructive to revisit Garvin's [6] views on quality. From his studies he concluded that "quality is a complex multifaceted concept" that can be described from five perspectives:

- transcendental view – quality is recognisable but not definable
- user view – quality means fitness-for-purpose
- manufacturing view – quality means conformance to specification
- product view – quality is tied to inherent characteristics of product
- value-based – quality is dependent on how much customer is willing to pay

Kitchenham and Pfleeger[3] have recently discussed Garvin's approach in the context of software product quality. We suggest Garvin's "model" is a useful starting point not as a quality model in its own right but rather as a *specification of a set of requirements for quality models or alternatively as a set criteria for evaluating product quality models*. By this we mean that any proposed model should be rich enough to accommodate Garvin's five perspectives if we regard them as model criteria to be satisfied. Garvin's proposal implies that some interested party holds each of the perspectives. It follows that in designing both quality models and quality products it is important to try to meet the needs of the different interest groups. We will revisit these issues in the discussion which follows. We will also use Garvin's "criteria" to evaluate our proposed model.

4. Design of A Software Product Quality Model

The design for a software product quality model is realised by constructing *a quality model* and a *software model* and then linking these two models to produce a *software product quality model.* Construction of these models will now be described.

4.1 Quality Model

Background for what we will discuss on quality models has a long historic basis that dates back to the Ancient Greek philosophers. Russell's doctrine [4] of *logical constructions* and what Price [5] calls the 'Philosophy of Universals', (which is based on Aristotle's theory of *universalia in rebus)* and the 'Philosophy of Resemblances' provides some useful insights. What this latter theory tells us about the world is that there exists a phenomenon called 'recurrence of characteristics'. If this phenomenon did not exist there could be no conceptual cognition. In other words, if there were no recurrence in the world we could not acquire concepts like those we need to discuss and characterise software product quality.

A second important concept advanced by the Philosophy of Resemblances is the distinction between *determinable* and *determinate* characteristics. Characteristics have different degrees of determinateness. Price claims that these two adjectives are too fundamental to be defined but that their meaning can be illustrated. For example, the characteristic of *being coloured* is determinable and the characteristic of *being red* is a determinate of it. Being red is again a sub-determinable, and has under it the determinates, *being scarlet, brick-red,* etc. Being a mammal is a complex determinable characteristic. Being a dog, or a being a whale are two of the

determinates of this determinable. Whenever two objects resemble each other with less than the maximum intensity we can always say that the same determinable characteristic characterises them both, though not the same determinate. As we have seen two objects may, for example, have different shades of red.

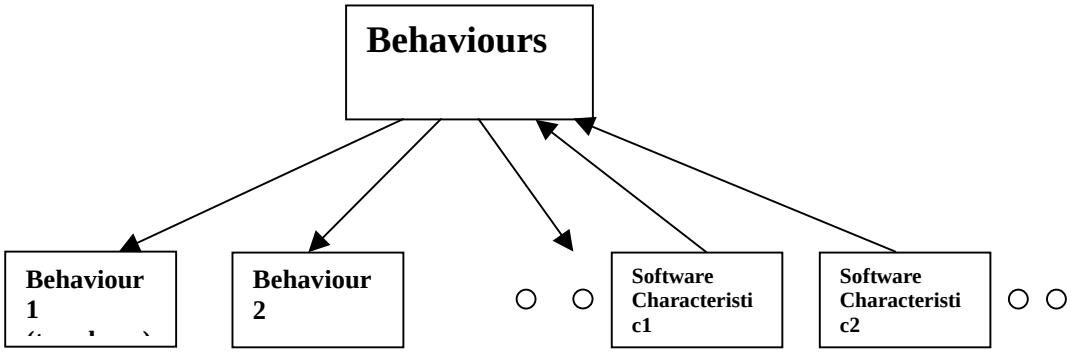
Here we need to extend this work on determinables and determinates to include the notion of incrementally satisfiable determinables. This extension allows us to make claims like: 'the more reliability determinates that are satisfied the greater is the reliability of a given the software system'. This whole approach will be used to underpin the quality model construction process that follows.

We suggest that a constructive strategy can be employed to characterise the behaviours and uses of software that contribute to its quality. In constructing a model for software product quality it is appropriate to apply both *bottom-up* and *top-down* strategies. We seek to enumerate concrete properties and classify them as belonging to software characteristics and, in turn, at the next level up, we seek to enumerate software characteristics that characterise each *behaviour* and each *use* – this corresponds to bottom-up construction. We also employ derivation and/or decomposition to characterise or define abstract properties (e.g. behaviours and uses) in terms of subordinate behaviours, uses and software characteristics – this corresponds to a top-down approach. Both bottom-up and top-down construction have important roles to play in building a quality model.

We will start this discussion by considering the two principles that guide decomposition and derivation.

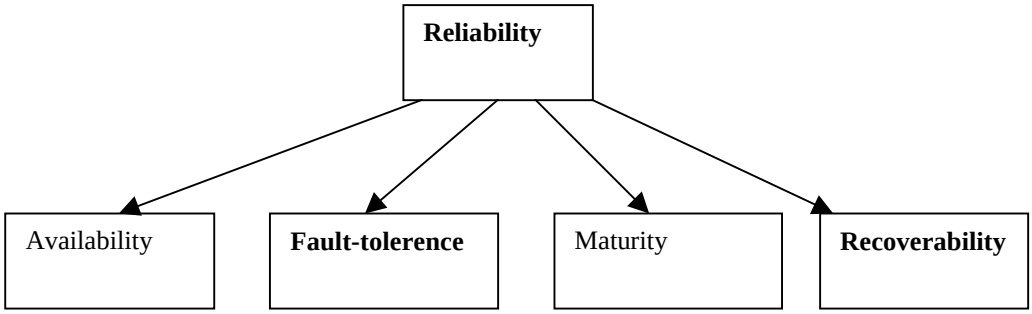
Principle 1:
A *behaviour* can be decomposed and hence defined in terms of subordinate properties which may be described either as behaviours or software characteristics.

The following diagram illustrates this notion schematically. Subordinate behaviours may in turn be hierarchically defined, e.g.



In adopting this principle the challenge we face is where to draw the line between subordinate defining behaviours and contributing software characteristics (note the direction of the arrows in the diagram above is intended to differentiate *defining* from *contributing* causal relationships). The quality attribute reliability provides a good case in point. Abstract properties like *fault-tolerance* and *recoverability* are

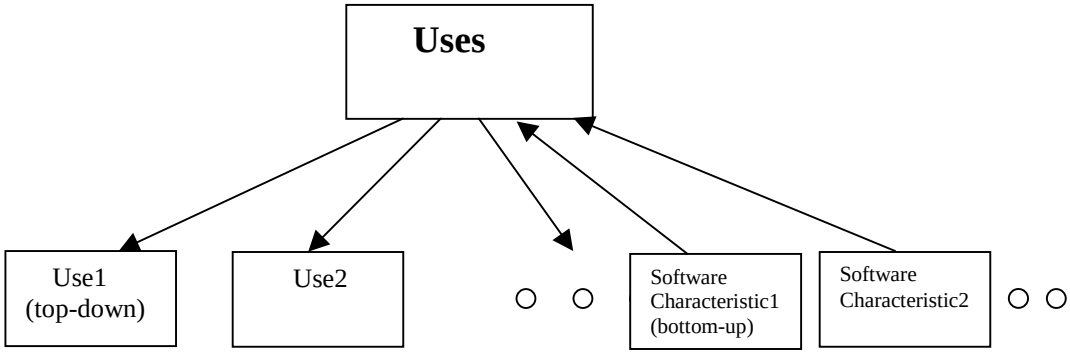
determinates of *reliability*. The issue is, are fault-tolerance and recoverability defining behaviours of reliability or software characteristics that contribute to reliability. What we are confronting here is the tension between top-down definition of quality attributes and bottom-up characterisation (that is, abstract description) or classification of software properties (both non-functional and functional properties) that contribute to high-level quality attributes. Whether we regard fault-tolerance as a behaviour or a software characteristic does not really matter. What is important to recognise is that we have two avenues for arriving at our goal. Some properties, like *modularity* or *correctness*, are clearly software characteristics rather than behaviours or uses. With others like fault-tolerance, it is not so easy to make the distinction. The underlying intuition we are using in making such distinctions is that behaviours tend to be system-wide qualities whereas it is meaningful to associate software characteristics with particular components as well as with an overall system. We can circumvent this issue using determinable-determinate decompositions. As an example, in ISO-9126 [ ]the determinable, reliability can be decomposed into the determinates fault-tolerance, maturity and recoverability (more will be said about this later). Schematically we have:



Uses of software can be treated in a similar way as the following principle suggests.

Principle 2:
A *use* can be decomposed and hence defined in terms of subordinate properties which may be described either as uses or software characteristics.

The following diagram illustrates this notion schematically. Subordinate uses may in turn be hierarchically defined, e.g.

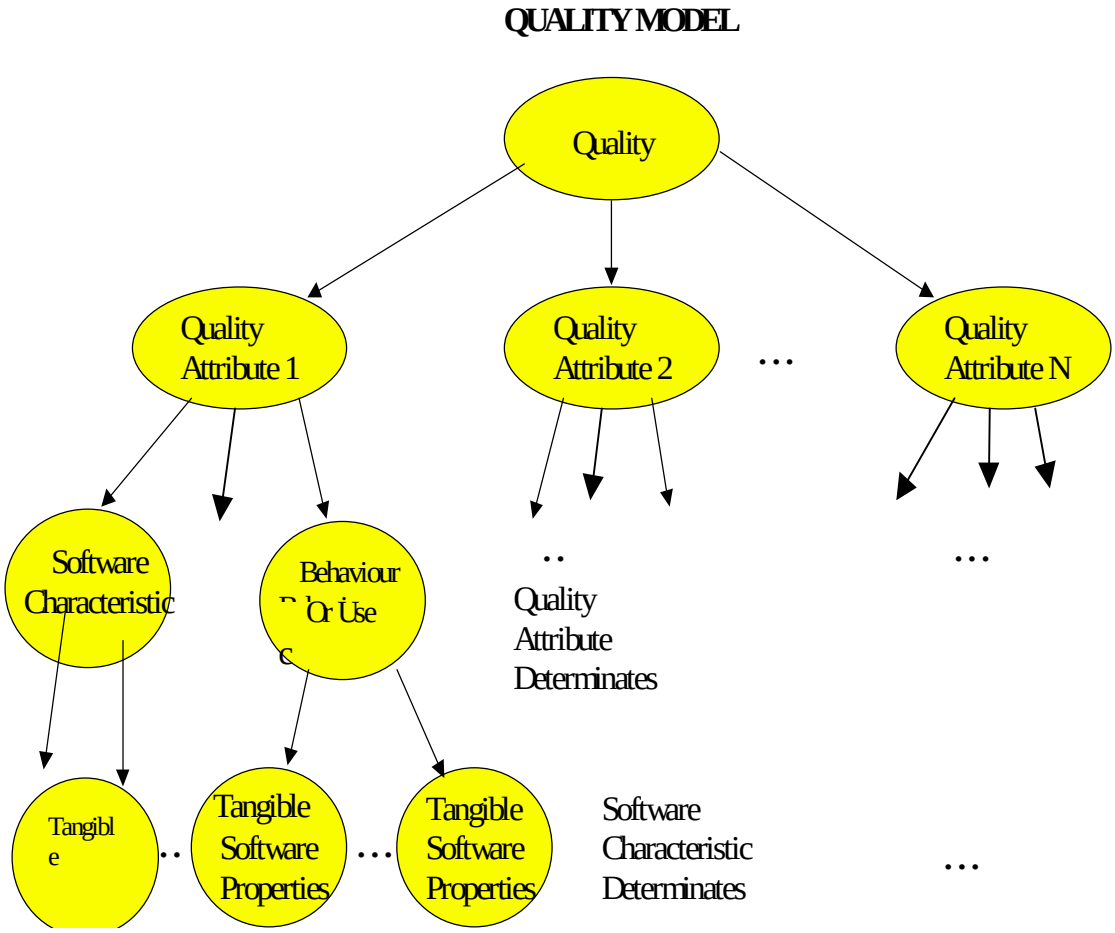


As an example, *usability* can be decomposed into the subordinate uses *learnability* and *operability.*

In proceeding to construct a quality model we will employ the following additional design principles, guidelines and assumptions:

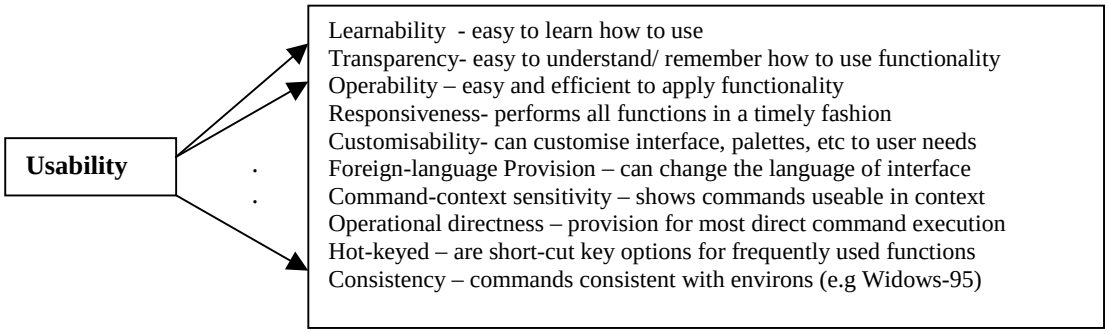
- We choose to associate abstract properties, called quality attributes, with software
- The quality of software may be characterised by a set of high-level quality attributes.
- The quality attributes of software correspond either to a set of domain-independent behaviours of software or a set of domain-independent uses of software.
- The quality attributes of a quality model should be sufficient to meet the needs of all interest groups associated with the software
- Each high-level quality attribute of software is characterised by a set of subordinate properties which are either behaviours, uses or software characteristics.
- Each software characteristic is determined or contributed to by a set of tangible properties that we will call quality-carrying properties.
- Quality-carrying properties may embody either functionality (e.g. a check to see whether all inputs are within their expected ranges contributes to the design principle of modular protection) or non-functional properties (e.g. identifiers should be self-descriptive).

The diagram below characterises the architecture of the quality model we are proposing.

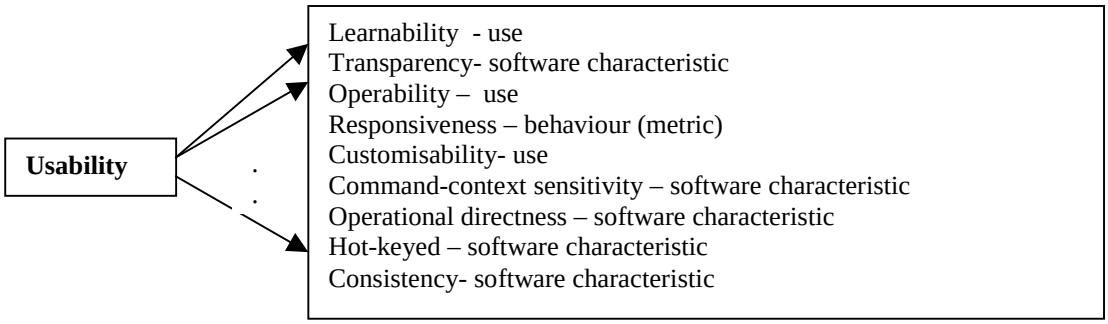


Case Study - Usability

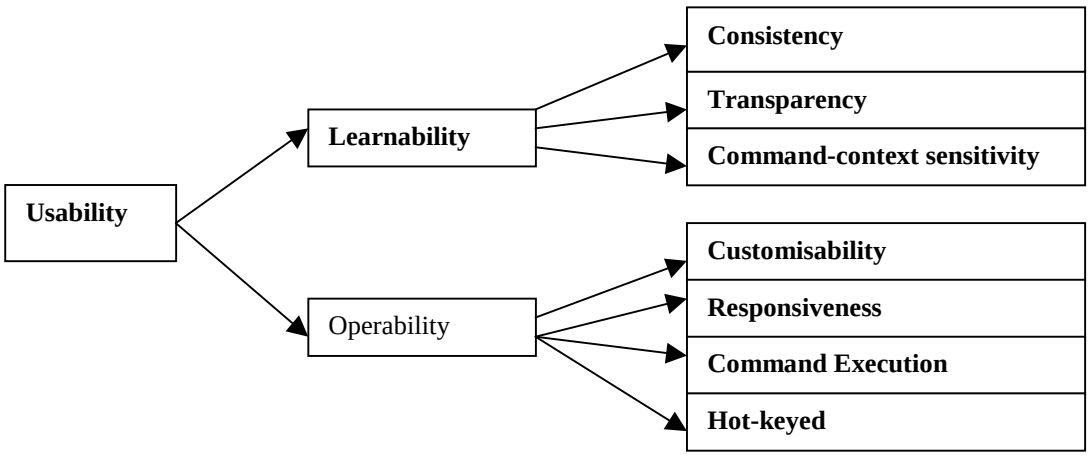
To illustrate the processes of definition and characterisation we will now explore one quality attribute (usability) in slightly more depth. Our intent is not to give a comprehensive software product quality characterisation for usability but rather to illustrate the process. Our starting point is to list (see below) a set of properties that define/characterise or are manifestations of usability.



In order to arrange these properties into an appropriate hierarchy using the definition/characterisation strategy we have described we first classify the properties as either *behaviours, uses* or *software characteristics* according to our definitions of these terms (see below).



Behaviours and *uses* are candidates for being the top-level determinates of usability. We may also have some behaviours acting as subordinate behaviours of others, etc. To accommodate this we must systematically ask for each property whether it contributes to or determines each behaviour or use. For example we must ask does customisability contribute to learnability, to operability and so on. Software characteristics, in turn characterise behaviours and uses. Again a similar process is applied. A possible definitional hierarchy that is an outcome of this whole process is shown below.



What we have provided here is a long way short of a comprehensive specification of usability. Hopefully however, enough detail has been given to illustrate what is entailed in carrying out such a process. Note that the property *Foreign Language Provision* that was in our original list would end up being a determinate of customisability. It is therefore not shown in the hierarchical specification above.

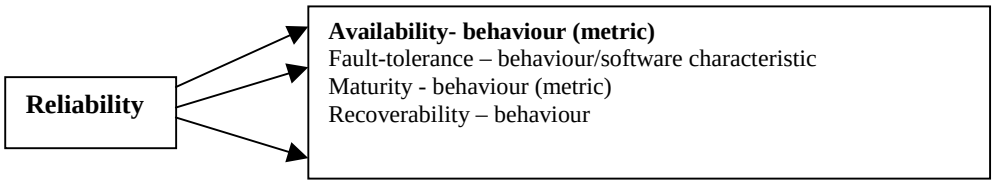
Case Study – Reliability

The characterisation of the quality attribute *reliability* throws up several important points relating to the design and implementation of quality models. In characterising any high-level quality attribute it helps to formulate an underlying functional model or usage model (depending on whether the attribute is a behaviour or a use) and a supporting metrics model. Such models help to minimise overlap of behaviours and address issues of completeness. We will now look at a functional and a metrics model for reliability.

Reliability characterises how a software system behaves in response to four kinds of faults (a) faults that lead to abnormal conditions that are generated externally and/or that show up as inputs to the software system which violate its capacity to satisfy its specified functional behaviour (b) faults that result in errors that occur during the course of execution of a software system that cause it to deviate from its specified functionality and thereby enter an abnormal state (c) errors that occur in the outputs of a software system that are caused by faults in the implementation of the functionality (d) faults that lead to *failure* of the software system.

A defensible functional model for reliability in this context is to say that at the highest level there are three non-overlapping classes of functionality that a system must implement to exhibit reliable behaviour: error-detection, error-reporting (through the *functionality* subordinate quality attribute *visibility*), error-resolution. The previous functionality deals only with situations where the system is able to avoid failure (that is , the system exhibits the behaviour, *fault-tolerence*). There is also another kind of functionality needed to recover/restart after failure of the software system (*recoverability*). Within each of these types of functionality there may be varying degrees of sophistication and completeness and alternative strategies for dealing with the situation.

Suitable metrics which characterise the reliability of a software system are the behaviours *maturity* and *availability*. Maturity may be interpreted as a running average of the elapsed-time between failures over the life-time of the system while availability is a running average of the down-time after each failure.



In this functional model fault-tolerance, for example abstracts over a number of other behaviours and software characteristics such as fault-diagnosis, reconfigurability and survivability (the behaviour where a system needs to continue operating when one or more of its sub-systems have ceased to function). In some applications a behaviour

like survivability may be such a critical quality requirement that there is justification for raising it to the same level as reliability even though technically it is just a very specialised form of reliability.

Below we provide a possible (but not definitive) instantiation of the quality model. It bears many similarities with ISO-9126 apart from the inclusion of *Reusability* at the top-level and the inclusion of additional subordinate properties. From *Functionality* down to *Maintainability* we are talking about a single software/hardware/system environment. Beyond that we have included quality attributes that reflect considerations for use in other environments.

Functionality
- Correctness
- Security
- Interoperability
- Visibility

Reliability
- Availability
- Fault- tolerance
- Maturity
- Recoverability

Efficiency
- Processor economy
- Resource economy
- Communication economy
- Order behaviour
- Throughput

Usability
- Learnability
- Operability

Maintainability
- Analysability
- Modifiability
- Testability

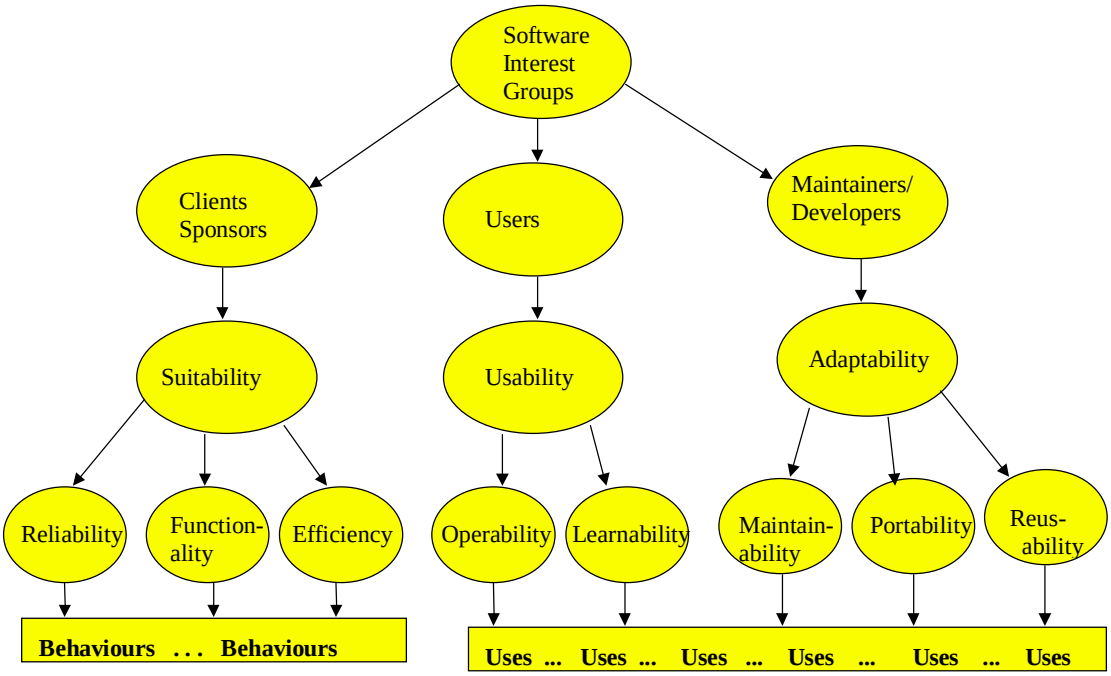
Portability
- Machine Independence
- System Independence
- Replaceability
- Installability
- Data commonality

Reusability
- Representation Independence
- Application Independence
- Data Encapsulation
- Function Encapsulation
- Interfaceability

We may now summarise much of what we have established thus far in our goal-oriented approach to constructing a model for software product quality. We start out with the principal interest groups then look for what are their primary quality requirements. Settling the quality needs of the different interest groups is at best an informal process. One possible, although defensible high-level needs specification is: *suitability* (or *fitness-for-purpose* which is characterised by a set of behaviours of the software), *usability*, and *adaptability* (which is characterised by a set of uses of software by different interest groups). Below we show a mapping onto a modified/extended version of ISO-9126 [7]. The primary interests of clients/sponsors are in the suitability of a software system for its intended purposes. The behaviours, *functionality*, *reliability* and *efficiency* provide an accurate high-level characterisation

of *suitability*. In contrast users and developers/maintainers are interested in the usability and adaptability of a software system which reflect the different uses of the system. The high-level uses, *usability*, *portability*, *maintainability* and *reuse* often represent priority quality needs of these interest groups. Of course, sponsors may also be interested in maintainability, etc. In some instances, at least, this will be a secondary quality need for this interest group, and so on

GOAL-DIRECTED SPQ MODEL DESIGN



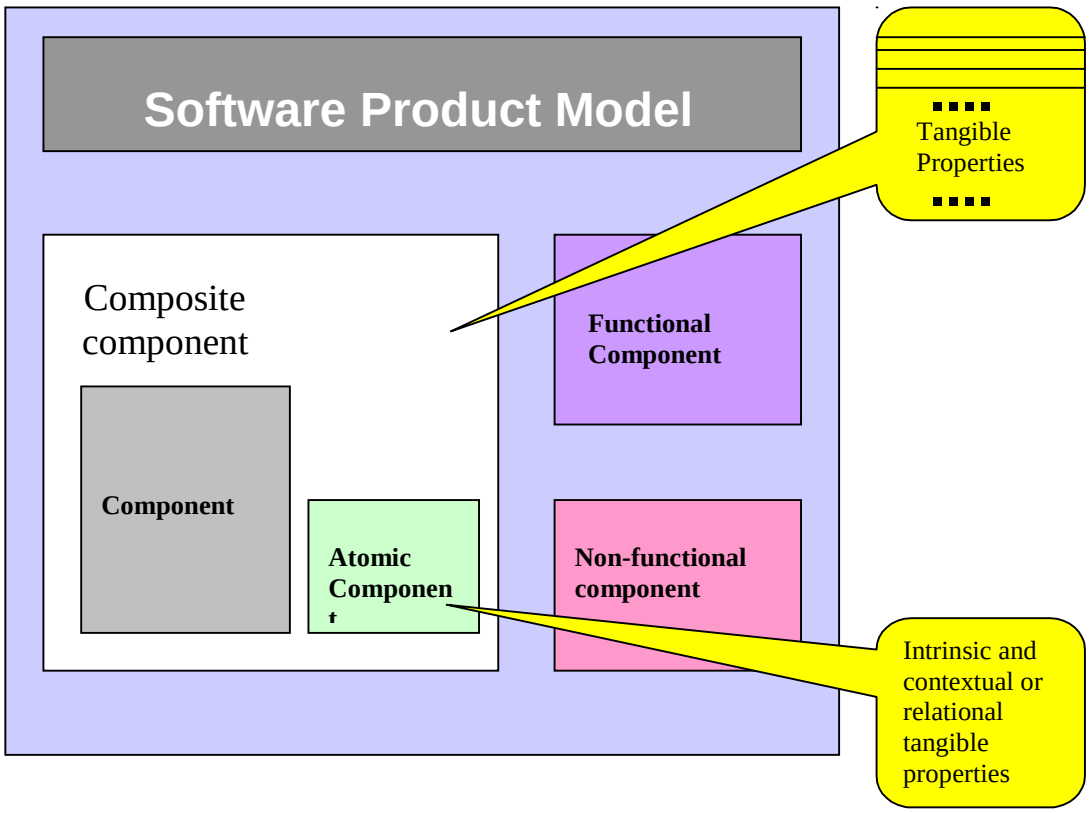
Quality models like this are never absolute or fixed, either in terms of the chosen primary interest groups or in terms of the set of high-level quality attributes that we settle upon. If we had chosen to include *auditors* as a primary interest group then we would have had to add the high-level quality attribute, *verifiability* which is a high-level use of software. The point we are trying to make here is that quality needs vary in different contexts. However the framework that we are proposing is robust and flexible enough to accommodate variability, change and refinement.

We have now spelled out enough about quality models to be ready to tackle the second task of constructing a software model.

4.2 Software Product Model

At the heart of constructing a software product model are two fundamental assumptions:

- Software is composed of both simple or *atomic components* and *composite components.*
- Software components of various kinds exhibit *tangible properties* that impact the quality of software.

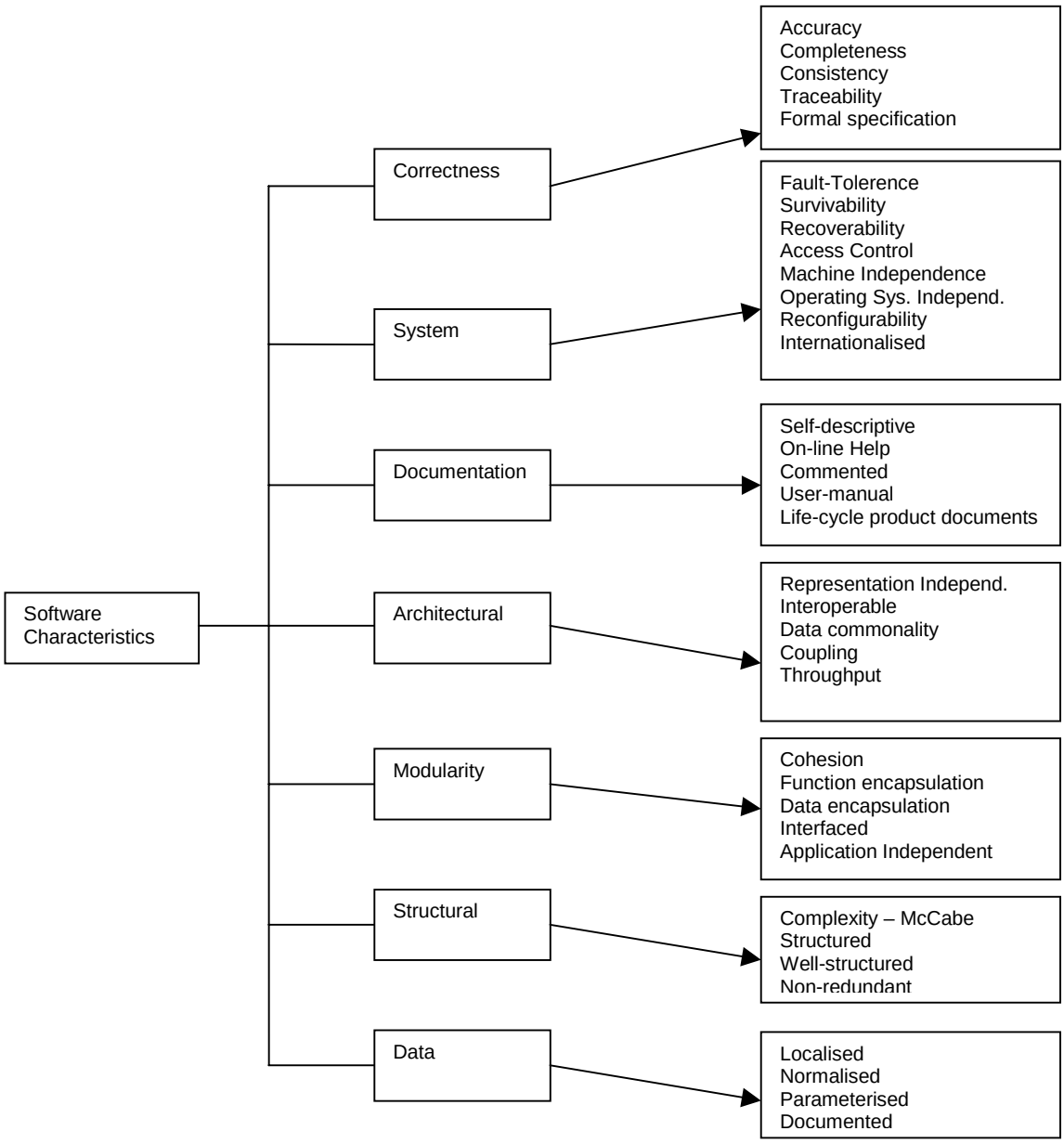


The following set of statements assist us in constructing a software product model:

- Components include different types of statements which are in turn composed of lower-level components such as variables, expressions and guards, etc.
- Modules are components. It is convenient to treat different kinds of modules as separate components.
- Different kinds of simple and composite data that flows between modules and sub-systems, and across interfaces are treated as distinct components.
- Software components exhibit concrete properties (including metrics) that characterise the nature of software, its form, and its structure.
- Software exhibits behaviours and is open to uses – both these sets of properties are abstract
- We choose to associate abstract properties with software, called software characteristics, in order to provide high-level descriptions of its nature, its behaviour, its uses, its support for change, its form and its structure. Examples are: machine-independence, modularity and correctness. Some software characteristics can be associated with software components.

- Software components exhibit concrete properties that contribute to, or determine their software characteristics
- Software characteristics can be either generic, product-specific, language-specific, or domain-specific
- Tangible quality-carrying properties of software can be either generic, product-specific, language-specific or domain-specific, functional or non-functional. Tangible properties can also be either intrinsic/internal or contextual/relational
- We associate abstract properties, called quality attributes, with software to characterise its desirable behaviours and uses.

Below we will suggest one possible classification scheme for software characteristics. In the interests of building a comprehensive model for software product quality it is useful to have a framework into which software characteristics and their defining tangible properties can be fitted. Such a framework can help us systematise and structure our knowledge about software. It can also point to areas where there are gaps, misclassification and other problems in our knowledge and understanding of software characteristics. Over time we can correct, refine and improve the framework to more accurately suit our needs.



4.3 Linkage Model – Axioms of Software Product Quality

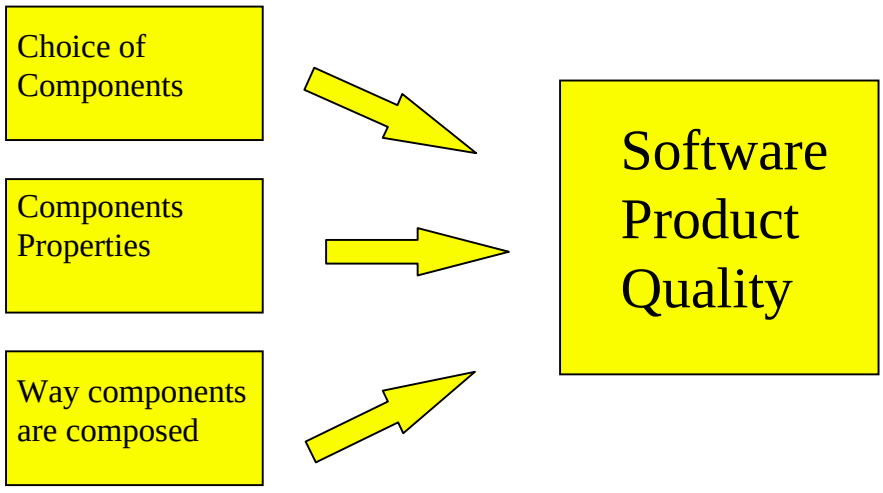
Now we have models for quality and software the task that remains is to link these two models to create a model for software product quality. In tackling this task we need to answer the fundamental question:

Question:
What is it about software that determines its quality?

Our response to this question is embodied in the following set of axioms which form the building blocks for a linkage model and empirical theory of software product quality.

First Axiom of Software Product Quality:
If we admit that software is composed of components then the choice of those *components*, their *tangible intrinsic and contextual properties*, and the way those components are *composed*, determines the *quality* of software.

This axiom may be represented schematically as shown below.



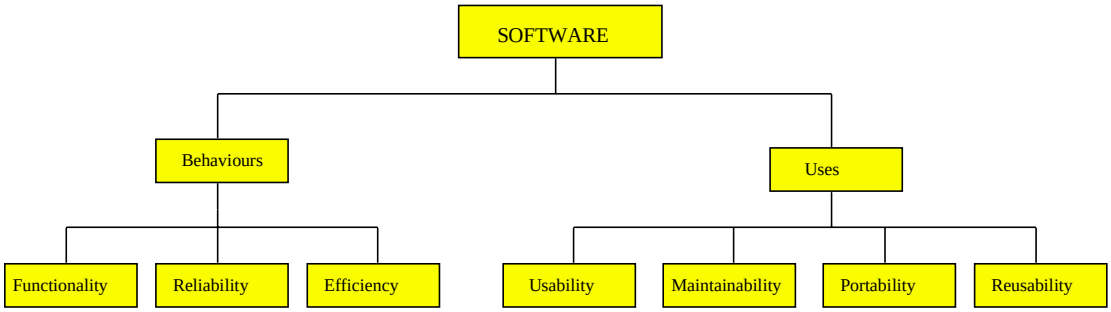
Example:
At a low-level, variables, types, expressions and loops, etc are examples of components. At a higher level access control modules, software-hardware interface modules and communication data are examples of composite components.

The second axiom stems from an observation linking behaviours, uses and quality attributes.

Second Axiom of Software Product Quality:
Software exhibits a set of quality attributes and it exhibits certain observable *behaviours* and *uses* that correspond directly to its quality attributes.

Our knowledge of software and experience with its use allows us to instantiate such a framework.

Example:
One possible instantiation, an augmentation of the ISO-9126 Standard[7], was shown earlier and is again shown below.



Third Axiom of Software Product Quality
Tangible quality-carrying properties of software components contribute to one or more intangible, high-level quality attributes of software.

Example:
Functionality that checks whether a parameter, which is used as a divisor, is non-zero is an example of a tangible quality-carrying property which contributes to the *fault-tolerance* and hence the *reliability* of a given software system.

Fourth Axiom of Software Product Quality
Associated with each tangible quality-carrying property of a component is a *verifiable empirical statement* that links the property either to a software characteristic, a behaviour or a use and then to a high-level quality attribute.

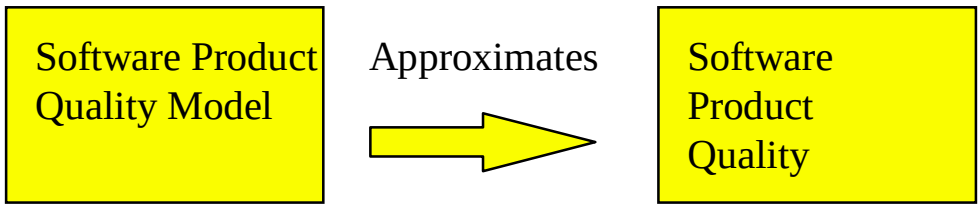
Example:
An example of a verifiable empirical statement is:

> *Self-descriptive identifiers contribute to the analysability and hence the maintaintability of software.*

Most software engineers, through experience accept the truth of this statement. We may however design a number of possible experiments to test its validity. We could, for example, do this by providing two implementations of a given self-contained module that performs a well-defined function. One implementation would use accurate, meaningful, self-descriptive names for all identifiers used in the module; the second would use identifiers such as x, y, etc, which give no clues as to their purpose. We could then give two groups with comparable backgrounds the two modules and test things like how long they took to work out what the module does, what the role of each variable was, etc.

What is important about this fourth axiom is that it ensures the present proposal for software product quality is ultimately built on a set of empirical verifiable statements. It does not go so far as to demand that such statements have already been verified but there is always the option to do so. Hence there is the opportunity either to validate or to refute and correct any or all the quality claims that contribute to the model.

As we have already seen the choice of components, component properties and the way components are composed determines the quality of software. Any such proposal for a software product quality model will always only ever be an approximation to software product quality.



The challenge in building a software product quality model is to align the properties associated with software's nature (its software characteristics) with the properties that describe its desirable behaviours, ease-of-use and response to change (its quality attributes). In creating a linkage model we need to satisfy the following requirements:

- We need to satisfy the principle of correspondence for the quality model by mapping quality properties onto components.
- We need to make tangible quality-carrying properties component-based and accumulative in their affect across a software system
- There is not a one-to-one mapping between quality determinables and software determinables. For example, the software characteristic, modularity can contribute to the quality attribute maintainability and to say reliability (by providing modular protection)
- A challenge is to identify a comprehensive generic set of quality-carrying properties for each software characteristic and to anchor these to components.
- Another challenge is to identify a comprehensive set of quality-carrying properties for each component. Such properties will contribute to different quality attributes.
- We may expect that for some application domains there will be additional domain-specific quality-carrying properties that have a significant impact on the quality of software
- There is tangible functionality and tangible non-functional properties of software that contribute to its quality.

5. Software Product Quality Model Implementation

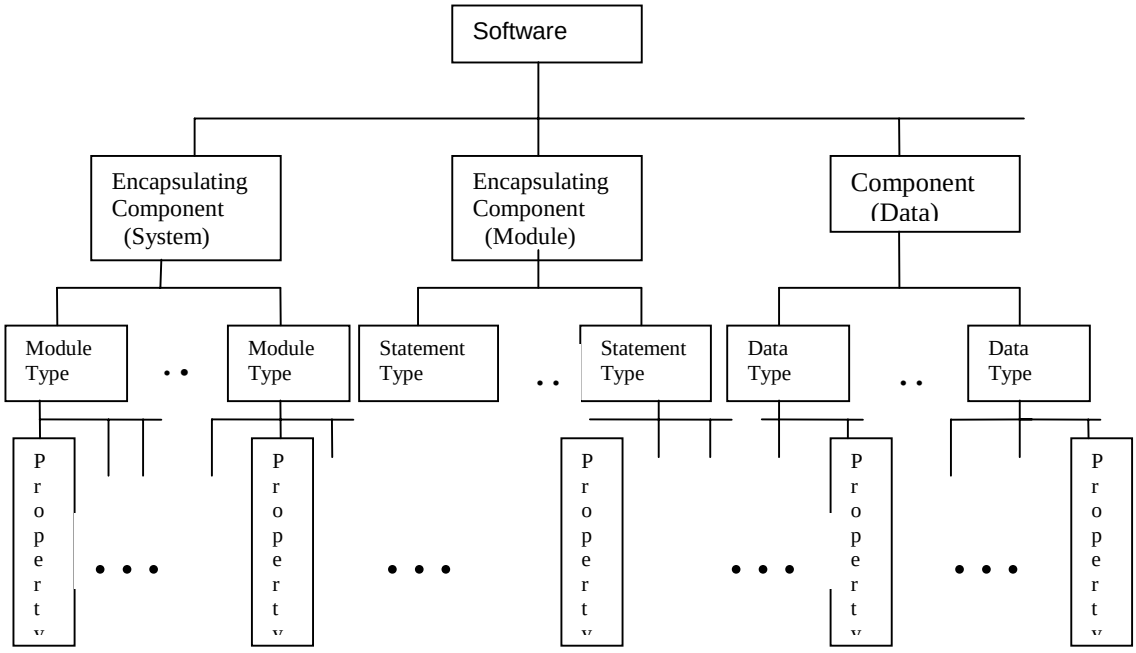
There have been two main approaches to implementing software product quality models (a) a rule-based approach (b) an approach that tries to define software characteristics then qualify these definitions with sets of rules and properties.

The first of these approaches involves compiling large numbers of ad hoc suggestions and rules of thumb. Unfortunately this approach has not proved all that successful. The problem arises because we are talking about potentially many hundreds, at least, of disparate properties that may be used to characterise high-level quality attributes. Usually such compilations are not classified systematically and there is no link between the rule/property and any corresponding high-level quality attributes.

The second approach at least implied, if not followed, in trying to define software product quality has been to provide a high level characterisation using quality attributes and software characteristics and then to proceed to identify a rich and accurate set of tangible quality-carrying properties that help define software characteristics like fault-tolerance, machine-independence, and so on [2, 7]. There is no doubt that information arranged in this way is certainly helpful but it does not directly link to components (this issue is addressed in section 5.3).

Information about quality arranged in either of these ways is not very convenient for meeting the requirements we have set for a software product quality model. That is, none of the software interest groups benefit very much from these characterisations of software product quality. What is needed are better ways of structuring the information about software product quality. The Axioms of Software Product Quality and the accompanying models for software, quality, and linkage that we have sketched suggest a more practical way to proceed. What this entails is to focus on the different types of components that we find at various levels in software. As our earlier discussion suggests we may associate various intrinsic and contextual or relational quality-carrying properties with each type of component.

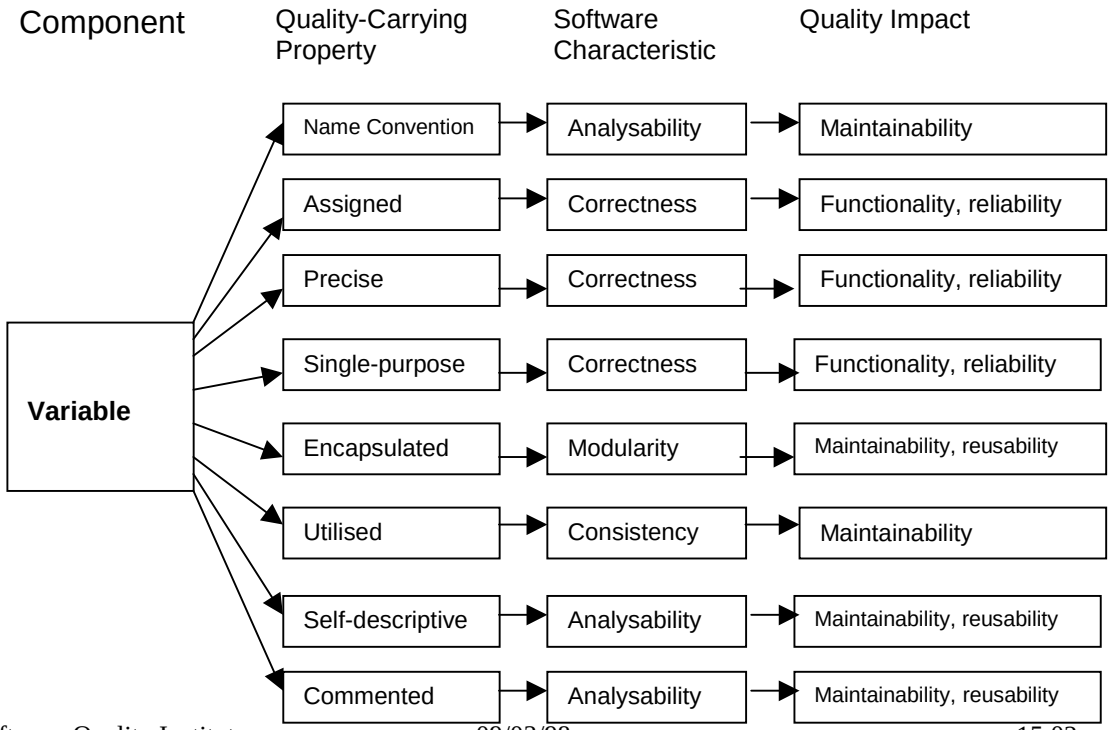
Our diagram below suggests one method for mapping tangible quality-carrying properties of software onto various classes of software components that make up a software system. While these properties map directly to this component framework they have their own separate classifications relative to the software characteristics that contribute to quality attributes.



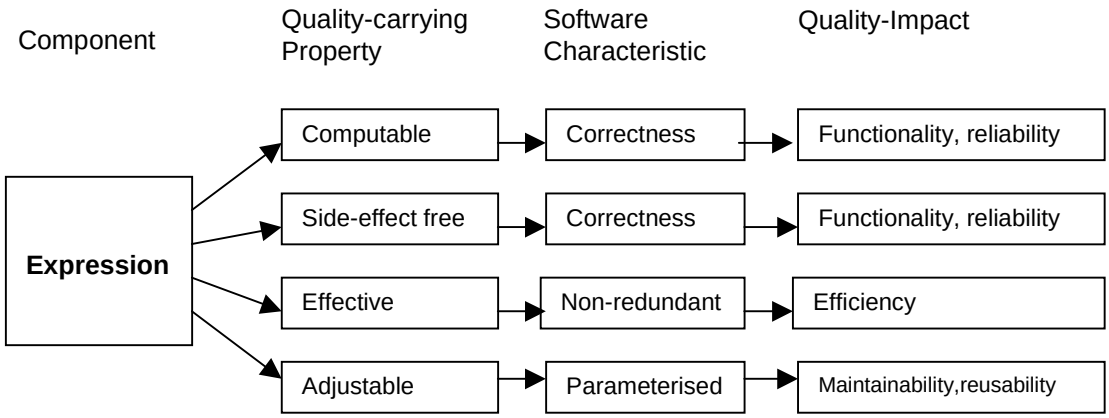
To use an extended version of this framework to implement a model for software product quality we must systematically work through the characterisation of components at the various levels. We will now visit these different levels.

5.1 Statement Component Level

The lowest level in our software product quality model is made up of the components that are used to build statements. The grammar associated with a given programming language tells us precisely what are the different types of statement components (e.g. guards, expressions) and types of data (e.g. variables, constants, records, etc). We may associate a set of quality-carrying properties with each type of statement component and data. We may also link each quality-carrying property to its parent software characteristic and/or high-level quality attribute. Characterisations for the statement components, *variable* and *expression* are shown below. A more detailed treatment of these components, including a discussion of their various quality-carrying properties is given elsewhere [9].

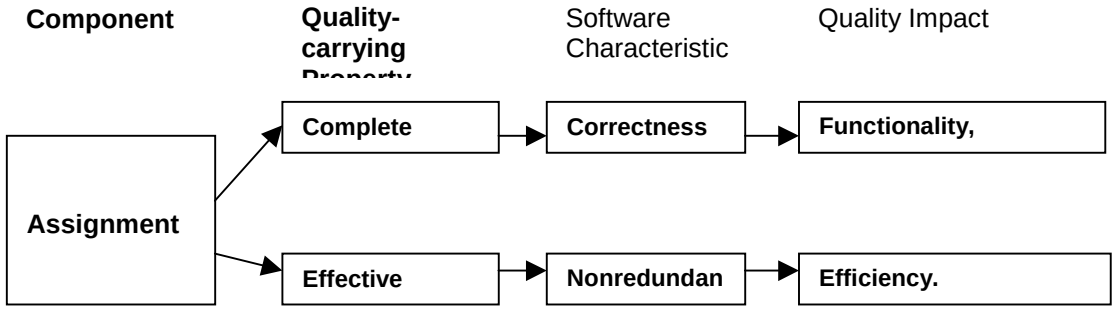


Expressions are made up of variables, constants and operators. Expressions exhibit separate properties from their constituent components.



5.2 Statement Level

The statement level in our software product quality model is made up of the statements that are used to build modules. Each statement is built from elements defined at the statement component level. Again the grammar associated with a given programming language tells us precisely what are the different types of statement. Examples in most languages include statements such as assignments, if-statements, loops, etc. As at the statement component level, a statement exhibits quite distinct properties from its constituent components. For example the quality-carrying properties of an assignment statement (below) are quite distinct from its constituent components – the expression and variables.



It is important to understand how this model works and how defects are attributed to components. Consider an assignment statement where one of the variables in its expression has not been assigned. The question arises is this an assignment defect, an expression defect or a variable defect? *The convention applied in this model is that defects are always attributed to the most deeply nested or lowest level component with which they are associated*. Using this convention such a defect obviously violates a variable quality-carrying property (i.e, that variables must be *assigned* before use). Most of the defects associated with assignment statements are either variable defects or expression defects. The only problems with an assignment itself are that it must be effective. That is, it must change the state of the computation and it must be complete – i.e. it must avoid statements like x := x which are not properly formed.

Elsewhere [8,9,14] we have illustrated in detail how this quality model specification may be done systematically for a range of different statements and types of data that occur in most conventional imperative programming languages at the intra-module level. Such a proposal is useful for dealing with a lot of generic quality issues at the module/object/structured programming level.

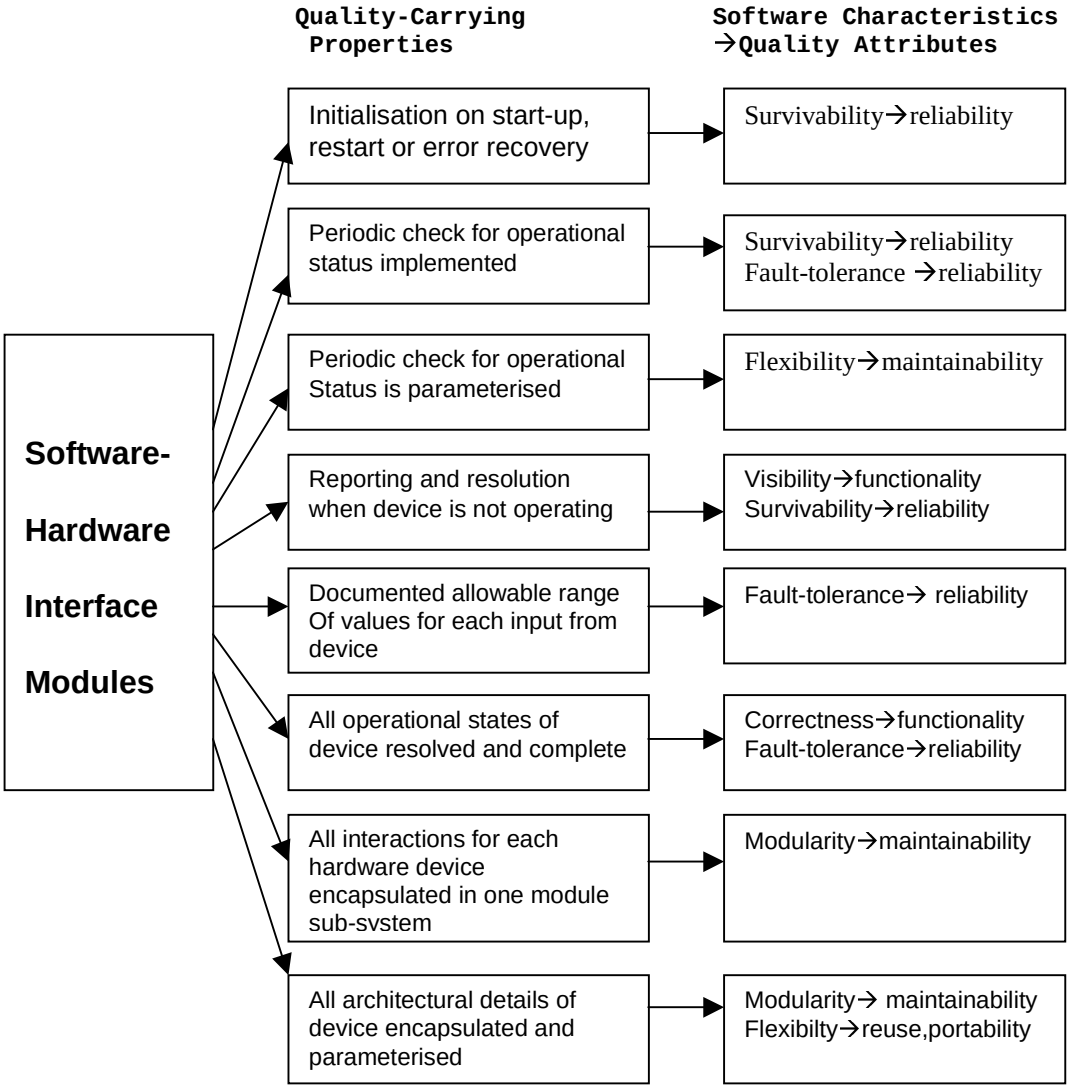
5.3 Module Level

At the module level the quality issues associated with components vary greatly depending on the nature of the application. For example, the quality requirements for a *transponder control module* are entirely different from *a password authentication module.* Therefore, if we try to apply the same strategy as was applied at the statement component and the statement levels to deal with the major quality issues associated with modules we end up with a very large number of properties associated with the single component type module. *The problem is that only a small number of these properties are relevant to any given application.* A practical way to overcome this problem is to first classify modules into a relatively small number of generic categories. A sample set of generic module classifications is given below.

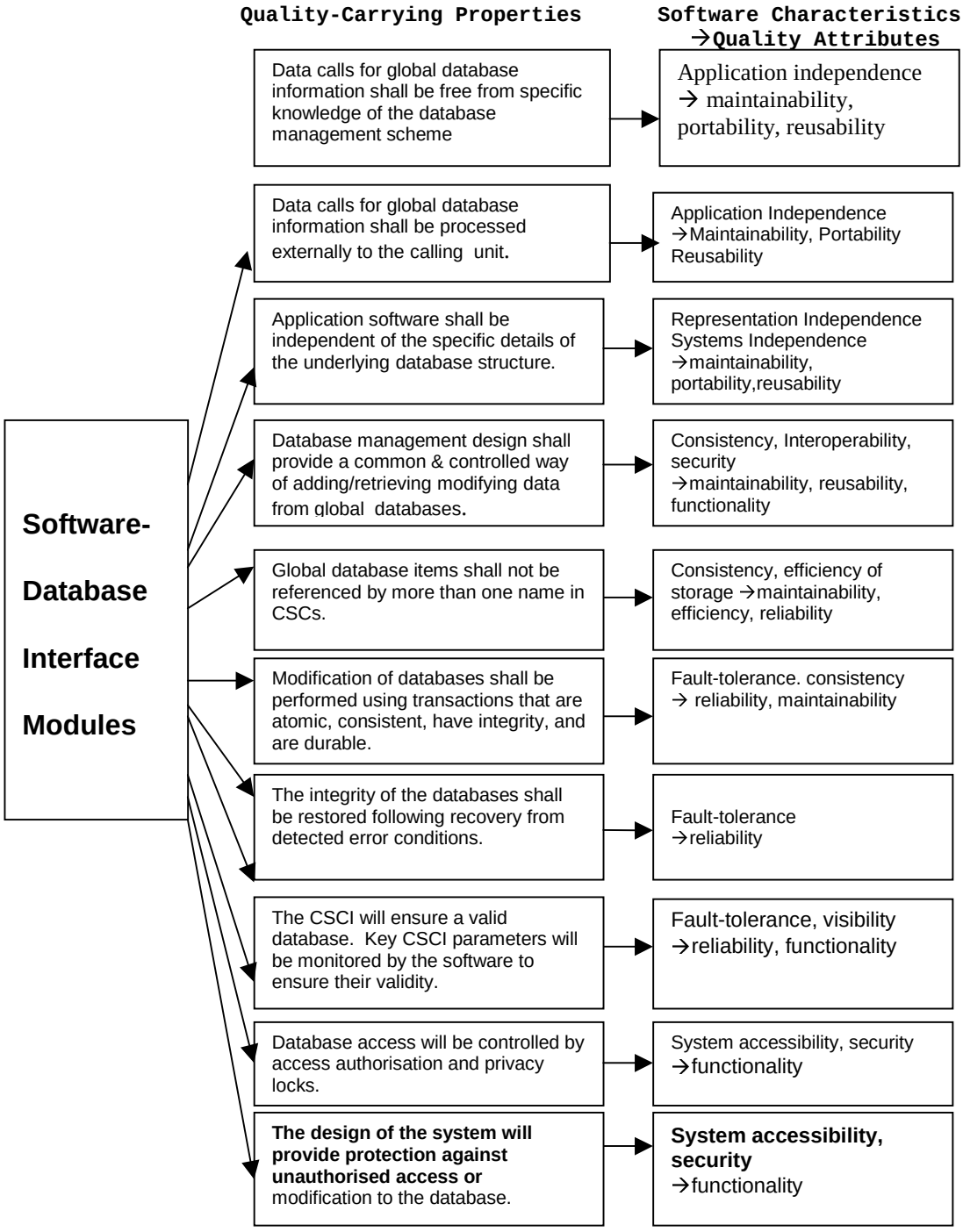
Sample Generic Module Classifications	
Software-Hardware Interface Module	Computation/calculation Module
Software System Interface Module	Software-Database Interface Module
File-Processing Module	Control Module
Operator Interface Module	Retrieve Module
Search Module	Edit Module
Data Structure Construction	Data Transformation
Analyse(Decision) Module	Select/Choose Module
Initialise Module	Data Modification Module

We may then associate the relevant quality-carrying properties (usually a small number) with each generic module type (see the two examples below)

Software-Hardware Interface Module



Software-Database Interface Module



This framework may be qualified/extended by including *application-specific modules* within a given generic category. For example, a *transponder* module belongs to the generic *software-hardware interface* module category. Within the generic categories we try to specify a set of quality-carrying properties that are likely to apply to modules that fall within that umbrella.

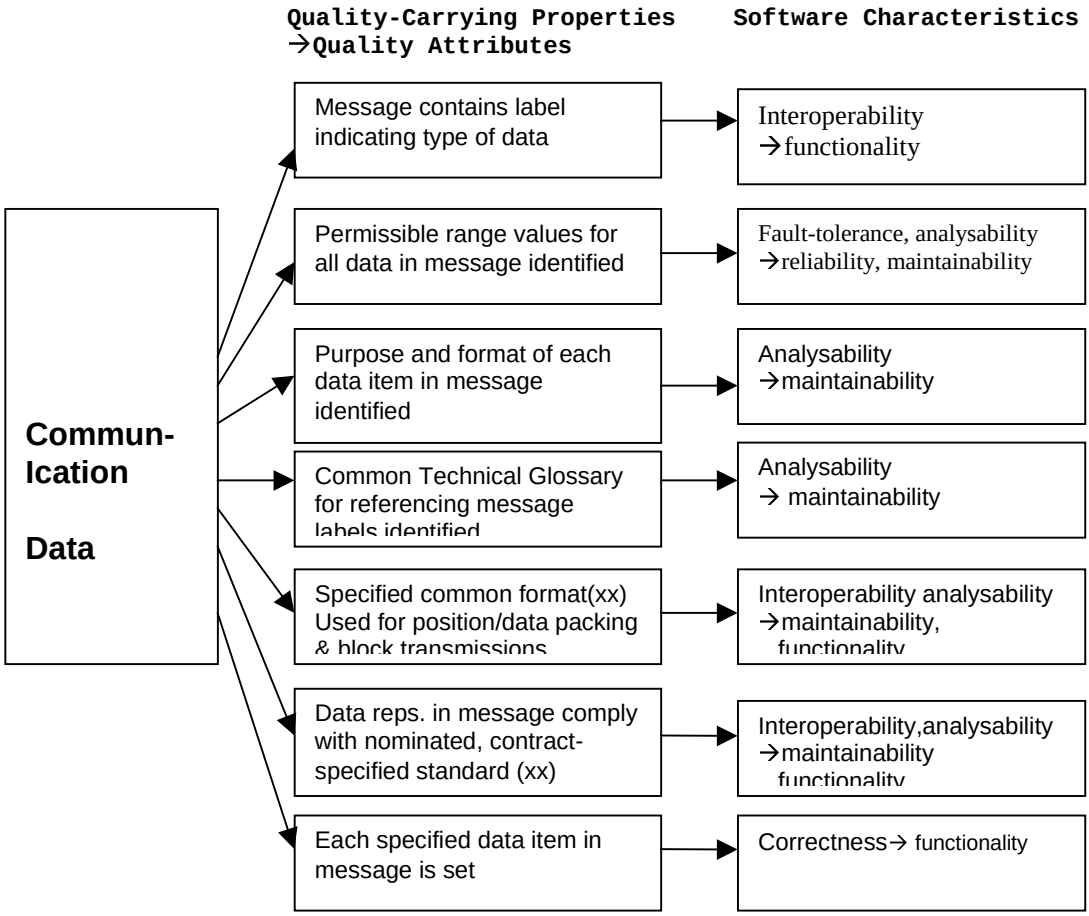
Just as modules can be classified into a small set of generic components so too can composite data sets (sample data classifications are given below).

Sample Generic Data Classifications

Communication Data (messages)
Composite Data
Computation Data
Data structures
Operator Input Data
Security Data
Transaction Data
Concurrently Shared Data
Meta-level Data
Pointer/Addressing Data

As for modules it is possible to associate a set of quality-carrying properties, etc with each generic composite data entity (see example below for communication data). Other types of composite data can also be treated similarly.

Communication Data



Elsewhere we have provided a comprehensive process for module classification [10]. This whole proposal for module categorisation has two advantages. First, modules in a system can be put into these different high-level and qualified categories and then tangible properties (including functionality to satisfy the quality requirements) can be assigned.

Software engineers can use such a model, constructively, and in a practical way. They first classify the type of module then build in its particular set of tangible properties that are needed to satisfy its quality requirements. These properties often have functional components to ensure such things as *fault-tolerance* and *survivability*, etc. It is also easier for auditors and quality personnel that are involved in verification. They just look at the module classification and check for the small number of specialised generic quality requirements for the particular module type. If we can get to this mode of operation then we will be well-placed to engineer in tangible properties that deliver particular high-level quality attributes. There is also the possibility when dealing with code, rather than other software products, to automate some of the checking using static analysers. We claim that this whole approach to software product quality is more constructive, intellectually manageable, more effective, verifiable and more practical than existing practices.

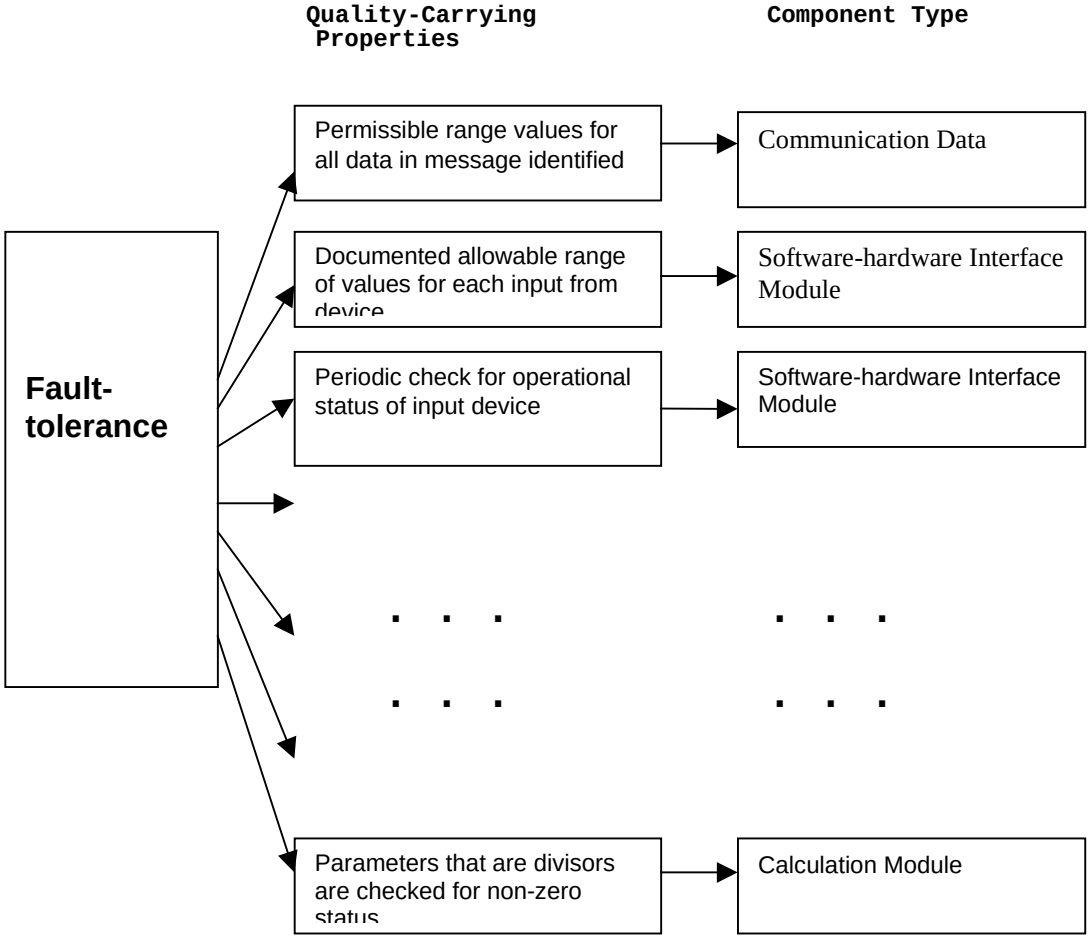
Assuming, in any implementation of this classification scheme, we limit the different types of generic modules and generic data to only a small number it is likely that such a model will only be strong enough to capture a generic set of properties for each module and data type. For example, we may have a set of generic tangible quality-carrying properties for say, *software-hardware* interface modules (see above). However when we are dealing with a specific type of hardware device, for example, a transponder, or a mass spectrometer, we may expect that there will be an additional set of *domain* or *application-specific tangible properties* that need to be satisfied to meet the quality requirements of the particular software-hardware interface. Hence there is a need for the module classification framework to be hierarchical. The same sort of reasoning applies for data and architectures. Elsewhere[10] we have set up descriptions for other types of modules and data. Each of these different types of module and data can be specified in a similar way to the examples above. We are currently carrying out this exercise as a basis for building up a comprehensive model for software product quality. A very good starting set of quality-carrying properties can be found in the appendix of reference [2]

5.4 Defining Software Characteristics

What we have presented thus far is a framework that allows us to specify sets of tangible quality-carrying properties that are chosen to ensure that a range of high-level quality requirements are satisfied for chosen sets of statement, module and data components. As a quality model this is good as far as it goes. Unfortunately it does not go quite far enough. We find that clients, software engineers and others often express at least some of their quality requirements by identifying certain software characteristics that need to be satisfied by the software and hence by its modules, data and other components.

For example, we are interested in being able to answer the question what tangible properties does a module need for it to be fault-tolerant, interoperable, reusable, etc. That is, we want to talk about fault-tolerant modules, interoperable modules and so on. Then we want to be able to ansewer the question, what tangible quality-carrying properties does a module (or other component) need to possess for it to be fault-tolerant. This requirement may be above and beyond what is needed for a module to satisfy the quality requirements of a software-database module, etc and so on. We can handle this requirement using a software characteristic-quality carrying property-component framework. An example is given below for fault tolerance.

Fault-tolerance



By introducing this framework what we are doing in effect is providing a means for incrementally defining software characteristics like fault-tolerance by identifying a set of tangible determinates of that determinable and linking them to the relevant components. This process needs to be done systematically for each of the software characteristics in our model. A useful selection of such determinates can be found in Deutsch and Willis[2]. Our task when specifying the quality requirements of a particular classified module is then to consult the properties associated with the relevant software characteristics and select those tangible properties that need to be included in the module-type being specified. The information included in this arrangement of quality-carrying properties is the same as that used in specifying component properties – the only difference is the perspective and concentration.

6. Evaluation And Comparison With Other Models

Work by Kitchenham [1] has sought to use a systematic approach to evaluate, compare, rationalise and refine earlier software product quality model proposals. Noting Kitchenham and Pfleeger's [1,3] criticisms, we have sought to build on and refine earlier proposals using a constructive, systematic approach.

In evaluating and comparing the different models for software product quality we will use Garvin's criteria[6] together with the following questions as criteria.

- What guiding principles were used to choose the set of high-level quality attributes
- What criteria are used to choose and define software characteristics/defining behaviours and uses
- What architecture and implementation strategy is used for the model
- Can the model be validated.

6.1 Evaluation Against Garvin's Criteria

Earlier we made the suggestion that Garvin's five perspectives could be used as criteria for evaluating a software product quality model. We will now argue that the model we have proposed accommodates all of Garvin's criteria.

Transcendental Criterion: Let us start with the transcendental view of quality. This is accommodated because there is always scope to add additional quality-carrying properties to more accurately characterise a particular software characteristic. Similarly, it is also possible to add more determinates to more accurately define a given quality attribute. That is, the model has an open framework that will allow it to evolve and strengthen as our understanding of software product quality grows. In addition the model supports correction and refinement and the incorporation of domain-specific knowledge.

User's Criterion: A second perspective of Garvin's is that quality software should meet the *user's view* or *fitness-for-purpose*. Our response here is that the present model certainly does this. In fact it goes further than Garvin by accommodating the needs of a number of different interest groups (clients, sponsors, users, maintainers, verifiers, etc) in a direct way.

Manufacturing Criterion: We would argue that our model is also rich enough to meet the needs of the *manufacturing view* which stipulates that quality means conformance to a specification. The important thing here is to be able to specify those quality requirements to what ever degree of specificity is appropriate. With the present model it is possible to specify quality requirements in terms of high-level quality attributes, or to take the specification down to the software characteristics level. And, if that is not enough, the specification can be taken right down to the listing of quality-carrying properties which imply higher level software characteristics and quality attributes.

Product Criterion: The product view of quality focuses on the internal properties of software components. These internal properties correspond to the quality-carrying properties in the present model.

Value-based Criterion: The framework and model components that satisfy the previous four criteria provide a rich but open-ended description and model for software product quality. If the customer is willing to pay then this model can be exploited and if necessary expanded to achieve the desired level of quality. For example, NASA were willing to pay $1000 per line (20 – 100 times the average cost) to ensure the on-board software for the Space Shuttle met its extremely demanding reliability and other quality requirements. What is important to recognise here is that the *processes, people and tools* used to specify, construct, verify, test and quality assure software may change dramatically if there are stringent quality requirements and the customer is willing to pay to ensure that they are met.

6.2 Selection Of High Level Quality Attributes

The original Bauhaus principles of good design [11,12] provide a useful starting point for selecting high-level quality attributes. *Fitness-for-purpose* is the Bauhaus principle that is most relevant for software. It demands or implies that software should exhibit behaviours and be amenable to uses that meet the needs of the different interest groups involved with software. Identifying the different interest groups and their particular needs has played an important part in influencing our selection of a set of high-level quality attributes. *Suitability, usability* and *adaptability* are the highest-level quality attributes that probably best capture the notion of fitness-for-purpose in the software context. There are always significant change pressures associated with software: the need to change functionality, the need to adapt to different hardware and software systems environments. In addition software must be easy to use and exhibit suitable functionality, reliability and performance. We have sketched these ideas earlier in diagram form.

The different quality models have advocated somewhere between six and twenty quality attributes at the highest level. Miller's well-known experimental result concerning human short-term memory (people can remember seven plus or minus two items in a given category) has been an important guiding principle in choosing the high-level quality attributes. Like most of the other quality models we have concentrated on quality attributes that pertain to software designs and implementations but not requirements or other software products (elsewhere we have dealt with requirements[9]). The other important principle we have used to select quality attributes has been to ask the question "is a given candidate attribute a *behaviour* or a *use* according to our definitions of these two terms?". Models like McCall's violate these selection criteria. McCall's model [1,3] has eleven high-level quality attributes. It also includes properties like correctness which is clearly a product property (software characteristic) rather than a quality attribute. The international standard, ISO-9126 [7] conforms better to these selection criteria.

6.3 Principles For Definition

The definitions we have used for quality attributes, software characteristics, and their defining terms are all essentially based on causal relations. For example fault-tolerance "causes" or contributes to or is a manifestation of the behaviour**, reliability**. Similarly, *application independence* "causes" or facilitates the use, **reusability**; *machine independence* contributes to the use, **portability**, and so on. Definitions based on causal relations are widely used in many disciplines and are advocated in the classic work on the theory of definition[13]. In formulating definitions we have employed both top-down and bottom-up strategies. Subordinate behaviours and uses and contributing software characteristics are employed. Our approach to defining software characteristics (or subordinate behaviours) like fault-tolerance is to say that such terms are best characterised by the accumulated set of tangible product properties that contribute to fault-tolerance. Most of the other models are silent on the underlying strategies that they have employed to construct definitions for quality attributes, and so on.

The incentive to satisfy Miller's requirement has forced us to treat behaviours like *survivability*, and others which appear as top-level quality attributes in some models as subordinate properties – this has hence influenced the definition construction process. Take the case of survivability. It is easy to reason that survivability is a specialised kind of reliability – the ability to operate when anomalous conditions apply (i.e, when part of the overall system is not functioning). Similar sorts of arguments can be used for other behaviours and uses.

6.4 Architecture And Implementation Strategy

Most of the Software Product Quality models that have been proposed have used either conventional top-down hierarchies (e.g. ISO-9126) or bi-directional hierarchies (McCall's model) for characterising quality. As a general rule, bi-directional hierarchies, because of their complexity, should be avoided. The fact that top-down hierarchies exist which express essentially the same information suggests that bi-directional models are unnecessary.

The architectures of most models only go as far as providing an architecture for quality. They neglect to include a model for software or any sort of prescription for how quality attributes can be mapped on to software or how tangible software properties can be mapped to quality attributes. As a result, there is a gulf between high-level quality attributes and the myriad of tangible software properties that influence the quality of software. This lack of structure hampers our efforts to make software product quality intellectually manageable, implementable and applicable. Our architectural proposals for software product quality have been designed to overcome what we see as deficiencies in earlier models.

With regard to an implementation strategy most existing models are either relatively weak or silent. ISO-9126 advocates the use of quality "indicators" but leaves it at that. In contrast the present model provides considerable implementation detail.

6.5 Scope For Model Validation

For a model to be validated it must possess a set of verifiable statements. The statement form we have chosen which involves a tangible property that is in turn linked to a software characteristic or quality attribute is open to empirical verification by use of appropriately designed experiments. Other software product quality models do not contain an extensive set of verifiable statements that involve tangible product properties. We claim this feature is an absolute essential of any comprehensive model of software product quality.

Acknowledgements

I would like to thank Ray Offen and my colleagues at the SQI for helpful comments on this work.

Conclusion

A framework that may be used to characterise software product quality has been presented. The framework has been developed in a goal-directed way in order to meet the needs of the different interest groups associated with software. The most significant advantages of the approach we have taken is that it allows the problem of software product quality to be broken down into intellectually manageable components. A component-based implementation approach enables the information to be organised in a way that makes it easy to understand, test, correct, refine, extend and apply. Another important advantage of the model is that it has enough structure to deal with software product quality for large and complex projects.

References

1. Kitchenham, B., (1987) Towards A Constructive Quality Model, Soft. Eng. Journal, pp105-112.
2. Deutsch, M, Willis, R. (1988) Software Quality Engineering, Prentice-Hall, Englewood Cliffs, New Jersey.
3. Kitchenham, B., Pfleeger, S.L., (1996) Software Quality: The Elusive Target, IEEE Software, vol. 13(1), pp. 12-21.
4. Russell, B.,(1917) Mysticism and Logic, Routledge and Kegan, London
5. Price, H.H., (1969) Thinking and Experience, 3rd Edition, Hutchinson, London.
6. Garvin, D. (1984) What Does “Product Quality” Really Mean?, Sloan Management Review, Fall 1984, pp. 25-45.
7. Software Product Evaluation – Quality Characteristics And Guidelines For Their Use, ISO/IEC Standard, ISO-9126, Int’l Organization For Standardization, Geneva, 1991 and Revised Draft, February 1997.
8. Dromey, R.G., (1995) A Model For Software Product Quality, IEEE Trans. Soft. Eng., pp 146-162
9. Dromey, R.G (1996) Cornering The Chimera, IEEE Software, Vo1.3(1) pp 33-43.
10. Dromey, R.G. (1997) Software Product Quality For Large Projects, (unpublished results)
11. Dromey,R.G, McGettrick, A.D (1992) On Specifying Software Quality, Software Quality Journal, vol 1(1), pp. 45-74.
12. Dodd, T. (1978) Design and Technology in the School Curriculum, Hodder and Stoughton, London
13. Ogden, C.K., Richards, I.A., (1969) The Meaning of Meaning, Routledge and Kegan Paul, London.
14. Pan, S., R.G.Dromey, Beyond Structured Programming, ICSE-17, Berlin, 1996.