OOSC2 Reviews

Design By Contract: A Missing Link In The Quest For Quality Software

Todd Plessel

plessel@vislab.epa.gov

Lockheed Martin / US EPA

August 11, 1998

Table of Contents

1. Abstract

This paper considers software correctness - what is is, how it relates to other software product quality factors, and the effectiveness of various techniques aimed at achieving it. The indirect trade-offs between correctness and other quality factors result from competition for human resources during various development phases aimed at addressing particular quality factors. This competition is due in large part to the sub-optimal human-intensive techniques employed to address correctness. Such current techniques are much less effective than they could be because they lack of an essential ingredient: the contract.

Meyer's Design By Contract is presented as a high-yield technique with a low-effort and low-trade-off requirement [1]. It is argued that Design By Contract not only improves the effectiveness of other techniques aimed a verifying correctness, but is, in fact, a necessary condition for correctness. The concept is explained and illustrated with examples in various programming languages from the past, present and future.

2. Software Product Quality Factors

2.1 Quality Is Everything Because Everything Is Quality

What is software quality? There are various definitions offered by researchers and practitioners have divergent views on the subject. Some focus on process - the human activity associated with software development, while others focus on the products - the artifacts that result from the process. There is surprisingly little consensus even on quality definitions with respect to the most important artifact: the source code.

Ask several people involved in software to define and rank the software qualities and you will find that each focuses on a few factors and lumps all the rest (if considered at all) under a vague notion of 'other concerns'. Programmer's views will differ from marketer's views which will differ from end-user's views. Some actively and earnestly pursue their notion of quality while others are content

to pay lip service to the idea or are even hostile to the concept. Without a comprehensive and shared view of quality it is no wonder that software is so often disappointing.

The first step in reconciling these different views is to encompass them all. Quality is everything because everything is quality. By 'quality is everything' we mean, if the quality of software is poor in any regard then it will taint everything that uses it with this poor quality. For example, if a software library contains memory corruption errors then every application that uses it will contain memory corruption errors. By 'everything is quality' we mean, every aspect of the software maps to one or more primary quality factors. Meyer provides a comprehensive set of primary quality factors that should cover everyone's viewpoint [1].

2.2 Meyer's Internal and External Software Product Quality Factors

Correctness

The ability of software to perform their exact tasks as defined by their specification.

This is presumably a concern of programmers.

Correctness can be argued as the most important quality. After all, if the software doesn't do what it is supposed to do then nothing else about it really matters. For all practical purposes it is useless. Of course, all software contains some correctness defects yet we continue to use it anyway so things aren't quite so simple. For example, having "bug-free" software available too late to be applied is also useless. Nonetheless, correctness and how to achieve (or at least 'improve' it) is a main focus of this paper.

Robustness

The ability of software systems to react appropriately to abnormal conditions.

This is closely related to correctness, but is different, as we shall see. Testers focus mostly on these two factors.

Efficiency

The ability of a software system to place as few demands as possible on hardware resources such as processor time, internal and external memories and communication bandwidth.

For some programmers (such as those creating esoteric HPF codes for custom supercomputers) this is the supreme goal and all else is a distant second or worse, perceived as detrimental to efficiency.

Ease of use

The ease with which people of various backgrounds and qualifications can learn to use software products for their intended role.

End-users are in the best position to judge this factor (and program authors the worst).

Functionality

The extent of possibilities provided by a system. The applicability of software to solve a range of problems.

This a primary concern for end-users and marketers of the software.

Timeliness

The ability of a software system to be made available when or before its users want it.

Again a major concern for marketing. (We must ship something - anything - to preempt the competition!)

Cost

The financial cost of developing or acquiring and using the software.

Development cost is an overriding concern to managers while end-user cost is a focus of marketers and users while programmers are oblivious.

Extendibility

The ease of adapting software to changes of specification.

This internal quality factor, like the rest, are a concern of developers - both original designers (hopefully) and 'maintenance programmers'. Maintenance programmers are in the best position to judge this quality.

Compatibility

The ease of combining software elements with others.

This is a concern of programmers and sometimes of end-users.

Portability

The ease of transferring software products to various hardware and software environments.

This is obviously a concern (bane) of maintenance programmers.

Reusability

The ability of software elements to serve for the construction of many different applications.

This is (should be) a concern of original developers, but maintenance programmers (attempting to reuse something) are in the best position to judge actual reusability.

Understandability

The ease with which a programmer can understand the source code of a software system.

Rarely a concern for original developers - who are not in a position to judge anyway, but perhaps the major concern of maintenance programmers.

Testability

The ease of testing a software system for correctness and robustness.

This is obviously a concern to testers. But often misunderstood and unappreciated by developers. (Cyclomatic complexity? Multiple-exit points? So what!)

Have we left anything out? Some will say "maintainability", "adaptability", "modularity", or "elegance". But these can be shown to be secondary qualities that derive from (map onto) several of the above primary factors. Others will offer: "accuracy" and "precision". But these are encompassed, with suitable specification, by correctness. For an in-depth analysis of such linking of qualities, see [15].

2.3 Tensions and Trade-offs

If we accept the completeness of the above definitions, the next step is to realize that they cannot all be optimized - there is tension between these factors. For example, functionality and timeliness are usually inversely related - it takes time to implement functionality - at least until reuse enters the picture.

Efficiency and portability are often inversely related - if software targets a specific platform capability, such as native application programmer interfaces (APIs) (e.g., Cray scientific subroutine library), then portability is lost or at least understandability is often severely impacted (#if _CRAY... #else...).

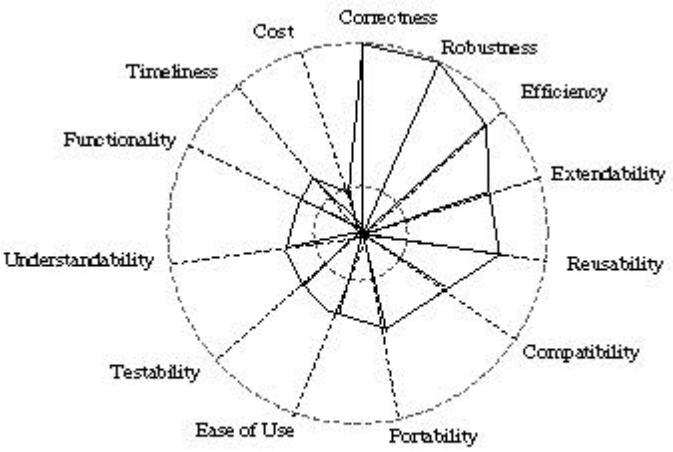
Extendibility can sometimes impact efficiency, e.g., added levels of indirection to yield flexible solutions has an inescapable overhead (e.g., nested virtual function dispatch, etc.). It is interesting to ponder the many trade-offs made - consciously or unconsciously - in actual software you may be involved in or familiar with.

Implicit assumptions are a source of many problems throughout each phase of software development - analysis, design, implementation, testing, maintenance, etc. This paper aims to demonstrate a technique that makes certain kinds of assumptions explicit so that development is a more conscious activity.

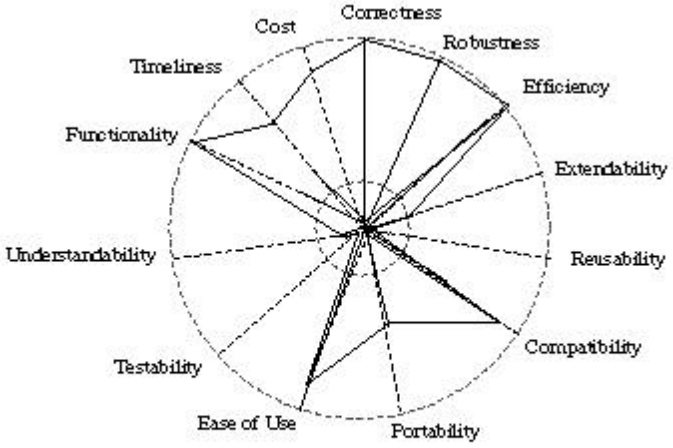
2.4 Quality Diagrams

These tensions can be illustrated by kivat diagrams, where each dimension is a spoke in a wheel. The outer circle is the idealized optimization of all dimensions. Of course, not every factor can be optimized. When some are optimized (pushed toward the outer circle) others are retracted. A small

round wheel represents the ultimate mediocre compromise - a little of everything yields not much of anything. It will satisfy no one. It is revealing to depict various viewpoints (e.g., developer, tester, manager, user, etc.) by such diagrams.



Kivat diagram depicting relative magnitude (attained or ranked) of each software product quality.

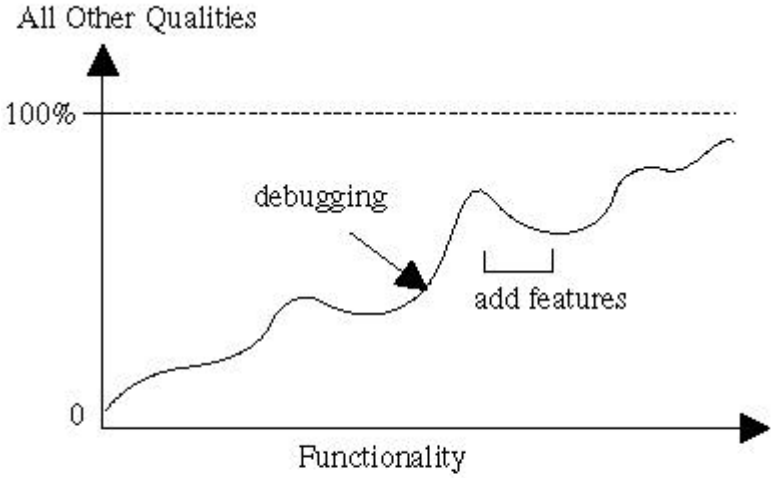


Another viewpoint of the relative importance of each software product quality.

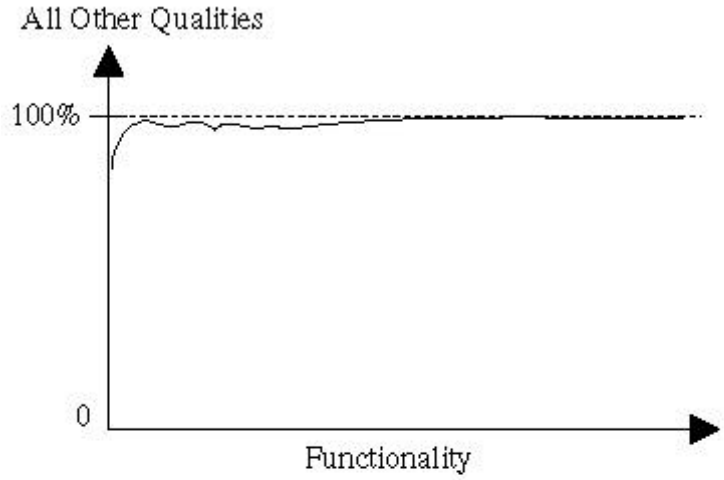
Note also that, in practice, the units for each dimension are not absolute but normalized usually by size (lines of code (KLOC) or function points (FP)). For example, ideal correctness is 0 errors, ideal cost is $0, etc. In practice these are normalized to defects/KLOC and $/KLOC, respectively.

2.5 Quality First

Meyer proposes the Quality First approach as an alternative to the norm highlighted by Osmond [5]. The aim is to try to 'pin' certain qualities at their optimum while incrementally increasing others over time. In other words, not all qualities are considered equally important (as discussed above), but the order of optimizing some factors can be critical.



Typical development addresses functionality and other qualities simultaneously. This 'balanced' approach allows certain defects, such as correctness and robustness problems to become a serious drain during all down–stream activities. For example, testing and documentation are much more costly when correctness and robustness are not adequately achieved earlier.



The *Quality First* approach doesn't add functionality until the other qualities are perfect. This results in less time spent debugging and ensures that the product is of known high–quality at all times..

Consider the reasonableness or unreasonableness of addressing certain qualities last. For example, suppose cost considerations are deferred until late in the project. For some projects, this would not matter much, for others it would result in termination (over-running a fixed-cost budget).

Consider functionality. Functionality is generally increased monotonically from inception. This is the basis for 'evolutionary prototyping' or 'creeping featurism' - depending on your disposition. Are there any factors that there is a consensus for 'maintaining optimal' at all times?

Consider understandability. Software is easy to understand by its authors, but without good in-line documentation (comments) can become a serious problem for others, or even the original author

years later. Productive programmers know that source code is 'write-once-read-many' and that it takes far less effort to document code as it is written than after the fact.

Consider correctness and robustness. Can these be retrofitted late in a project? Certainly not. Low correctness and robustness (high number of errors per KLOC) has a draining effect throughout each subsequent development phase.

Therefore, while there may be disagreement on the relative ranking of the various software quality factors, some only make sense if they 'come first'. These would include: understandability, correctness and robustness, for without these, it is very hard to make progress on the others.

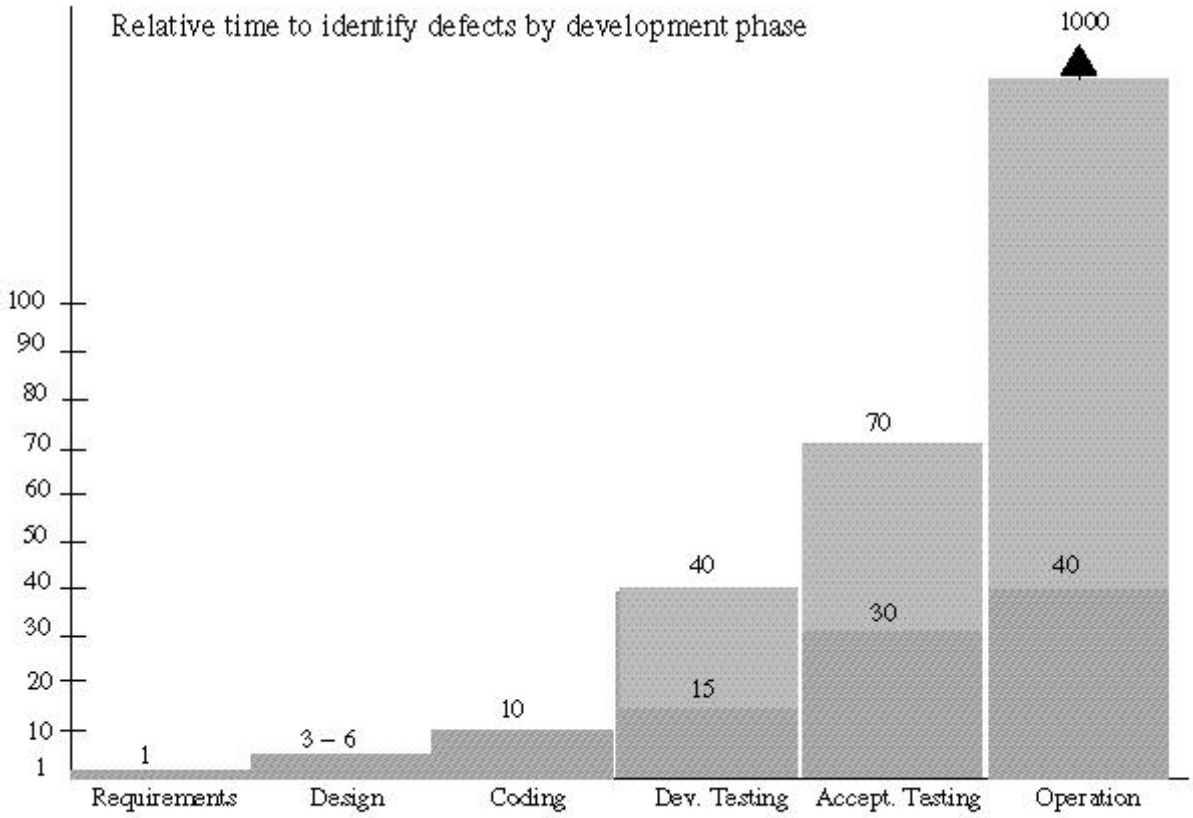
3. Defects

There are various definitions in the software engineering literature for 'software woes' such as 'bug', 'error', 'fault', 'failure', etc. [1]. Some differentiate based on the software development phase (compile, test, usage) in which the problem arises. For our purposes, we will simply use an all-encompassing definition:

> *Defect: A property of the software that can and should be improved.*

Why is software defective? Because people are imperfect. How does it become defective? Through errors of commission (normal defect injection) and errors of omission (missed opportunities at improvement). Errors of commission include the obvious things such as incorrect program logic, mismatched comments, etc. Errors of omission can be missed opportunities for better designs and implementations (e.g., a design pattern could have been used to yield a better solution) [16]. This can be counted as a defect injection.

Certainly programmers do not intentionally write defective software (except in certain pathological cases - e.g., driven by mis-applied metrics...). The well-known exponential increase in the cost to remove defects, must be actively addressed.



Boehm, Barry, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice–Hall, 1981.

Clearly our software development process should seek to minimize the number of defects in the product as early as possible. To achieve this various processes are applied as 'quality filters'. (Some are described later.)

This leads to the Defect Equation:

Di = Di-1 + Ii - Ri for i = 1,2,3,...

The number of defects remaining in the product after the ith development phase (Di) equals the number of defects remaining after the previous phase (Di-1) plus the number of defects injected during the ith phase (Ii) minus the number of defects removed during the ith phase (Ri).

The goal is to minimize Di in the last iteration. That is, minimize the number of defects in the final version of the software that is released to users. Of course, we cannot know what that number actually is. Some of the software development literature on the subject seems to miss these two points, instead focusing obsessively on Ri, the only known quantity (aka 'yield'), sometimes to the point of committing ends-means reasoning errors [17]. At most, Ri is a lower bound on (Di-1 + Ii). (Why?)

Instead we must estimate the unknowns. Certainly we may begin with initial condition D1 = 0 (undeveloped software contains no defects). We can (should) keep an accounting of the number of defects removed during each phase. An example would be failures during test. Humphrey advocates (somewhat controversially) tracking compile-time errors and even syntax errors caught

during pre-compilation 'desk-checking' [17]. This is one way of tracking part of the defect injection rate. But there will always be defects that escape our detection and the detection of all subsequent phases - even final usage.

We can never know these for sure. But there is an interesting proposal for estimating further by considering the defect removal rate of, for example, code reviews. Simply start with a presumed "defect-free" program and then intentionally inject a known number of defects and then count how many of these defects are uncovered in subsequent trial code reviews []. This results in an effectiveness measure of Ri which is then used to estimate Di.

Suppose such trial reviews were found to be 70% effective - finding 70% of the known defects. If 5 defects are found in 100 lines of 'real' code reviewed then we have:

X * 0.7 = 5

X = 5 / 0.7 ~= 7

X - 5 = 2

So there are likely two remaining unknown defects in the 100 lines of reviewed code. Of course, the five known defects must be removed (except possibly in Cleanroom).

Such corrections to the code provide opportunities for injecting new defects. It is reported that some 30-50% of defects found are attributed to attempts to fix other known defects. This 'bad-bug-fix' syndrome is a major component of Ii. If we have kept track of such information (as suggested by Humphrey) we can estimate Di. And so on to the next iteration...

A problem with such models is that they can yield high-precision answers despite underlying ignorance. Simplistic experiments that form the basis for some of these estimates do not adequately control the many variables: people, problem domain, tools, languages, methodologies, etc. As a reality-check one might compare published empirical results and recommendations of software experiments to those of, say, prescription drug clinical trials. The latter are held to much higher standards of statistical rigor.

In their simplicity, they also don't account for the difference in importance between various kinds of defects. Minor defects such as inappropriate use/non-use of whitespace in formatting are usually counted the same as major ones such as race conditions.

But equally important and not adequately addressed in the software development literature, is the "cumulative drain" effect of subsequent attempts to remedy such defects. First, not all defect removal processes are equally effective or have equal cost. And second, the order that such processes are applied does not affect the net defect count (by commutivity of addition) but it does affect total cost (the knapsack problem). Therefore we should consider the overall cost-effectiveness of various orderings of defect removal activities.

Next we will consider some typical defect detection/removal processes - their effectiveness and cost.

4. Techniques That Address Correctness Defects

4.1 Static Analysis Tools

Modern software development environments include a suite of software tools that provide some form of static analysis that can find defects in source code. An obvious example is a compiler. Despite Humphrey's definitions [17], a compiler (that is not deficient like most C++ compilers) will, by definition, only accept legal programs and this implies detection of all syntax errors in the source code. The compiler is thus a (mandatory) quality filter - a compilable program is a necessary (but not sufficient) condition for quality.

Of course, if typographical errors happen to yield valid (but unintended) syntax, the compiler will usually not detect these defects (unless it causes other problems - such as unresolved names during linking).

After compilation, most programmers are anxious to run the program. But there are other quality filters that can be applied first. For example, commercial tools such as lint, FlexeLint, CodeCheck, CodeWizard, etc. offer excellent static defect detection capabilities for some programs (namely, C and C++ source). These tools identify likely problems and violations of coding standards.

Then there are metrics tools. These compute various software metrics such as size (e.g., lines of code), magnitude (Halstead's length, volume, etc.), complexity (e.g., McCabe's Cyclomatic Complexity), modularity (e.g., fan-out coupling), stability and abstractness (e.g., Martin's Main Sequence Distance).

Note that some of these are design metrics. It is interesting that implementations can be "reverse-engineered" to infer design attributes that can then be evaluated against suggested standards. For example, "the cyclomatic complexity of a routine should usually be less than 10 and if it exceeds 15 the routine should almost certainly be rewritten". Or "a category's Main Sequence Distance should be near zero" - implying an ideal balance between "abstractness" and "instability" [18].

Static analysis tools provide a very low-cost (amortized) highly effective (and reliable) means to detect certain kinds of defects.

4.2 Testing

Testing is a time-honored approach for finding certain kinds of defects - namely correctness, robustness, efficiency, functionality, ease of use and, indirectly, testability defects. In a bottom-up approach there are different levels of testing - from the lowest level (unit testing) through intermediate levels (integration testing) and higher levels (acceptance testing / aka validation).

Unit testing aims to verify the above qualities of individual modules (e.g., routines and/or classes). So called, "white box / glass box" techniques are used to assure that all statements are executed at least once. Boundary Value Analysis is used to test programs in the vicinity of the bounds of the inputs - just below, at, and above the expected limits - to uncover very common kinds of defects related to handling input near the boundary of expected values.

Integration testing involves testing assemblies of modules. Here one focuses on the integration code ("glue code") that connects already verified modules. Eventually, the entire system is assembled and tested using "black-box" techniques which focus on the functional behavior: inputs and outputs, without regard to internal logic. Finally, validation testing occurs as users begin to use the system in (sample or real) scenarios.

Formal variants of black-box testing include the independent statistics-based testing that is part of the Cleanroom approach to software development. However, Cleanroom is controversial, in part, because it downplays (or eliminates entirely) unit testing and thus does not directly address code coverage [14].

Testing activities can typically consume 30 to 40 percent of the total development time. To increase productivity, CASE tools can be applied to partially automate significant portions of testing activities including: analysis of testability, test case generation and execution, code coverage analysis, performance profiling, documentation, etc. Commercial tools for these purposes (e.g., Insure++, Purify, PureCoverage, CASEVision, Jtest, TestCenter, TestWorks, etc.) are a very low-cost and highly effective means for implementing parts of the "testing quality filter".

4.3 Formal Technical Reviews

Despite the merits of testing tools, there are some defects that won't be found by testing. In fact, reports indicate that good old-fashioned design and code walkthroughs/reviews/inspections (variations of formality) detect more defects than testing [17, 19]. From the informal peer reviews to formal Fagan inspections to more modern approaches, this is certainly an effective means of finding defects [20, 21].

Perhaps an obvious reason for this, which is never mentioned in the literature, is simply that people will work more carefully when they know they will be presenting their work in front of their colleagues. The influence of ego/guilt psychology is a reality that can be harnessed. Like independent testing, and debugging, the other major reason for the success of inspections may be the "two-eyes are better than one" factor - the fact that those most familiar (the authors) are in the worst position (psychologically) to look for defects in their creations.

Much of the bureaucracy of these processes can be reduced by modern versions such as asynchronous distributed/concurrent reviews. Equally important, the goal of such reviews can and should be expanded to further improved developer (individual) and organizational knowledge and effectiveness [21].

For example, an overlooked opportunity for improving the ability of developers to produce good designs and implementations is to review existing examples of excellent designs and implementations. Identifying and discussing what makes certain examples excellent (and using them as role models) is a "non-negative" form of enlightenment that is the best long-term solution to improving an individual's ability to consistently create high-quality software.

In any case, inspections are very human-intensive, time-consuming activities. As such, they require a high yield (defects found) to offset their high cost. With the above additions of asynchronous concurrency, education, training and reuse of excellence, this quality filter can be both effective and economic.

4.4 Formal Proofs Of Correctness

At the extreme end of the formality spectrum is the so-called "program proving". In this highly human-intensive activity, mathematically skilled analysts specify and then prove system functionality using specialized languages (e.g., Z, VDM, Larch, etc.) derived from predicate calculus.

Despite claims of its advocates, learning this method and developing the requisite mathematical skills to apply it effectively is non-trivial. In the beginning at least, one often makes at least as many mistakes in the mathematical specification and proving activities as they would in implementing the toy-sized programs they are attempting to derive specifications for and prove. This reality is further magnified by the fact that the method does not scale up to "real world"-sized systems. The largest programs that have been proved correct are only a few thousand lines of code.

Another fundamental flaw is that the method does not address the correctness of that which the program depends on: libraries, operating system, hardware, etc. If any of the program's infrastructure is seriously flawed then the 'proved correct program' is just as susceptible as any other to failing to meet its requirements. (Attempts to prove such infrastructure such as compilers, operating systems and microprocessors have not been successful.)

Nonetheless, correctness proofs have their place: they are used to verify parts of "critical" software such as operating system schedulers, network protocols, and military and medical embedded software systems. It is applied to yield software with at least *fewer* correctness defects than might otherwise be delivered. This method certainly has value but requires significant advances in CASE tools to automate it so it can be applied more cost-effectively.

4.5 Limitations of These Techniques

Each of these "quality filters" are an important component of a quality-oriented software development approach. But they each have limitations in applicability, effectiveness and cost. To summarize their strengths and weaknesses:

Static analyzers

+ Very effective (for certain kinds of defects)

+ Very low cost (amortized tool cost)

+ Very reliable (automated so 100% of certain kinds of defects are found)

- Narrow focus (can't find many kinds of defects - but improving...)

Inspections

+ Very effective (high yield)

+ Wide focus (consider all possible kinds of defects)

- High cost (human time intensive)

- Variable reliability (human variational effectiveness)

Testing

+ Very effective (for certain kinds of defects)

+/- Cost (low-cost tools, moderate overall time)

Formal Proofs of Correctness

+ Effective (for some parts of a system)

- High learning curve (for both staff and customers)

- Very high cost (human intensive)

- Not scalable

- Not full-life-cycle (gap between specification and implementation)

4.6 Ordering For Cost

Limitations of individual techniques can be partially overcome by combining techniques. One study reports the relative effectiveness of combining 16 permutations of four techniques: (1) formal design inspection, (2) formal code inspection, (3) formal quality assurance, (4) formal testing. By combining all four methods one can achieve 99% defect removal efficiency [20].

However, what is neglected is the relative cost associated with applying each combination of techniques. This is the "knapsack problem". In other words:

1. The costs of applying each technique depends on which techniques were already applied before it.
2. There is an optimal ordering of techniques that yields the same end results (number of defects removed) but with minimal cost (schedule time).

Consider the following examples:

Suppose one uses a CASE tool that features a language-aware editor (e.g., like *emacs* or *MacPascal*) that prevents or detects and corrects defects such as formating, syntax errors and naming, style/coding standards violations etc. *as they are typed*. It might even feature default code templates to guide the program author. And concurrent static analysis that warns when cyclomatic complexity of a routine exceeds 10, for example. And an incremental compiler that compiles code segments (e.g., routines) as they are completed. Such front-line quality filters would eliminate time spent in subsequent phases detecting and correcting these specific kinds of defects and would certainly reduce overall development time. There would be no waiting for compiles and wading through long lists of error messages to correct "trivial errors".

Having applied such a CASE tool would not eliminate the need for subsequent code reviews but it would certainly reduce their focus by eliminating the need to verify that class of defects known to have been filtered already. So the code review process would be made more cost-effective.

Conversely, had the code review been done before compilation - as advocated by Humphrey [17] - the code review would need to include 'passes' to check for this class of defects in addition to the checks for classes of defects not filterable by the compiler/CASE tool. While subsequent compilation would not require correction of these defects, the time spent finding them manually is orders of magnitude more than the time saved by not detecting and correcting them in the compilation phase. Therefore it is more cost-effective to apply static analysis tools such as smart editors and incremental compilers *before* code reviews.

As a second example consider what sometimes happens when a user reports a defect with a program and, after much analysis, it is determined (by the developers) that the program is "right" because the requirements are vague and do not specify what should be done in the particular instance discovered by the user. (A real example is a C compiler that launched a game when a C program executed a statement that leads to "undefined behavior" according to the ANSI C specification!) Nonetheless, if a change is demanded by the customer it will often be very costly compared to having detected and corrected the requirements in an initial phase before design and coding. Think of the cascading effect of changing requirements can have on all downstream processes and products.

Clearly, like smart CASE tools, requirements reviews are an important low-effort high-yield quality filter that should come earlier in the chain of processes.

These are obvious end-cases. The next example is from a mid-process activity - code review and illustrates a serious flaw in current recommended 'best practices'. (This is from an on-line publication about the merits of a particular CASE tool for code reviews [22]:)

```
Name: dectonum

Specification:
This function scans the string str, beginning at the byte position
given by first, for the character representation of a decimal value.
This value, converted to numeric, is returned as the value of the
function, and last is set to indicate the next character in str after
the value that was found. If any scanning or conversion errors are
found, converror is set to true. The maximum length value to be
scanned is 4 decimal digits.

 Source-code:
 int dectonum(char *str,int first, int *last, BOOLEAN *converror)

 {
   int n, i;
   BOOLEAN scanning;

   n = 0;
   i = 0;
   scanning = true;
```

```
  *converror = false;
  while (scanning) {
    if (str[first + i] >= '0' && str[first +i] <= '9')
      n = n * 10 + str[first +i] - '0';
    else if (str[first +i} == ' ')
      scanning = false;
    else {
      *converror = true;
      scanning = false;
    }
    i++;
    if (i > first + 3]
       scanning = false;
  }
  *last = first = first + i;
  return n;
}

Subject: Incorrect expression
Category: Coding Incorrect
Criticality: Hi (Fatal Error)
Source-code: dectonum
Lines: 21
Description:
This expression will fail when first != 0. It should have
been (i > 3).
Consensus: Confirm:2* Disconfirm:0 Neutral:0
Related-issues:
Proposed-actions:
Comments:
```

This is supposed to demonstrate the usefulness of a particular CASE tool for helping to review code. But instead, what the example illustrates is the inefficient process used to develop this software. While it is true that the code contains an error in the implementation - a correctness defect, what is more important are the other higher-level defects:

1. The signature of the routine:
    - Input argument 'str' should be declared 'const' since it is read-only.
    - Argument first is not really necessary (str + first) would suffice in the call.
    - Names and formatting are poor.
    - Argument converorr has non-standard type, name, and is unnecessary.
2. The specification is too vague:
    - Does not specify whether or not signed integers are valid. E.g., -2, +3. (the implementation indicates that they are not.)
    - Does not specify what whitespace characters are valid. I.e., blanks, tabs, newlines and how many of each.
    - Contains an unnecessary arbitrary limit of only 4 digits to be accepted.
3. The entire routine is not needed (error of mission) since there already exists a routine, strtol(), in the Standard C Library that provides this functionality.

As a result of not identifying these design defects earlier, considerable time is wasted in subsequent process phases: code review, testing, documentation, etc. Consider what could happen after the

above error is fixed: a tester may create test cases that use signed integers and tabs for whitespace. This would result in rejection by the routine. Yet nothing in the (rather verbose) written description of the routine specifies that these are invalid inputs. So this would spawn more problem reports and finger-pointing argumentation between the author of the routine and the tester. Another waste of precious schedule time that could have been avoided by proper ordering of phases, for example: design review (of interface) followed by coding.

4.7 We Need Additional Low-Effort High-Yield Techniques

Yet the design review may not have resolved the vagueness of the interface specification. Even software designed for widespread reuse suffers from vagueness of interface specification. Imprecise interface specification is a major cause of problems in software use/reuse. And even in development processes that incorporate design reviews, code reviews and rigorous testing and documentation, much time is wasted finger-pointing before clarifying the specifications.

But it doesn't have to be this way. Besides ordering 'quality filter processes' for minimizing cost, we also need additional low-effort high-yield techniques to eliminate problems resulting from imprecise specifications. This is the missing link in current recommended 'best practices'.

5. A Missing Link: The Contract

5.1 Back To Basics

The two crucial questions about software correctness are:

1. *What exactly is the software supposed to do?*
2. *Is it is doing exactly what it is supposed to do?*

Correctness is Relative to a Specification

To answer the first question, we need a precise specification to define what it means for a particular piece of software (e.g., routine) to be correct. If the specification is vague then we cannot answer the question.

To answer the second question, we need some way of ensuring that the software - the actual *executing program*, not some *paperware 'proof'* - is doing precisely what is called for in the specification (no more, no less).

Proofs of correctness are an attempt to answer question 1 but are seriously flawed because of: (1) the gap between the paper proof and the actual implementation (code) and its runtime behavior and (2) the difficulty in mastering and applying the method to large-scale development.

The low-effort high-yield solution is Meyer's *Design By Contract* which was introduced a decade ago in his definitive work, *Object-Oriented Software Construction*, and is a cornerstone of his *Eiffel* language [2].

Design by Contract (DBC) is a discipline for defining precise checkable interface specifications for software components based on the theory of Abstract Data Types (ADTs) and the business

contracting metaphor. As will be shown, DBC greatly increases the ability of a programmer to write correct software *and* know that it is correct.

ADT theory is a mathematical approach for the specification of functions and properties of entities defined for modeling. It is akin to the formal proof techniques and will not be covered here. For an example of how it is applied to software development, see [2].

The second idea is easy to understand and generally applicable:

5.2 Business Contracting/Sub-contracting Metaphor

In the business world, contracts are precise (legally unambiguous) specifications that define the obligations and benefits of the (usually two) parties involved. For example, consider an advanced reservation non-refundable discount vacation package deal.

	Obligations	Benefits
Client	Pay entire cost one month in advance. Bring only limited baggage and valid passports and visas, etc. Arrive at airport two hours before flight leaves.	Enjoy a low-cost hassle-free all expenses paid vacation.
Supplier	Locate, select and book airline carrier, auto rental, hotel, etc. within required time frame and budget.	Make profit on sale even if client does not show up at the airport on time and properly equipped. May even keep possible partial refund from hotel etc. in such cases.

Subcontract

Considering the above example, the supplier (travel agent) will sub-contract to various other suppliers (e.g., airlines, hotels, etc.) but only if they can do so within the constraints already promised to the client (time frame, fraction of cost).

How does this apply to software correctness?

Consider the execution of a routine. The called routine provides a service - it is a supplier. The caller is the client that is requesting the service. We can impose a contract that spells out precisely the obligations and benefits of both the caller (client) and the callee (supplier). This contract serves as the interface specification for the routine. For example, consider a routine that merges two sorted sequences.

	Obligations	Benefits
Client	Ensure that the both of the sequences to be merged are each already sorted.	Quickly obtain a sorted combined sequence.

Supplier	Efficiently merge the two sorted sequences into one.	Need not check for and handle unsorted sequences.

Note that the obligations of one party correspond to the benefits of the other party. This simplifies matters in that we need only focus on the pair of obligations.

How can this contract be expressed in software?

Assertions

Assertions are boolean-valued expressions that specify some condition that must be satisfied (evaluate to true). The Eiffel language will be used to illustrate the concept since it provides the most extensive support for the notion. Here is Eiffel code for this example:

```
merge (a, b: SEQUENCE [G]): SEQUENCE [G] is
        -- Merge already sorted a and b (in one pass).
    require
        a_already_sorted: sorted (a)
        b_already_sorted: sorted (b)
    deferred
    ensure
        result_is_sorted: sorted (Result)
        result_is_right_size: Result.count = (a.count + b.count)
        -- result_has_all_of_a:
        --     For all i in [1, a.count]
        --         Result.has (a.item (i))
        -- result_has_all_of_b:
        --     For all i in [1, b.count]
        --         Result.has (b.item (i))
        -- result_has_nothing_more:
        --     For all i in 1..Result.count
        --         a.has (Result.item (i)) or
        --         b.has (Result.item (i))
    end
```

In this example, the *require* clause lists assertions that spell out the obligations of the client (caller of the routine). In this case, that both input sequences must already be sorted. Each assertion has a name followed by a colon followed by a boolean expression. The *ensure* clause lists the assertions that spell out the obligations of the supplier (the routine itself) and has the same form. Note that some of the obligations of the supplier are specified as comments (--). (More on this later.)

The assertions in the require clause are pre-conditions of the routine. The assertions in the ensure clause are post-conditions of the routine. Together, they specify the routine's contract with the client: *"if you promise that the two input sequences are sorted, then I promise that the result will be sorted and contain all items from the input sequences and nothing more."*

Implicit in this contract is that the pre-conditions are sufficient and complete [2]:

No Hidden Clause Principle: The pre-condition is the only requirement that a client must satisfy to get served.

Contract-based assertions answer question (1) "*What exactly is the software supposed to do?"* by providing a means to state the precise interface specification of software, in terms that are relevant to the software elements. This is far better than stating requirements in some abstract mathematical terms which leave one wondering if the actual software implementation is somehow faithful ('isomorphic?') to the mathematical specification.

DBC also addresses question (2) *"Is it doing exactly what it is supposed to do?"* - by monitoring.

Assertions Can Be Monitored at Runtime

Runtime monitoring of assertions means that code is generated to evaluate assertions (boolean expressions) to make sure they contract is satisfied (all assertions evaluate to true). This closes the gap between specification and implementation of the contract.

What if an assertion evaluates to false?

If an assertion evaluates to false then this indicates a violation of the contract. Such contract violations indicate an error (correctness defect) in the software. Specifically, a violation of the client's obligation (require clause) indicates an error in the client (caller) and a violation of the supplier's obligation (ensure clause) indicates an error in the supplier (routine implementation). These errors must be removed before the software can be considered correct.

In practical terms, if an assertion evaluates to false an exception is raised transferring control out of the routine and, (unless intercepted - e.g., by a debugger tool) will abort the program leaving a call stack trace and core file to help the programmer when fixing the defective program.

5.3 What DBC is Not

DBC Is Not Defensive Programming

Why not just make the routine 'more robust' by checking for and handling these cases? This is the often advocated 'defensive programming' approach which is at odds with another related principle [2]:

Non-Redundancy Principle: Under no circumstances shall the body of a routine ever test for the routine's pre-condition.

There are several justifications for this:

1. Sometimes there is no reasonable way to 'compensate for' violations.
2. Even when there is a way, the resulting code is more complicated and less efficient - sometimes greatly so (consider above merge example).
3. Defensive programming does not scale up. Consider a sequence of nested calls. If each routine is written defensively, it is likely that the system will contain many redundant checks - in the body of implementation code - and the total complexity and performance will suffer greatly.

Doesn't all this checking occur anyway with assertions and make the program run too slow?

Best of Both Worlds: Demonstrable Correctness vs Maximal Efficiency

In Eiffel (and other languages that support assertions) one can enable or disable runtime checking of assertions on a class and kind basis. For example, for a particular set of class, you may decide to enable checking of ensure clauses but not require clauses. This can be useful if you believe the class implementation to be correct (e.g., extensively reviewed, tested, etc.) but still want to help catch errors in client code that uses the class.

This flexibility provides the best of both worlds: demonstrable correctness (during verification and validation) and maximal runtime performance (during production use) with the ability to choose the degree on a per-class basis.

Not an Input Checking Mechanism

Assertions are for documenting interface specifications and optionally verifying them. These are correctness properties not robustness properties. This means assertions are not usable for 'handling' resource allocation failures. That is, failures due to factors beyond the program's control, such as user input, file I/O, network disconnects, etc. do not constitute defects in the program and so cannot be 'asserted away'. Instead, conscientious failure detection and mitigation/correction code must still be written to handle these cases.

5.4 Aspects of Assertions in Eiffel

This section explains in more detail important aspects of assertions in Eiffel.

Comments vs Full Implementation

The above example illustrates the limitations of assertions for expressing requirements. Currently, Eiffel does not support the full predicate calculus - universal quantifiers, etc. (In fact, predicate calculus may not be sufficient for such needs [2].) In such cases, one must either express them as comments (which cannot be checked at runtime) or implement them with auxiliary functions. For example:

```
...
ensure
    result_is_sorted: sorted (Result)
    result_is_right_size: Result.count = (a.count + b.count)
    result_has_all_of_a: Result.has_each (a)
    result_has_all_of_b: Result.has_each (b)
    result_has_nothing_more: Result.has_only (a, b)
end
```

The *has_each* and *has_only* functions would implement the required checking.

Abstract Assertions

Eiffel allows assertions to be specified for deferred routines and even in terms of deferred routines. These are sometimes called abstract assertions. For example, feature *remove* in deferred class *MY_DISPENSER* has an assertion that is implemented in terms of deferred feature *count*.

```
count: INTEGER is
        -- Current number of items stored.
    deferred
    ensure
        non_negative: Result >= 0
    end

remove is
        -- Remove accessible item.
    require
        not_empty: not empty
    deferred
    ensure
        one_less: count = old count - 1
    end
```

Inherited

Eiffel assertions are inherited so that redefined features in descendant classes must also satisfy them. This is also done in a logically consistent manner - i.e., pre-conditions (require clauses) can only be weakened and post-conditions (ensure clauses) can only be strengthened. This is implemented by effectively *oring* the require assertions and *anding* the ensure assertions as indicated by the modified forms **require else** and **ensure then**. For example, in class *MY_DEQUE*, *put* is inherited from *MY_DISPENSER* and renamed *put_last* but still inherits the assertions from *put* (the parent contract):

```
put_last (an_item: G) is
        -- Make an_item the last one.
    do
        ...
    ensure then
        item_is_last: last_item = an_item
    end
```

Note that the inherited assertions do not appear in the code of the redefined routine, but are implicit. However, since they are still a part of the routine specification, they will appear in the generated documentation views (called *short form* in Eiffel) such as:

```
put_last (an_item: G) is
        -- Make an_item the last one.
    require
        not_full: not full
    ensure
        one_more: count = old count + 1
        item_is_last: last_item = an_item
    end
```

Class-Level Assertions

In addition to applying assertions to features, Eiffel also supports class-level assertions, known as *class invariants* (from ADT theory.) Class invariants are a set of assertions that must be satisfied at all times (after creation) except during execution of a class routine. They can be thought of as implicitly *anded* with all routine pre-conditions (except creation) and all routine post-conditions.

For example, *MY_DISPENSER* contains the following **invariant** clause:

```
invariant

   not_void: a /= Void
```

implicitly *anded* to the above.

Eiffel's support for routine and class-level assertions are the means for implementing Design By Contract and supporting the notion of ADTs. But there is even more to Eiffel's assertions:

Loop Assertions

Beyond software interface specifications, assertions play a crucial role in assuring the correctness of software implementations. One way is with loop invariants and loop variants (motivated by previous work on proving program correctness [7,8,9,10,11]).

- *Loop Invariant*: An assertion which must be satisfied before and after each iteration of a loop.
- *Loop Variant*: An integer expression which must be non-negative prior to the first execution of a loop, and decreased by every iteration, so that it will guarantee loop termination.

Here is an example:

```
gcd (a, b: INTEGER): INTEGER is
        -- Greatest common divisor of a and b.
    require
        a_strictly_positive: a > 0
        b_strictly_positive: b > 0
    local
        x, y, remainder: INTEGER
    do
        from
            x := a
            y := b
            remainder := x \\ y
        invariant
            remainder_big_enough: remainder >= 0
            remainder_small_enough: remainder < y
            x_strictly_positive: x > 0
            y_strictly_positive: y > 0
            -- GCD (x, y) = GCD (a, b).
```

```
        variant
            remainder
        until
            remainder = 0
        loop
            x := y
            y := remainder
            remainder := x \\ y
        end
        Result := y
    ensure
        result_strictly_positive: Result > 0
        result_small_enough: Result <= a and Result <= b
        result_divides_both: a \\ Result = 0 and b \\ Result = 0
        result_is_greatest:
            -- For all c in [Result, a.max (b)] c \\ a /= 0 or
            --                                 c \\ b /= 0
    end
```

Check Assertion

Finally Eiffel provides a **check** clause for assertions suitable for verifying intermediate stages of a computation.

```
check
    rewound: empty implies before
end
```

Note also Eiffel's implication operator: *a* **implies** *b (=* **not** *a* **or** *b)*

5.5 How DBC Relates To Other Techniques

Now that we have seen what Design By Contract is and how it works, let's consider where it lies in the spectrum of quality filter techniques.

Formality Spectrum

As the discipline of software development matures it is natural that it become more formalized and standardized. It will likely retain aspects of art (creativity, ingenuity, imagination, intuition) and engineering (practical knowledge backed by mathematics and principles of scientific investigation). Current practice includes a range of techniques on the formality spectrum, yet all move toward the more formal end.

Consider design pattern languages - a formalism for capturing knowledge about good designs and communicating it to others [16]. Human-computer interface design is now based on important principles learned from studies of ergonomics, perception and related fields.

Yet it is clear there is resistance to increasing formalism - often justified by the lack of sufficient understanding of principles and/or lack of faith in results from empirical studies compared to similar but more mature fields such as architecture [23].

Design By Contract is inspired by formal approaches embodied in specification languages such as Z, VDM, etc. Yet is less mathematical and more software-oriented, constrained, in fact, to expression in terms of software elements (variables, routines, classes, etc.). It is as formal as possible without loosing touch with the software.

DBC is Practical

Much of the potential power of formal specification proofs lies in the hope for automating the proofs. If there were tools to do the mathematical analysis required to determine if specifications were complete, consistent and correct (yield the desired outcome) then much progress could be made in software design. But so far, these tools have not materialized. This makes proofs of correctness impractical for large-scale development.

DBC, on the other hand, is practical. Its goal is not proof of correctness, but rather runtime verification of 'no contract violations'. A less ambitious but very useful and easily attainable goal. It is a compromise between the ideal and what is practical (today).

DBC is Seamless

Mathematical specifications introduce a gap between the specification and the implementation. At most, it leaves one wondering if the actual software implementation is faithful to the specification. As long as the specifications are non-executable or otherwise not a part of the actual software, there will always be this gap.

DBC defines specifications in terms of actual software elements (e.g., routine signatures, class invariants, etc.) so there is no gap. DBC provides a seamless transition from design (interface specification) through implementation (filling in the code for the routine body).

DBC is Low-Effort

The software-oriented rather than mathematically-oriented nature of DBC makes it a far easier technique to master and explain to other non-mathematical staff (e.g., customers during joint-application development). DBC uses the language of the problem domain (assertions about domain objects) and implementation domain (software infrastructure elements). As such it is readily accessible to customers and programmers alike.

Errors made in defining assertions are easily caught during execution and trivial to fix. Compared to errors committed during formal proofs of correctness and proofing the proofs, DBC is a very low-effort technique.

DBC is Low-Trade-off

DBC does not require trade-offs against other qualities. For example, defensive programming has been touted as a technique for achieving robust software. Yet it requires a large trade-off against efficiency and understandability due to the many levels of redundant checking for the same 'error conditions'.

Because DBC employs assertions which can be selectively disabled, there is no permanent efficiency degradation. A code with assertions disabled performs no checking of contract violations and so runs as fast as possible. And since assertions are free of side-effects and are not used for robustness checking, there is no trade-off against robustness. A correct program produces the exact same result (in much less time) with assertions checking disabled as with them enabled.

DBC is zero-trade-off.

DBC Is High-Yield

DBC is a high-yield technique - i.e., it is a very powerful quality filter. Its precision and automatic monitoring of assertions traps many defects that would otherwise propagate to later phases where identification and correction are much more costly.

It can only be fully appreciated by those that have transitioned into using DBC during their own development efforts and observed first-hand the higher quality of the resulting products and the lower effort needed to produce them. Nonetheless, it is helpful to consider the impact of DBC on other development processes and phases.

5.6 Impact of DBC on Other Activities

DBC During Analysis

Customer-friendly

During the analysis phase, developers must first understand the problem domain and identify domain objects and their relations with others and their constraints. This entails discussions with the customers to elicit requirements of function, behavior and constraints.

DBC is customer-friendly. Since the contract is defined in terms of these domain objects the constraints can be made clear and explicit while not resorting to mathematical language that would tend to alienate customers and other developers. Everyone can understand the business contracting metaphor. This allows business rules (constraints) to become an integral part of the software from the very beginning.

For example, consider this feature from an *ACCOUNT* class:

```
withdraw (amount_to_withdraw: MONEY) is
        -- Withdraw money.
    require
        positive_amount: amount_to_withdraw > 0
        sufficient_balance: balance >= amount_to_withdraw
    deferred
    ensure
        correct_balance: balance = old balance - amount_to_withdraw
    end
```

One need not be a programmer, mathematician or lawyer to comprehend the above specification.

DBC During Design

Integrity via Inherited Assertions

Design includes the specification of classes and their features (e.g., public interfaces). Often these classes are abstract (deferred). That is, they are base classes that define the interfaces that are to be implemented by numerous specialized derived (concrete/effective) classes.

With DBC such class designs include precise interface specifications (the contract) and this contract is automatically inherited by all derived classes. This effectively ensures that the integrity of the design is maintained during the implementation phase.

For example, consider the invariant for an *ACCOUNT* class:

```
deferred class

    ACCOUNT

...

invariant

    non_negative_balance: balance >= 0

end -- deferred class ACCOUNT

deferred class

    ROTH_ROLLOVER_IRA_ACCOUNT

inherit

    ACCOUNT

...
end -- deferred class ROTH_ROLLOVER_IRA_ACCOUNT
```

In this design, *ROTH_ROLLOVER_IRA_ACCOUNT* cannot have a negative balance.

DBC During Implementation

Implementation of effective (concrete) classes are constrained by their ancestor contracts. But new features (routines) may be added with their own contracts. Requirements must be achievable by clients of the class so they must be expressed only in terms of 'public' features of the class - not 'private' attributes. For example:

```
remove is
        -- Remove an item.
    require
        not_empty: not empty
    do
        ...
```

```
ensure
    not_full: not full
    one_less: count = old count - 1
end
```

Where features *count*, *empty* and *full* must be 'public' features accessible to clients of the class. Without this constraint, it would not be possible for clients to understand, let alone satisfy, the contract.

DBC During Testing

DBC Removes Ambiguity

The existence of a precise interface specification - expressed in terms of the software elements themselves - provides the perfect target for testing. The contract defines exactly what the software is supposed to do and assertions monitoring provides the means to verify that it is doing what it is supposed to do. This does not eliminate the need for testing, but it does change the approach to testing.

Assertions-based Boundary-Value Testing

The contract allows testers to focus on the boundary of the contract. That is, with assertions enabled, boundary-value testing is made very easy since violations of the pre-conditions and post-conditions will be trapped - resulting in an exception (usually) which can be caught by the test driver program and logged to an output file for documenting the test cases.

This eliminates the 'finger-pointing' that occurs when software is under-specified and testers report defects that the authors consider acceptable because of the vagueness of the requirements...

Invariant as Built-In Self-Test

DBC also provides the analog to the hardware built-in self-test. The class invariant defines the constraints on the state of the software before and after all feature accesses (public routine calls). As long as this evaluates to true the integrity is assured.

DBC During Usage

Efficiency

Once the software, or portions of it are, deemed correct the assertions monitoring can be disabled. This completely eliminates the runtime penalty associated with evaluating assertions. The assertions are effectively erased from the code. This provides the maximal efficiency and improved opportunities for optimizations such as routine inlining, loop unrolling etc.

For example, consider array accesses inside of loops:

```
...
loop
   a.put (x, i)
```

```
    ...
end
```

With assertions disabled, this becomes raw (unchecked) array indexing. In languages that lack this control over assertions (e.g., Pascal, Java) there is no way to eliminate the array bounds checking overhead.

DBC During Reuse

Documentation

DBC provides the basis for documenting the software interfaces. Potential reusers of a software component need to understand what the software provides them and what they must do to obtain these benefits. This is the contract.

The short-flat forms obtainable in Eiffel provide the ideal specification - as simple as possible, but no simpler. Like the Unix man page synopsis which details the routine signature, but providing the crucial missing part: the pre-conditions and post-conditions of the routine.

Design By Contract is the much needed missing link in the software development practice. Unfortunately it is *a best non-practice*.

5.7 If Design By Contract Is So Great Why Isn't It Widely Applied?

Ignorance

The vast majority of those developing software - even that intended to be reused - are simply ignorant of the concept. As a result they produce application programmer interfaces (APIs) that are under-specified thus passing the burden to the application programmer to discover by trial and error, the 'acceptable boundaries' of the software interface (undocumented contract's terms). But such ad-hoc operational definitions of software interface discovered through reverse-engineering are subject to change upon the next release and so offers no stable way to ensure software correctness.

The fact that many people involved in writing software lack pertinent education (e.g., CS/CE degrees) and training (professional courses, read software engineering journals, attend conferences etc.) is *not* a reason they don't know about DBC since the concept is not covered adequately in such mediums anyway. That is, *ignorance of DBC extends not just throughout practitioners but also throughout educators and many industry-experts*.

Documentation should be enough

There is a misconception that DBC is all about documentation and good documentation should be sufficient anyway. DBC is certainly more than documentation. And documentation is certainly not enough, for several reasons:

1. It is usually too verbose and yet does not spell out the interface specification in enough detail to answer all the client's usage questions.
2. It does not help clients determine if they are using the API correctly since no runtime

checking occurs.

Consider the ancient and familiar Standard C routine 'strcat'. The Unix man page for it describes the gist of it but does not say if it is valid to concatenate a string onto itself as in the following program:

```
#include <stdio.h>  /* For printf(). */
#include <string.h> /* For strcat(). */

int main( void )
{
  char s[ 11 ] = "hello";
  strcat( s, s ); /* What happens in this case? */
  printf( "%s\n", s );
  return 0;
}
```

At the very least, it should say that such usage results in undefined behavior - the typical burdensome standard C cop-out. But it doesn't say that it is invalid so it is reasonable to expect it to work (yield "hellohello") - based on the *No Hidden Clause Principle*. However, testing this program on several different platforms yielded several different results including:

- crashes due to segmentation faults
- crashes due to uncaught signal
- crashes due to arithmetic errors (!?)
- hangs in an infinite loop
- success (yielded "hellohello")

If this routine were designed and implemented according to DBC, it would have to include a pre-condition that the arguments be different:

```
char* strcat( char* d, const char* s )
{
  PRE3( d, s, d != s )
  const old_d_length = strlen( d );
  ...
  POST( strlen( d ) == old_d_length + strlen( s ) )
}
```

otherwise with a weaker pre-condition, it is now obligated under the *No Hidden Clause Principle* to produce the expected result:

```
char* strcat( char* d, const char* s )
{
    PRE2( d, s )
    const old_d_length = strlen( d );
    const old_s_length = strlen( s );
    ...
    POST( strlen( d ) == old_d_length + old_s_length )
}
```

and finally, an instrumented version of the library, libcdebug, would be provided to allow clients to detect errors in their code that misuses the API.

Too Formal

When introduced to the concepts, some programmers are often skeptical of its benefits yet too intimidated by the formality to try it themselves. DBC and Eiffel's supporting assertions are actually very simple to master - a very reasonable compromise compared to formal specification languages such as VDM and Z [13]. Repeating an earlier example:

```
withdraw (amount_to_withdraw: MONEY) is
        -- Withdraw money.
    require
        positive_amount: amount_to_withdraw > 0
        sufficient_balance: balance >= amount_to_withdraw
    deferred
    ensure
        correct_balance: balance = old balance - amount_to_withdraw
    end
```

One need not be a mathematician or CPA or even a programmer to comprehend the above specification.

Correctness is not always a goal

In the commercial sector, the trends seem to prefer shipping buggy code before your competition does and patching it in a later release since this is somehow a more profitable strategy. That this is profitable is partially the fault of users that are willing to buy software of dubious quality. (Compare a software license with a hardware warranty.)

Nonetheless, DBC is such a low-effort technique, why not apply it anyway to boost productivity by saving time in down-stream activities such as testing?

There are other ways to achieve correctness

Testing

Testing is not a substitute for DBC. After all, without a precise specification, what is there to test against? Much time will be wasted in finger-pointing when the software behaves in a manner deemed incorrect by a tester but not its author.

(On the other hand, DBC and assertions are not a substitute for testing either. Testing is still needed not only to verify correctness (by checking assertions at runtime) but also to verify robustness in cases of external failures that are outside the scope of correctness and to ensure 100% code coverage.)

Inspections/Reviews

The same applies to design and code reviews. Without a precise specification, in terms of the software elements themselves, what is there to verify the design against? Who can say whether or not a routine is correct if there is no set of agreed-upon, documented, unambiguous, executable

assertions to compare to?

Formal proofs

The same applies to proofs of correctness. Ultimately what matters is what the software does - not what some mathematical paperware says. The gap between the paperware proof and the running code is a downfall of formal proofs.

Yet Eiffel's assertions were inspired by work in formal correctness proofs. However, the technology does not yet exist for applying correctness proofs to real world sized programs in a timely manner. Eiffel's assertions are the best practical compromise in this direction.

But I do use assertions already

Some programmers do use assertions already. For example, C has long provided an *assert()* macro that can be used for runtime checking of boolean expressions. However, this is ad-hoc application of assertions akin to Eiffel's **check** clause. It is a far cry from Design By Contract - lacking crucial support for inheritance, for example.

Nonetheless, it is possible to support (to a degree) DBC by creating specialized macros for differentiating the assert() macro usage into pre-conditions and post-conditions and diligently writing class invariants (e.g., in C++) that call their parents, etc. [28].

For example:

```
#include <Assertions.h> // For PRE*(), POST*(), CHECK*(), DEBUG().

...

void String::chunkSize( size_t newChunkSize )
{
  PRE( 0 < newChunkSize ) REMEMBER( size_t, theLength )

  theChunkSize = newChunkSize;
  DEBUG( bool ok = ) resize( theLength, true ); // May shrink no fail.
  DEBUG( CHECK( ok ) )

  POST( theLength == OLD( theLength ) )
}
```

This certainly improves the ability to write demonstrably correct FORTRAN/C/C++ programs but still does not go far enough. Assertions alone do not constitute Design By Contract. Assertions are a means for supporting the concept. Assertions are to Design By Contract as dynamic binding is to object-oriented programming - an enabling mechanism - not a substitute for design.

6. Back To The Future

Birth of the Concept

According to Hoare [7]:

"An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

The notion of assertions was further developed by Hoare [7,8,9], Floyd [10] and Dijkstra [11]. Unfortunately, commercial programming languages rarely provide support for assertions, let alone the extensions (e.g., inheritance) needed for supporting Design By Contract. Some attempts to remedy this are given below.

Algol W

Wirth and Hoare's Algol variant allowed executable assertions [2]. It was not widely used and Pascal, Algol's direct successor dropped support for assertions. This is a puzzling omission given that Pascal was intended to be used to teach programming.

ANNA

ANNotated Ada was an attempt to retrofit assertions onto an existing fairly widely used (at least in defense) programming language [24]. It was based on Ada83, not the OO Ada95. As such, its assertions were limited to data transformation rather than a more abstract level required for DBC. Nonetheless, it has been used to help bridge the gap between formal specifications written in Z and an actual Ada implementation.

It is a shame that ANNA was not incorporated into Ada95. It might have had a better chance of becoming more popular.

Eiffel

Meyer's Eiffel language provides the most comprehensive support for assertions in an object-oriented language. As discussed above, assertions form the foundation of Meyer's Design By Contract method.

Further developments may be forthcoming in increasing the power of Eiffel's assertions through an Intermediate Functional Language (IFL). The goal is to move closer to the ideal of specifying and generating provably correct programs, but without sacrificing ease of use and general applicability. See [2] for more details on IFL.

Sather

Sather is a direct descendant (some would say "cousin") of Eiffel. [26]. It is a research project at the UCB International Computer Science Institute (ICSI) on object-oriented language design for high-performance parallel computing. Here is an example of assertions 'pre' and 'post' used in Sather:

```
class CALCULATOR is
  readonly attr sum: INT; -- Always kept positive.

  add_positive (x: INT): INT pre x > 0 post result >= initial(x)
  is
      return sum + x;
  end
```

Despite the influence of Eiffel, assertions were an after-thought in Sather. They were not in the initial version and even the current version (1.2) lacks support for assertions in abstract classes. Even the documentation describes assertions as "safety features for the earnest programmer to annotate the intention of code". This hardly recognizes their role in the important Design By Contract notion.

A++

Motivated by both ANNA and Eiffel, Cline and Lea sought to provide similar capabilities for C++ programs through Annotated C++ [25]. Here is an example of an annotated C++ class:

```
class Vector
{
private:
  int cap;   // Capacity of the Vector.
  T* data;   // Pointer to the actual data elements.
public:
  int size() { return cap; }
  T& operator[]( int i ) { return data[ i ]; }
  void resize( int s );
  Vector( int sz = 10 ) : data( new T[ sz ] ), cap( sszize ) {}
  ~Vector { delete [] data; }

// Here are the annotations in A++ syntax:

legal: cap >= 0;
coherent: data.nelems == cap; // (Not sure about data.nelems...)
axioms:
  [ promise return >= 0                                ] size ();
  [ int x;  require x  >= 0 && x < size()              ] (*this)[ x ];
  [ int sz; require sz >= 0; promise size() == init.sz ] resize( sz );
  [ int sz; require sz >= 0; promise size() == init.sz ] Vector( sz );
};
```

A++ even attempts to go further by providing a degree of static analysis that would eliminate redundant checking in cases where it (the A++ pre-processor) has proved that the checks are implied by previous checks.

After STL, A++ was perhaps the most promising advance for C++ programming. It is a shame that it never materialized beyond a research project.

iContract

Perhaps the latest development in Design By Contract is the iContract tool for Java [27]. Since the Java language provides even less support for DBC than C (there are no optional assertions), the

approach used here is to apply a pre-processor that recognizes stylized comments indicating pre-conditions, post-conditions and invariants, and generates instrumented Java source code that checks the various kinds of assertions.

Here is an example [27]:

```
// File Person.java

interface Person
{
  /**
   * @pre age > 0
   */

  void setAge( int age );

  /**
   * @post return > 0
   */

  int getAge();
}


// File Employee.java

class Employee implements Person
{
  private int age_;

  public void setAge( int age )
  {
    age_ = age;
  }

  public int getAge()
  {
    return age_;
  }
}
```

The above source code is processed using iContract:

```
$ iContract.Tool -mpre,post,inv Person.java
$ iContract.Tool -mpre,post,inv Employee.java > Employee_I.java
```

to yield instrumented code shown by this routine:

```
public int getAge()
{
    //#*#-------------------------------------------------------------
    int __return_value_holder_;
    //-------------------------------------------------------------#*#

    //#*#-------------------------------------------------------------
```

```
        /* return age_; */
        __return_value_hlder_ = age_;
        //----------------------------------------------------------#*#

        //#*#----------------------------------------------------------
        if ( ! ( __return_value_holder_ > 0 ) )
            throw new RuntimeException( "Employee.java:5: error: "+
                                        "postcondition violated "+
                                        "(Person.getAge()): "+
                                        "(/*return*/ age_) > 0" );
        //----------------------------------------------------------#*#

        //#*#----------------------------------------------------------
        return __return_value_holder_;
        //----------------------------------------------------------#*#
    }
```

The instrumented code is not intended for human consumption, but only to be fed to the Java compiler to yield code with a runtime-monitored contract. The tool even aims to implement inherited assertions and some predicate calculus operators such as 'forall' and 'exists'. iContract, when fully implemented, will be an important advance in Java development.

7. Conclusions

Quality Requires Design By Contract

Design By Contract backed by optional runtime assertions is the most important *best non-practice* in software development today. It is a low-effort, no-trade-off, high-yield quality filter that has a positive impact on all software development activities from analysis through reuse. It is not a silver bullet but it is a crucial missing link - a necessary but not sufficient condition for achieving quality software.

Without DBC, increases in software reuse - "components" such as JavaBeans - can worsen the already poor quality of most software. Without a clearly defined and enforced contract, potential reusers are at risk. They cannot know exactly what the software is supposed to do. They cannot know if it is doing what they think it should. And they cannot easily know if they are using it correctly. And all of what they discover through trial-and-error is subject to change on the next release because there is no documented enforceable contract.

Reusable Components - Contract = Disaster

The dangers of reuse without a contract have been illustrated by the 500 million-dollar failure of the Ariane 5 rocket which has been argued to result from reuse without a properly specified and checkable interface [3].

While such news-worthy incidents highlight the vulnerable nature of software-controlled systems the real tragedy is the more subtle on-going industry-wide drain of human resources wasted due to poorly specified and/or uncheckable interfaces.

Culture Shift: Too Soon or Too Late?

The simplicity and obvious benefits of Design By Contract lead one to wonder why it has not become 'standard practice' in the software development industry. When the concept has been explained to various technical people (all non-programmers), they invariably agree that it is a sensible approach and some even express dismay that software components are not developed this way.

It is just another indicator of the immaturity of the software development industry. The failure to produce high-quality products is also blatantly obvious from the non-warranty license agreement of commercial software. Yet consumers continue to buy software they suspect and even expect to be of poor quality. Both quality and lack-of-quality have a price tag, but the difference is in who pays and when. As long as companies can continue to experience rising profits while selling poor-quality products, what incentive is there to change? Perhaps the fall-out of the "Year 2000" problem will focus enough external pressure on the industry to jolt it towards improved software development methods. There is talk of certifying programmers like other professionals. If and when that occurs, the benefits of Design By Contract just might begin to be appreciated.

But it is doubtful. Considering the typical 20 year rule for adopting superior technology, DBC as exemplified by Eiffel, has another decade to go. But if Java succeeds in becoming a widely-used language and JavaBeans become a widespread form of reuse then it would already be too late for DBC to have an impact. iContract will be a hardly-noticed event much like ANNA for Ada and A++ for C++. This is because *the philosophy/mindset/culture is established by the initial publication of the language and its standard library*.

For Pascal it was Jensen & Wirth's "manual", for C it was "the K & R white book". And C++ is a case study in the problems caused by releasing an immature language that lacked adequate compilers and a standard library. For Java it is too late for DBC to make inroads via iContract because it is not a language-level mechanism and, equally importantly, it is not applied in the Java Standard Libraries.

At this point, only Eiffel (and its derivatives) include DBC as a cornerstone of their culture. This is great for Eiffelists, but DBC is too valuable to be ignored by those, who for one reason or another, do not use Eiffel. As shown above, DBC can be applied (with varying ease) in a variety of languages.

Try it!

The benefits of this easy small step towards formalism can only be fully understood and appreciated by those that have adopted it and reflected on the positive impact it has had on the various activities of their development work.

The next time you set out to develop a reusable software component, think about the 'what' before the 'how'. Consider the 'what' from both the client and server's viewpoints (separately). Then codify these obligations and benefits so that they can be optionally monitored at runtime. (Provide two versions of the component - a 'debug' version with checking enabled and an 'optimized' version without checking.) Finally, document this contract as part of the component's interface.

The next time you set out to reuse a software component, first examine its contract. Ask "What exactly does it require (of each input)?" and "What exactly does it promise (of each output/result)?"

These answers are its (documented) contract. Then ask "How can I be sure that I am using it correctly?" and "How can I be sure that it does what it is supposed to?" Satisfactory answers to these questions require that the documented contract be optionally enforceable - that is, checkable at runtime. Ask the developer of the component to provide you with 'debug' and 'optimized' versions of the component (if pre-compiled).

Welcome back to the future of software development!

8. References

1. Meyer, Bertrand, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, Upper Saddle River, New Jersey, 1997, pp. 3-19, 39-64.

2. ___, pp. 331-410.

3. ___, "Design by Contract: The Lessons of Ariane," *IEEE Computer*, vol. 30, no. 1, January 1997, pp. 129-130.

4. ___, "The Next Software Breakthrough," *IEEE Computer*, vol. 30, no. 7, July 1997, pp. 113-114.

5. ___, "Practice To Perfect: The Quality First Model," *IEEE Computer*, vol. 30, no. 5, May 1997, pages 102-106.

6. Davis, Alan, *201 Principles of Software Development*, McGraw-Hill, New York, 1995.

7. Hoare, C.A.R., "The Emperor's Old Clothes" (1980 Turing Award lecture)," *Communications of the ACM*, vol. 24, no. 2, February 1981, pp. 75-83.

8. ___, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576-583.

9. ___, "Proof of Correctness of Data Representations," *Acta Informatica*, vol. 1, 1972, pp. 271-281.

10. Floyd, Robert F, "Assigning Meanings to Programs," *Proc. American Mathematical Society Symp. in Applied Mathematics*, vol. 19, 1967, pp. 19-31.

11. Dijkstra, Edsger W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

12. Jones, Cliff B., *Software Development: A Rigorous Approach*, Prentice Hall International, Hemple Hempstead, U.K., 1980.

13. Wing, Jeannette M., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, September 1990, pp. 8-24.

14. Beizer, Boris, "Cleanroom Process Model: A Critical Examination," *IEEE Software*, vol. 14, no. 2, March/April 1997, pp. 14-16.

15. Dromey, R.G., "A Model For Software Product Quality," Technical Report, Software Quality Institute, Griffith University, October, 1994. http://www-sqi.cit.gu.edu.au/publications/Model_prod_qual.ps

16. Gamma, Erich, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addision-Wesley, Reading MA, 1995.

17. Humphrey, Watts, *A Discipline for Software Engineering*, Addision-Wesley, Reading, MA, 1995.

18. Martin, Robert, "OO Design Quality Metrics," September, 1995. http://www.oma.com/PDF/oodmetrc.pdf.

19. Jones, Capers, "Software Defect Removal Efficiency," *IEEE Computer*, vol. 29, no. 4, April 1996.

20. Fagan, Michael, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol 15, no. 3. 1976.

21. Johnson, Philip M., "Reengineering Inspection," *Communications of the ACM*, vol. 41, no. 2, February 1998, pp. 49 - 52.

22. ___, FTRtool:http://www.ics.hawaii.edu/~johnson/FTR/.

23. Davis, Al, "Predictions and Farewells," *IEEE Software*, vol. 15 no. 4, July/August 1998. pp. 6-9.

24. Luckham, David C, et al., "ANNA - A Language for Annotating Ada Programs," *Lecture Notes in Computer Science 260*, Springer-Verlag, 1987.

25. Cline, Marshall P. and Doug Lea, "The Behavior of C++ Classes," ftp.clarkson.edu.

26. Omohundro, Stephen and David Stoutamire, "Sather 1.1," 1996. http://www.icsi.berkeley.edu/~sather/Documentation/Specification/Sather-1.1/index.html.

27. Kramer, Reto, "iContract - The Java Design By Contract Tool," http://www.promigos.ch/kramer

28. Plessel, Todd, "Assertions.h - C/C++ assertions macros," http://www.elj.com/eiffel/dbc/Assertions.h

(Ed: Banner provided as an OO Community Service.)

[Eiffel Liberty *(New)* | GUERL | sOOap] [Top]

Page is http://www.elj.com/eiffel/dbc/
Last Modified: 02 Dec 1998 (Created: 02 Dec 1998)
