Reference: Bertrand Meyer, *The Grand Challenge of Trusted Components*, in ICSE 25 (International Conference on Software Engineering, Portland, Oregon, May 2003, IEEE Computer Press.

The Grand Challenge of Trusted Components

Bertrand Meyer
ETH Zürich, Chair of Software Engineering
and Eiffel Software, Santa Barbara

se.inf.ethz.ch

Abstract

Reusable components equipped with strict guarantees of quality can help reestablish software development on a stronger footing, by taking advantage of the scaling effect of reuse to justify the extra effort of ensuring impeccable quality. This discussion examines work intended to help the concept of Trusted Component brings its full potential to the software industry, along two complementary directions: a "low road" leading to qualification of existing components, and a "high road" aimed at the production of components with fully proved correctness properties.

1 Overview

Progress in software engineering over the past four decades has undeniably had a beneficial effect on software quality. The advances fall short, however, of the kind of break-through that appear necessary to cope with the increasing demands on our software. The general idea developed in this article, reuse, is far from new; but the notion of *Trusted Component* brings in a critical concept: extreme attention to component quality. This is indeed the definition of the term:

> **Trusted Component**
> A Trusted Component is a reusable software element possessing specified and guaranteed property qualities.

(The term "trusted" as introduced in [14] appears destined to stay, but any confusion should be avoided with the notion of trust in secure computing. Although security properties will be part of the quality a component must provide, "trust" in Trusted Components extends to all other quality factors.)

2 Approaches to quality

The considerable existing literature on improving software quality advocates a variety of approaches, which one may classify along two orthogonal criteria: Management vs Technical; and A Priori vs A Posteriori.

The first division affects which of *process* and *product* is viewed as more important:

- Some authors consider that most serious software engineering problems are, in the end, *management* problems, and that technology is a secondary concern. Naturally they focus on process-oriented solutions: Capability Maturity Model, ISO certification, Six-Sigma, relationship of IT with the rest of the company, user awareness. As an example of this approach, the Standish Group's CHAOS reports [19], well-known for their large-scale analysis of software project failures, offer a recipe for project success consisting of ten "success factors" that are *all* non-technical: more user involvement, executive support, clear business objectives, experienced project manager, small milestones, firm basic requirements, competent staff, proper planning, ownership (complemented by "other"). This is an extreme variant of the management-first approach, not accounting for the report's own finding of a significant improvement in project success rates between 1994 and 1998. (It is not immediately evident why "user involvement" or "executive support" should have made major advances during that period; on the other hand, 1994 to 1998 was also the time when, at least in a partial form, object technology was spreading through the industry, a technology factor discounted by the list. The factors given are not even software-specific.) In less exclusive forms, books on software engineering economics [4] and "Peopleware" [6] emphasize non-technology factors. Frequently heard advice about the need for better education also falls in this category, even when the education is about technical topics. Yet another example is the systematic use of code inspections, whether for open source ("enough eyeballs") or in commercial settings.

- By contrast, a large portion of the literature is devoted to technical solutions, from programming language features enhancing reliability (static typing, modular mechanisms...) to implementation support (garbage collection), testing techniques, formal development methods, design principles.

The second division affects whether the techniques are applied to producing the software, or to assessing and correcting it once it exists:

- *A priori* techniques help avoid flaws in the first place; along ideas already cited, executive support and use of formal methods fall (from remote ends of the spectrum) into this category.
- *A posteriori* techniques apply when the software or at least some initial version already exists; they help identify and correct any deficiencies that may have escaped the a priori component of the quality effort.

Neither division is absolute: a technique such as configuration management spans the technology / management line; Design by Contract [10] [12] spans the a priori / a posteriori line. Most techniques, however, fit in one box of the following table showing some examples of the classification.

	Technical	***Management***
A priori	Design methods Object-oriented development Formal development	User involvement Executive support Better education for engineers and managers.
A posteriori	White-box testing Static analysis (e.g. PREfix) Proofs (of existing programs)	Testing and acceptance procedures

Figure 1: Varieties of quality advice — examples

Most of the ideas listed have a part to play in the search for software quality. As argued elsewhere [15], the field should avoid compartmentalization and the ever-present tendency to push various approaches as exclusive. There is no magical solution, and every little bit helps.

3 The limits of project-oriented techniques

Useful as they are, the solutions usually proposed tend to target individual projects. This seems unlikely to yield the breakthrough in quality and productivity that the field needs. So much new software gets produced every year — billions of lines — that a more efficient approach seems desirable. As to education, hordes of new developers enter the market every year; only a few of them will have had the benefit of a good software engineering education. It is unlikely for example that the reported 6 million Visual Basic developers are all trained computer scientists.

Education is of course necessary, but it's a never ending effort and we can hardly make the overall quality of software development dependent on its progress.

We can address the issue through Trusted Components. The availability of a large base of guaranteed-quality reusable elements would make application development less dedicated to producing software from the ground up, and hence less dependent on the individual skills of project developers; the focus would shift towards composition, combination, mix-and-match. This particularly applies to the huge amount of software being developed by non-computer-scientists. On such a large population, in whose concerns and jobs software is often only a part, our most earnest attempts at education can only have a limited effect. Our strongest lever is to give them impeccable components of ever broader scope, with a division of labor between component providers and component consumers.

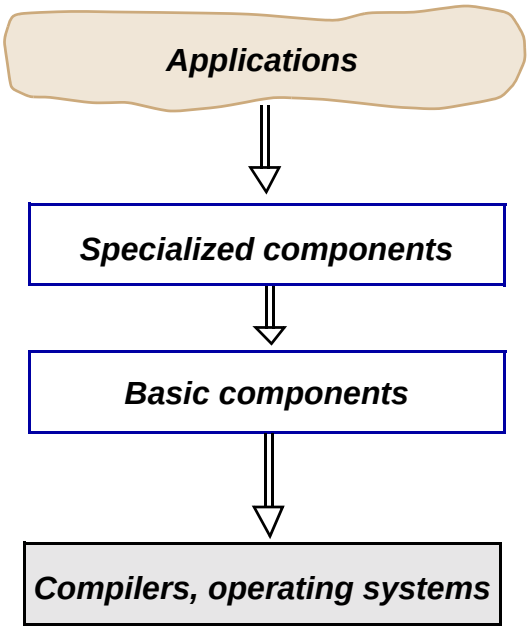

Figure 2: Levels of software and expertise

The application developers can still mess up when combining and extending components, but the more and better we give them the less this will happen. The division of labor suggested by figure 2 helps: an application expert may be best equipped to fight quality problems that depend on the specifics of the application, its users, its corporate environment; but to make sure that the infrastructure is right — to address typical software technology issues such as choice of data structures and algorithms, communication, synchronization, database optimization — it's more effective to rely on components put together by software professionals in these respective areas.

4 The case against quality

Dependence on programmer education is not the only obstacle to the potential effectiveness of the most widely cited approaches to improving software quality. An economic argument suggests that, applied to projects rather than components, they face a natural limit.

Considering the matter from an economic perspective is legitimate because quality, at least as perceived by project developers, their managers and the rest of an organization, has a cost. The theoretical argument that "Quality is free" because non-quality causes greater cost is of little interest to the project manager faced with looming deadlines and budget constraints. The practical question then becomes *how much* quality effort the project can afford. Every manager knows that a non-perfect but acceptable version on time is better than absolute perfection two years later, especially if the company has gone bust in the meantime for failing to release a product. There is a quality incentive, but it only leads to the acceptability point: the stage at which remaining deficiencies do not endanger its usefulness to the market. Beyond that point, most managers consider that further quality-enhancing measures yield a quickly diminishing return on investment.

This pragmatic view, which some have cited as one of the reasons behind Microsoft's success, is no longer the dirty secret of project management since Ed Yourdon gave it respectability under the name "Good Enough Software" [21].

> Although Yourdon has recently stated, citing Microsoft initiatives to address some security-related issues, that Good Enough may not be good enough any more in the post-September-11 climate, the overall industry attitude doesn't seem, at least so far, to have dramatically changed. (The hard economic times can have opposite effects on companies, some tempted to get rid of any non-core efforts such as quality initiatives, others realizing that improving quality is their best bet for survival.)

Good Enough Software is a global consequence of accumulating individual optima. The example of capitalism suggests we should strive, in human systems, for an equilibrium where the combination of individual strategies that each person considers optimal for his own purposes matches the global optimum that best benefits the group as a whole. The prevalence of the Good Enough approach indicates that in the software engineering field we have not reached such an equilibrium: the ideal for society is that all software should be as good as possible; but if I am a project manager the ideal for me is only to make my product good enough. In fact, by spending more money and time to make my product better than it strictly has to be for market acceptability, I am putting at risk the product, my company and my own career.

This is the culture that, in many places, has made "perfectionism" a dirty word for software development. The reasons are clear: programmer perfectionism can indeed be detrimental to key goals of management, chief among them timely delivery of an acceptable product. But that culture is unlikely to create the conditions from which to expect a breakthrough in software quality.

The same forces work against correcting instances of *non-quality*: bugs or, more generally, anything that could be done better. If someone points out a possible improvement to an application, and I am responsible for that application, then improving it is *my* job. Not surprisingly, my reaction may be to avoid fixing anything that's non-critical. That may indeed be my best personal survival strategy in the Darwinian world of industrial software development; but globally it's not good for the industry.

This clash between the optimization of individual strategies and the pursuit of the common good may be a stronger obstacle to software quality improvement than any technical limitation of current quality approaches such as the ones cited in section 2. Many known ideas and solutions — from better programming languages and advanced tools to formal methods — have the potential of ensuring better processes and products, and yet are not as widely used as they could; the various justifications for refusals to use them are often smokescreens for the real reason: that quality is only one the criteria, and improving it doesn't necessarily justify the cost.

If this hypothesis on how companies and projects react is correct, it puts a natural limit on the practical effect of any quality technique that entails any costs, as long as the technique is directed to individual *projects*. Major progress can only come from changing the economic incentives so that individual perfectionism is good not only for everyone else but for the individual as well. Basing software development on Trusted Components appears to do this:

- The component developers (producers) make quality components available to application developers (consumers), who can turn them to their advantage in building their applications.
- It is the common interest of the consumer and the producer that the components should be good, and that deficiencies should be fixed. Because the managerial and political framework is different, consumers' choices have a higher likelihood of being based on technical merit, since their goal is the success of their application, based on the components.
- If in the course of developing an application a consumer spots a deficiency in a component, the burden of correcting it falls on someone else. It is then in the consumer's interest to exert pressure to have the deficiency corrected; this is also in the producer's interest in a competitive situation.

As a result there appears to be, in this scenario, a better match between optimization of the common good and the combined optimization of individual goals.

5 Components and quality

The concept of Trusted Component is of course based on the idea of software reuse, long present in the software engineering literature. But arguing for Trusted Components is not just arguing for reuse.

Although reuse has often been advocated — at the same level as techniques such as those listed in section 2 — as one of the ways to improve quality, a reuse-based development process is not by itself a quality guarantee. The expected advantages come from the scaling-up effect, which lets new applications benefit, again and again, from a one-time investment; but reuse scales up everything, the bad as well as the good. The improper reuse of earlier software in the Ariane 5 maiden launch disaster has been widely cited as an illustration of the dangers [9].

Trusted Components are the combination of *reuse* with a special attention to the *quality* of the components being reused.

The quality standard for Trusted Components is far higher than what is commonly exercised on application software. To be more precise, the only area in which similar criteria apply (or should apply) is the construction of life-critical systems as in transportation and defense. There the analysis of costs and benefits justifies the extra effort. With Trusted Components, the justification comes not from the net impact of each software element by itself, but from its cumulated impact through the many applications that will reuse it.

This multiplicative effect of reuse also adds some personal and technical factors to the economic and political elements discussed so far. We should not underestimate the benefits of accepting and even encouraging developers' perfectionism. The common managerial perception that perfectionism is bad — because it delays the project and hence harms the company — fosters a general attitude of cynicism, detrimental to the morale of the best developers. With reusable components, the picture changes:

- It pays to improve the details; perfectionism is justified by the expected reuse effect. "Worse is better" [7] no longer has to hold.

- Conversely, *not* improving the details becomes dangerous. There is no "minor" flaw any more, since with the potential of widespread reuse for a component *any* deficiency has an increased likelihood that *some* client application will exercise it.

So as managers we can, without a bad conscience, let component developers exert their best creativity and pay attention to details.

Component development is one of the most exciting opportunities for a talented software developer. It's the opportunity to take the time to do things right, and let many consumers benefit.

Component development is also the place to try out the newest, most advanced techniques, before they find their way into ordinary application development. The development of reusable components is the Formula-1 of software engineering.

The term "component" has not been defined. As the reader may have guessed, this discussion doesn't restrict itself to the case of binary, directly deployable components as advocated by Szyperski [20]; rather, we accept a broad range of components that can extend from classes of object-oriented libraries (or even subroutines of numerical libraries) to very large-grain components, provided they meet specific criteria:

Definition: component

A component is a software element (modular unit) satisfying the following three conditions:

1• It can be used by other software elements, its "clients".

2• It possesses an official usage description, which is sufficient for a client author to use it.

3• It is not tied to any fixed set of clients.

Condition 1 distinguishes a component, meant to be used by other software, from a program, meant to be used by people or by non-software systems. A given software element may be usable both as a program and as a component, for example an application (say Excel) that is available to human users but can also be packaged into a COM object for use by software clients. Condition 2 brings in information hiding and the need for interface specification. Condition 3 excludes a software element that would useful only as part of a certain program or set of programs, and hence could take advantage of insider knowledge about the context of its use; a component should instead be part of a library and usable by any client that respects the interface conditions.

[16], part of discussions with Szyperski [3], has argued that what matters most is not the binary nature of components but information hiding and abstraction. Without again entering this debate we note that most of the present article applies equally to Szyperski's more specific view of components. For example, the lessons gained from the systematic development of the EiffelBase object-oriented library of fundamental data structures and algorithms, and the resulting principles as described in [11], seem largely to extend to components of other kinds; and more generally the principles of Design by Contract [10] [12] appear particularly relevant for components [2] [17].

6 High and low road

If Trusted Components are indeed one of the major hopes for a breakthrough in software quality, how do we go about establishing a base of high-quality components?

An examination of the state of the art in industry and research shows a great discrepancy between what's possible in both worlds:

- In industry, components have taken off in recent years, with great practical success, from object-oriented libraries (C++, Java, Eiffel) to ActiveX controls, COM objects, Enterprise Java Beans, .NET assemblies. These efforts could greatly benefit from a stronger quality focus and in particular from *component certification* applied to existing commercial components.

- In research, one of the great advances of the past decade has been the practical improvements to formal methods, and in particular to proof techniques, making it possible to produce sizable, realistic systems equipped with full proofs. Although it has not yet changed the generally negative image that "formal methods" convey to many people, this practical progress is undeniable. Applications so far have largely been focused on life-critical systems; the reasoning developed above suggests that components are another ideal application area, as the two notions of reuse and formal development seem destined for each other [13].

The gap, however, is huge. Choosing one of the directions to the exclusion of the other would mean ignoring either long-term prospects for major advances or the short-term needs of industry. This suggests defining two complementary paths for the progress of Trusted Components, a low and a high road:

- The *low road* starts from today's components, commercial or open-source. The extent of the quality guarantees we can achieve there is by nature limited; in particular, *proofs* are excluded in almost all cases, if only because the source code is generally not available in the commercial case, but more fundamentally because one of the lessons of formal methods is that one can't hope to prove software unless it has been produced with that goal in mind. Even if not accompanied with absolute guarantees, *certification* is still possible in this case; as we'll see next, we may hope to provide interesting if incomplete quality assessments.

- The *high road* is intended to lead to components with fully proved properties. The ambition of this goal implies that it's more long-term, and that its realization must start with relatively fine-grain (but practically critical) components such as library classes.

The efforts along both directions are different. An effective Trusted Components effort should be both realistic, following the low road for immediate benefits to industry interested in the quality of today's components, and ambitious, following the high road for the ultimate goal of fully proved components. The Trusted Components effort that we have started at ETH, with the aim of establishing a Component Certification Center, is intended to address both sets of issues.

7 Towards a Component Quality Model

One of the most important tasks on the "low road" is to define criteria against which to assess components. In other words we need a Component Quality Model, intended to assess products rather than (as the Capability Maturity Model) processes for obtaining these products.

A first framework for a Component Quality Model appears in figure 3. It divides properties of interest into five categories, the "ABCDE" of component quality (I am indebted for this terminology to Richard Walker of ANU in Canberra):

Acceptance
- A.1 Some reuse attested
- A.2 Producer reputation
- A.3 Published evaluations

Behavior
- B.1 Examples
- B.2 Usage documentation
- B.3 Preconditioned
- B.4 Some postconditions
- B.5 Full postconditions
- B.6 Observable invariants

Constraints
- C.1 Platform spec
- C.2 Ease of use
- C.3 Response time
- C.4 Memory occupation
- C.5 Bandwidth
- C.6 Availability
- C.7 Security

Design
- D.1 Precise dependency documentation
- D.2 Consistent API rules
- D.3 Strict design rules
- D.4 Extensive test cases
- D.5 Some proved properties
- D.6 Proofs of preconditions, post-conditions, invariants

Extension
- E.1 Portable across platforms
- E.2 Mechanisms for addition
- E.3 Mechanisms for redefinition
- E.4 User action pluggability

Figure 3: Framework for a Component Quality Model (the ABCDE of Trusted Components)

The categories are largely orthogonal, as there is no single scale against which to assess components.

A, for Acceptance, is a non-technical dimension: a provider may claim a component is reusable, but it helps to have evidence of usage. This part of the classification addresses the natural question of a potential consumer: "Who else is reusing this?". A.1 to A.3 are a scale of evidence pointing to prior successful use of the component.

B, for behavior, addresses a key property of a good component: that it should be equipped with a precise list and individual specification of the functionalities it offers. Here too the levels make up a scale — a gradation towards better evidence of quality: at the minimum we expect examples of use, and probably (B.2) usage documentation. More precise specifications are desirable; for example it is dubious whether one can be comfortable with using a component unless it is equipped (B.3) with a precise specification of its *preconditions*, the conditions under which it will deliver a proper result. (Most commercial components, regrettably, don't include this in any precise way). The next two levels add more of the Design by Contract mechanisms: postconditions, providing either some or (more difficult) full specification of the outcome; invariants.

C, for constraints, covers performance considerations. Here we no longer have a scale, just a set of partly independent criteria, such as response time, security (protection against undesired use), bandwidth requirements, ease of use. Techniques of performance specification and enforcement are less developed than the techniques of behavior specification (the previous category), but the need is just as important.

D, for Design, a scale again, considers the *internal* perspective. What you want as a component consumer is a guarantee of the component's *external* properties — properties characterizing what it makes available to you — but any information on the principles and techniques that the component authors have used during development can reinforce your trust. At the very least you will want an indication of the component's dependencies on hardware and software elements [20]. The next few levels correspond to systematic design rules; our experience with EiffelBase, as documented in [11], has taught us that, beyond a few components, a library should follow strict and uniform rules for naming classes and their features, choosing arguments (operands and options), distinguishing between commands and queries (that is to say, staying away from side-effect-producing functions), uniform access. Knowledge that the library authors have followed a systematic policy of this kind is an important element to reassure component consumers. The last levels involve *proofs* and open up the "high road" of this discussion.

E, for Extension, addresses an important feature of software reuse: it's not only about reusing components as they are, but also (unlike reuse in the "hard" engineering disciplines, hard in the sense that their products are material and you can't just drive your fist into them) adapting a component to your specific needs. One aspect is portability: applicability to different platforms. For growing flexibility, you may have the possibility of adding your own mechanisms to the components; of redefining some existing components, as with inheritance in object-oriented development (under the guidance of contracts); and, the most flexible, the ability to plug in your own mechanisms, as with callbacks or higher-level versions of this notion.

This classification provides a first grid of criteria against which to assess components. The plan is, clearly, to develop it to the point where it can be used as a component certification standard.

8 The high road: proofs

The high road involves producing components with correctness proofs. It is in general meaningless to talk about "proving software": one can at best prove that the software satisfies specific properties. The phrase becomes legitimate, however, if we consider software elements that are already equipped with *correctness properties*; this is the case with the classes of Eiffel libraries which possess contracts. For this reason we are devoting our first efforts to this target, starting with the EiffelBase library which has the added interest of covering data structures and algorithms that are both often difficult to get right and constantly useful for everyday programming across essentially all application domains.

The contracts consist of preconditions, postconditions and class invariants, complemented (in the routine bodies) by loop variants and invariants, as well as "check" instructions. They have been present in EiffelBase and other libraries right from the start, but until now have been applied (even if the prospect of proofs was always latent) to other purposes [12]: as analysis and design aids, to get the software right; as support for readability; for documentation (the "contract form" is the official interface documentation of a class); as management aids (allowing managers to follow what's going on at a sufficiently high level of abstraction); for testing and debugging, through the run-time monitoring of contracts.

Thanks to theoretical and practical advances, it is now becoming possible to consider the contracts as properties — characterizing software abstractions and their implementations — to be proved mathematically. Beyond a certain level of complexity, manual proofs, tedious and error-prone, are not sufficient; we are working with proof tools such as Atelier B [1] [5] for this purpose. The challenges are numerous:

- Matching the usual top-down process of formal, proof-oriented development, usually applied to full systems, with the bottom-up style of component-based development and the needs of proving individual components.
- Developing appropriate modeling techniques for critical but tricky mechanisms such as pointers.
- Complementing the existing class contracts to arrive at full specification.
- Developing the appropriate proof techniques.
- Making sure these techniques are appropriate for mechanization, and feeding them into the proof engine.

As noted earlier, one can hardly hope to prove software that hasn't been built for this purpose. Even with the contract-rich style that exists throughout our libraries, we expect that the libraries themselves will have to be adapted to support proofs (apart from any correction of bugs uncovered during attempted proofs).

A number of ongoing papers [18] report on the progress we have achieved so far in these various directions.

> We don't limit ourselves to proofs but also consider *tests* of contract-equipped components, using the contracts to guide test generation.

Many other groups are, of course, working on related issues.

The proofs we have studied so far apply to object-oriented library classes, only a part of the component story. We feel, however, that this is the right way to start the proof effort:

- These libraries, as noted, play a key role in everyday development. The ability to rest on a fully proved set of data structures and algorithms classes would be by itself a major boost to trust in software quality.
- The problems involved — such as extensive use of pointers — are difficult. The ability to solve them is an important prerequisite for proofs of other kinds of components, and a good testbench for proposed proof techniques.
- One of the conditions for performing proofs in any application domain is the existence of a *theory* for that domain. For business-oriented components, building such theories will take time. Data structures and algorithms have been researched for several decades and enjoy a wide body of theory, even if only a subset is useful for proofs.

We hope that this high-road approach, with its specific focus, will be an important complement to the more short-term approach of the low road, and that its relevance will continue to grow.

9 A grand challenge

Tony Hoare, in advocating a large-scale effort to produce a "verifying compiler", has listed a number of conditions [8] for a research goal to be a "grand challenge" that should mobilize a significant part of the community, on a key unsolved issue, for a decade or so, with ambitious goals that can in principle be attained, but not without special effort, resources and dedication. ("Send a human to the moon before the end of the decade".)

For software engineering, the major issue of yesterday and today, which pollutes everything we do, is our inability to produce industrial software, on a regular basis, with a quality level of which we can be proud. Approaches focused on single projects, however good individually, will not bring a solution.

Developing a base of guaranteed-quality components, and extensive techniques for qualifying reusable components, is one of the most exciting prospects for the software world today, perhaps the idea with the highest potential for changing things for real. It satisfies all of Hoare's criteria (arising from "*scientific curiosity about the nature and limits of the discipline*", being generally comprehensible, going beyond what's initially possible, calling for cooperation, necessitating collaboration of specialties etc.), currently with the exception of "*enthusiastic support from (almost) the entire research community, even those who do not participate*". I believe it deserves that enthusiastic support, and I hope this survey of issues, goals and approaches will help catalyze a collective effort to solve the Grand Challenge of Trusted Components.

References

[1] Jean-Raymond Abrial: *The B Book*, Cambridge University Press, 2002.

[2] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau: *Technical Concepts of Component-Based Software Engineering*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh (Pennsylvania), Report CMU/SEI-2000-TR-008, available at www.sei.cmu.edu/publications/documents/00.reports/00tr008.html; consulted September 2002.

[3] *Beyond Objects* column of *Software Development* magazine, articles by Bertrand Meyer and Clemens Szyperski, 1998-2000.

[4] Barry W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981.

[5] ClearSy (name of company, no author listed): Web documents on Atelier B, www.atelierb.societe.com, consulted December 2002.

[6] Tom de Marco and Tim Lister: *Peopleware: Productive Projects and Teams*, 2nd edition. Dorset House Publishing, 1999.

[7] Richard Gabriel: *Is Worse really Better?*, in *JOOP (Journal of Object-Oriented Programming*), Fall 1992.

[8] Tony Hoare: *Criteria for a Grand Challenge*, at www.cra.org/Activities/grand.challenges/hoare.pdf, consulted February 2003.

[9] Jean-Marc Jézéquel and Bertrand Meyer: *Put it in the Contract: The Lessons of Ariane*, in IEEE *Computer*, vol. 30, no. 7, pages 129-130, July 1997.

[10] Bertrand Meyer: *Applying "Design by Contract"*, in IEEE *Computer*, 25, 10, October 1992, pages 40. Also in *Object-Oriented Systems and Applications*, ed. David Rine, IEEE Computer Press, 1994.

[11] Bertrand Meyer: *Reusable Software*: *The Base Object-Oriented Libraries*, Prentice Hall, 1994.

[12] Bertrand Meyer: *Object-Oriented Software Construction*, *second edition*, Prentice Hall, 1997.

[13] Bertrand Meyer: *The next Software Breakthrough*, in IEEE *Computer*, vol. 30, no. 7, pages 113-114, July 1997, available at archive.eiffel.com/doc/manuals/technology/bmarticles/sd/contracts.html, consulted February 2003.

[14] Bertrand Meyer, Christine Mingins and Heinz Schmidt: *Providing Trusted Components to the Industry*, in IEEE *Computer*, vol. 31, no. 5, May 1998, pages 104-105.

[15] Bertrand Meyer: *Towards more reliable software: Every little bit counts*, in IEEE Computer, November 1999, pages 131-133, available at www.inf.ethz.ch/~meyer/publications/computer/reliable.pdf, consulted February 2003.

[16] Bertrand Meyer: *The Significance of Components*, in *Software Development*, November 1999, available at www.sdmagazine.com/documents/s=7207/sdm9911k/, consulted February 2003.

[17] Bertrand Meyer: *Contracts for Components*, in *Software Development*, July 2000, variant available at archive.eiffel.com/doc/manuals/technology/bmarticles/sd/contracts.html, consulted February 2003.

[18] Bertrand Meyer: articles on proving classes at www.inf.ethz.ch/~meyer/ongoing/references.

[19] Standish Group: CHAOS reports at www.standishgroup.com/, consulted November 2002.

[20] Clemens Szyperski: *Component Software*: *Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[21] Ed Yourdon: *When Good-Enough Software is best*, IEEE *Software*, Vol. 12, no. 3, May 1995, pages 79-81.