

# The Quest for Software Components Quality

**Miguel Goulão**

Information Systems Group (INESC)  
Departamento de Informática (FCT/UNL)  
2825-114 Monte da Caparica, Portugal  
+351 212948536 (ext. 10731)  
[miguel.goulao@di.fct.unl.pt](mailto:miguel.goulao@di.fct.unl.pt)

**Fernando Brito e Abreu**

Information Systems Group (INESC)  
Departamento de Informática (FCT/UNL)  
2825-114 Monte da Caparica, Portugal  
+351 212948536 (ext. 10707)  
[fba@di.fct.unl.pt](mailto:fba@di.fct.unl.pt)

## ABSTRACT

Component based software development is becoming more generalized, representing a considerable market for the software industry. The perspective of reduced development costs and shorter life cycles acts as a motivation for this expansion. However, several technical issues remain unsolved before software component's industry reaches the maturity exhibited by other component industries. Problems such as the component selection by their integrators, the component catalogs formalization and the uncertain quality of third-party developed components, bring new challenges to the software engineering community. This paper presents an overview of the current research efforts on software components, from a quality assessment perspective. The emerging of empirical component based software engineering practices, to complement the current research trends with a more effective component's quality and complexity evaluation, is perceived as a research area that will become very active in a near future.

## KEYWORDS

Components, Components Selection, Certification

## 1 MOTIVATION

### 1.1 What is a component?

Industries such as the automobile, building, computer hardware or consumer electronics deeply rely upon the availability of application-domain specific components. Those components are often standardized, therefore allowing their availability from multiple sources. In those industries, developing a project is often an exercise of components selection (from catalogs of available ones) and composition (gluing the components together).

A software component is an independently deployable implementation of some functionality, to be reused as-is in a broad spectrum of applications [1].

One may consider an application to be a coarse-grained component executing in the environment provided by the operating system, while a plug-in is a finer-grained component [2]. Certain types of components, which have a very specialized and therefore more predictable usage, such as the ones used in the construction of graphics interfaces (widgets), are commonly used, nowadays.

Components may be sold as commercial products (COTS - Commercial Off The Shelf components), or developed in-

house.

Component Based Development (CBD) involves the technical steps for designing and implementing software components (producers' view), assembling systems from pre-built components (integrator's view) and deploying the assembled systems in their target environments.

### 1.1.1 Integrators view

From an integrator's point of view, a component is a black-box with a very well defined interface. Components are a unit of composition of software architectures. There is a widespread variety of composition of components [2, 3]. For example, with CORBA and JavaBeans the components interface is a list of functions. Other more demanding component models, such as the Catalysis [4] advocate the existence of plug-points or standardized connectors between components [5].

### 1.1.2 Producers view

When designing reusable components, one has to take into account what type of usage the components are expected to support [6].

The components internal technology may vary. It may be, for instance, an encapsulated set of cooperating classes. There are a number of proposals for several technologies to build components for distributed environments, with the corresponding component models, such as Microsoft's *Component Object Model* (COM, DCOM, COM+), *JavaBeans* from *Sun Microsystems* and *CORBA* (*Common Object Request Broker Architecture*) from *OMG* [2]. More recently, the distributed platform ".Net", from Microsoft, provides a framework for the construction of components that may publish or consume products through internet services. In this platform the communication between the components can be performed using *SOAP* (*Simple Object Access Protocol*) and *XML* (*eXtensible Markup Language*).

## 1.2 The Software Components Market

CBD is playing an increasing role in software systems development. Components are expected to reduce the cost and time to market of systems they are part of and increase their quality. In recent years we have witnessed a rising interest in the software engineering aspects of components production and usage. However, quantitative studies are still largely uncovered.

CBD is becoming increasingly important for the software

industry. According to studies performed by four major market researchers (Gartner Group, Giga, Ovum and PriceWaterhouseCoopers) the component market has been growing steadily [1] [7]. The Gartner Group predicted that at least 70% of all new software applications to be developed in 2003 will be assembled primarily from components.

The reuse of software components is perceived has a way of achieving considerable savings, both in the software development costs and in the development life cycles. The shorter production life cycles have, as a consequence, a reduced time to market, mainly due to the increased level of abstraction that can be obtained by packaging various sources of complexity in a component framework, and to the creation of families of components for specific kinds of systems, such as e-commerce applications, thus making product development simpler.

The development of a component industry also allows for the creation of new market opportunities. Components provide a flexible way of shipping functionalities in a conveniently packed black-box fashion that makes distribution simpler. It may be argued that components are easier to adopt than other reusable approaches such as design patterns, although their reuse may require some wrapping on them, or adaptations on the system where they will be integrated in [1]. While we agree that this relatively simpler approach to reuse is a key factor for CBD expansion in a much larger scale than previous reuse approaches, we believe that they are complementary, rather than alternative to reuse approaches such as the one of design patterns.

CBD is supposed to increase the quality of software systems. If the components have been reused in several occasions, they are likely to be more reliable than other software developed from scratch, as they were tested under a larger variety of conditions.

### **1.3 Uncertain quality of third-party developed components**

Software components reliability depends heavily on their usage. Often, only the final users of components can effectively test how the components will react with the system they are being integrated in. If the quality assurance of in-house developed software is a demanding task, doing it with software developed elsewhere, often without having access to its source code and detailed documentation, presents an even greater concern. Introducing software components of unknown quality may have catastrophic results. Regardless of their quality, the usage of components has also an impact on the software process itself. This impact should also be assessed.

The majority of the concerns described in this section have been discussed, in the past in a scientific area known as "Software Engineering". Since the context here is the one

of CBD, it makes sense to talk about Component Based Software Engineering (CBSE).

The aim of this paper is to make an overview of the current state of the art on CBSE and to identify emerging related research issues that are perceived as having a key role in this area with a special focus on quality issues

## **2 THE BRAVE NEW WORLD OF CBSE**

### **2.1 Component Based Software Engineering**

Building up on the IEEE definition of Software Engineering [8], CBSE is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of component based software. It is concerned with the rapid assembly of systems from components where components and component frameworks (sets of interfaces and rules of interaction that govern how components plugged into them may interact) have certified properties and those serve as a basis for predicting the properties of the system [9].

Shifting from a more traditional software developing life cycle to one where software is developed using software components has several implications. A case study on 15 NASA projects that use COTS components, where some key issues on this subject are identified, is presented in [10]. The usage of COTS motivates some process changes due to the creation of new activities, such as the procurement of suitable reusable components that will partially implement the systems being developed. An increased effort not only in requirements analysis, but also in system testing and integration activities is compensated by a smaller effort in activities such as coding, modular testing and code inspections, due to the reuse of components to implement some of the functionalities of the system. Design changes somewhat in its focus, with more emphasis being put to component's integration rather than to internal component details.

### **2.2 Estimation Ability**

New professional competences in the development teams are required to deal with the introduction of these changes in the software process. Activities such as project estimation become more complex. Rigorous estimation efforts are inherently complex to perform. An evidence of that complexity is the large number of software estimation models that have been produced during the last decades to support the software process control activities. The majority of those models were never conveniently validated through comprehensive empirical efforts, partly due to the difficulties in getting adequate samples for that validation. A discussion on the research difficulties in the software measurement area (which is, of course, crucial to estimation) can be found in [11]. The shift to a new approach to software development requires new estimation models to better capture the essence of CBD. Due to the novelty of CBD and related estimation models, there is still a lack of past experience in which one can support his

estimation efforts. An example of a face lifted model that aims supporting cost estimation in CBSE is the COCOTS [12], an evolution from COCOMO II [13].

### 2.3 Flexibility

In order to support the usage of COTS components, the requirements are usually made more flexible. The usage of a given component may induce a change in the initial requirements. If the component adds extra features that are worth taking advantage of, they may be added to the requirements. But the component may also prevent the developers from using some solutions that were initially planned, due to some sort of incompatibility between its behavior and the initial requirements.

### 2.4 Maintainability and Replaceability

The impact of components in software maintenance also has to be accounted for. The usage of components developed elsewhere creates a dependence on the components' suppliers that can emerge if some evolution or adaptation of the component is required. The "easy" answer would be to have a maintenance contract with the component vendor. Some component certification models such as [14] include the possibility of a maintenance contract as one of the items to be evaluated in a component certification process, although we believe a component's certification should not depend on a contract that can be made on an individual basis with each component user. The dependency on an independent components provider may cause some problems. For instance, the component vendor can go out of business. There are possible ways of circumventing these problems:

- alternative suppliers for similar components
- in-house development of components to replace the problematic ones.

Unfortunately, none of these solutions comes free. It is up to the developers to select the most effective one on a case-to-case basis. The former may show not to be feasible since other components may have just an approximate functionality, not a completely similar one. An adaptation effort may be required, along with the procurement activities. The latter requires a resources reallocation to implement the component.

Even if the dependency on the supplier is not a problem, when a problem occurs in a component-based system, it may not be easy to determine whether the responsibility of the problem (and, hence, of the solution, too) lies within the scope of the component, the system's scope, or the interaction between the system and the component [15].

## 3 ASSESSING COMPONENTS' QUALITY

### 3.1 Trust

After locating a component that may be a good candidate for inclusion in a particular system, a developer has to decide whether or not he should trust that component to do

the job. In general, trust is the confidence in the integrity of something. In the context of component-based development, the developer has to trust the component quality. This can be built using one (or a combination) of the techniques outlined in this section.

### 3.2 Certification

One way of creating trust in a component is through certification. Certification is the process of verifying property value levels associated with a component, and providing a certificate as evidence that the component conforms to well-defined standards and is adequate to fulfill a given set of requirements. Third party certification is generally viewed as a good way of bringing trust to projects where multiple partners have to interact. Therefore, it can be used to increase the trust between software components producers and users.

The CBSE community is still far from reaching a consensus on how certification should be carried out, what its requirements should be or even on who should perform it. Standards defining frameworks for assessing components are required before third party certification can be performed in a structured and repeatable fashion [16]. This view assumes that the certification is to be centered on the components themselves, rather than in the organizations that produce them.

Third party certification can face some difficulties, particularly due to the relative novelty of this area. As it happens in general, the span of the expertise of the certification body can be a handicap when it comes to very technical issues, because its know-how on the required details of the components is likely to be smaller than the one held by the component integrators. Even if the required expertise is held, sometimes it is not feasible for the independent certification body to simulate the operational environment in which the component will be embedded, as it changes from component buyer to component buyer. Only the component buyer can accurately assess the component's fitness for use in the target system.

It is important to establish certification boundaries. It has been suggested that we could bind software certification to the software's operational profile, rather than to the profile assumed by the software publisher [17] (although this is not feasible for the certification of COTS components, it might make sense in custom made ones). While this is a generic observation, it has a direct mapping for the special case of software components. The same component that is highly reliable in one system may fail in another one. This does not necessarily mean that there is a problem with the component. The component's failure may occur due to a reuse error.

The Ariane 5 disaster is a good example of a software reuse error. The accident was caused by a software fault in the on-board inertial reference system. It was the environment

around it that created a set of conditions for which the software did not have an adequate answer. The same software had been used with success in Ariane 4, but those conditions were not expected to occur there. While some authors use this example to claim that the usage of design by contract could have prevented this accident [18], others strongly disagree with this claim and the lack of hard evidences to back it up [19]. Nevertheless, a very subtle software reuse error provoked one of the most expensive software-related accidents ever.

In our opinion, design by contract [20] can be a helpful approach to the problem of establishing the boundaries of a certification. In a contract, both the involved entities (client and supplier) have some obligations to fulfill and some benefits to collect. The client may have to ensure some pre-conditions for the supplier. In return, the client may benefit from the post-conditions that result from the supplier's activity. The supplier may assume a set of pre-conditions before doing its activities and is obliged to guarantee a set of post-conditions when he finishes them. It is also possible (and desirable) to define invariants under which the component is expected to work. With design by contract, the certification can go as far as the pre-conditions, invariants and post-conditions allow it. This is a powerful tool for component suppliers, as they have the responsibility of assuring that the component conforms to a specific contract. The burden of determining whether or not that is enough is passed on to the client.

### 3.3 A standard for software components quality

In 1997, a committee was gathered to work on the development of a proposal for an IEEE standard on software components quality. The initiative was eventually suspended, since the committee came to a consensus that they were still far from getting to the point where the document would be a strong candidate for a standard. Still, it is worth analyzing the approach that they followed. The draft standard provided a method for assessing the quality of a software component source code package by manually examining its contents [21]. The result of that assessment is a triple that provides a numerical classification for code construction, module assembly and code and module verification and validation.

The grades on that triple were given with respect to a discrete integer scale that ranged from (1, 1, 1) to (9, 4, 8). Lower values in the triple indicated poor quality, while the higher ones stand for a higher quality. The grades were achieved by fulfilling a set of requirements. In order to achieve level  $x$  for one of those categories, all the requirements of level  $x-1$  had to be met, along with the specific requirements of level  $x$ . For example: source construction grade 5 can be achieved by adding source code peer examination, or some tool supported equivalent activity to level 4 code construction.

The candidate standard was mainly process oriented, in the

sense that it defined a set of steps that the component developer should follow to achieve a higher quality level. Clearly, this sort of approaches can increase one's confidence in the outcoming component. A parallel can be established with software process maturity models such as CMM [22], BOOTSTRAP [23], SPICE [24] and so on. However, a mature development process does not assure the quality of a component by itself.

### 3.4 Component specific quality issues

The standardization approach described in the previous sub-section has a significant drawback. While maintaining the normal challenges of in-house developed software quality assurance, the introduction of reusable software components developed elsewhere creates a set of extra concerns for the developers of component based software systems. In most cases, the source code of the components is not available. This reduces the set of quality assurance techniques the developers would have available otherwise, by preventing them from using verification techniques, for instance.

### 3.5 Testing Techniques

There are a number of possible testing techniques that can be applied in component certification. According to their nature, they may, or may not, be available to some of the players in the certification process. Table 1 summarizes the availability of each of the main techniques assuming that the component is shipped without its source code.

	Component Provider	Component Buyer	Certification Body
<b>White-box testing</b>	Yes	No	Yes
<b>Black-box testing</b>	Yes	Yes	Yes
<b>System level</b>	No	Yes	No
<b>Fault-Injection</b>			
<b>Operational</b>	No	Yes	No
<b>System testing</b>			

**Table 1 – Component Testing Techniques**

These testing techniques allow the certification of a component for usage under a specific set of constraints. They should be combined to support the achievement of higher quality in the development and integration of a software component.

Although the overall objective of testing is to ensure that the product (here, the software component) is free from failures and meets its user expectations, there are many limitations to prevent this objective from being fully achieved [25]. Although testing cannot prove the absence of problems in a software component it can help uncovering many of them, thus playing a major role in the achievement of trust in a software component.

#### 3.5.1 White-box testing

This technique is only available to entities with access to the component's source code: usually, the component provider, but it should also be made available to a third party certification body. Although it is not common for a

component's provider to give away its source code, it seems reasonable that a certification body should have access to it in order to perform the component verification with a wider range of available techniques such as white-box testing.

### 3.5.2 Black-box component testing

The selection of test for black-box testing cases can be made according to several criteria such as the system's operational profile and input domain partitioning techniques. It is also necessary to have an accurate oracle available, so that the tests' results can be compared with the expected ones. Black-box testing has some shortcomings, as it can fail to exercise important portions of the source code [26]. This makes the component buyer vulnerable to undesired behavior that may not be uncovered by testing.

### 3.5.3 System-level fault injection

This technique is used to determine how the system will tolerate component's failures. The basic idea is to corrupt the behavior of a component so that it produces incorrect results to given stimuli. This can be accomplished with instrumentation of the system's source code, by adding a corrupting handler to the usage of the components. The handler is responsible for tampering with the results of the component's operations. Rather than enabling the discovery of new problems with the component itself, this technique aims at increasing the system's robustness [26].

### 3.5.4 Operational system testing

Once the component is integrated in the system, system testing can be used to assess how the component and the system interact with each other. If any problem occurs with the component, whether it relates to some reliability issue or to the impact of the component in the overall performance of the system, the testers will have a more accurate assessment of the system's behavior in that event.

## 3.6 Component Wrapping

When evidence shows that a system will not completely tolerate a component, but its integration is still the most valid option, there are some techniques to circumvent this problem. The system can be modified, so that it deals with the problems the component is likely to create, but this is a centralized solution that will make the system more complex. Although the developer may not have the possibility of making modifications on the component, he can create a wrapper around it, so that the component's behavior is limited to what the system can tolerate. Wrapping can be made to control the component's input data, so that the problematic input data combinations can be avoided. Wrapping can also be used to filter the outputs generated by the component. The results from previous tests on the component are precious to the development of an adequate wrapper, as they help identifying the situations that the wrapper must prevent [26]. An example of component wrapping can be found in Figure 1, where a component wrapper is built with the help of a

ComponentStub class that is used as a filter between the original component and the interface provided by its wrapper.

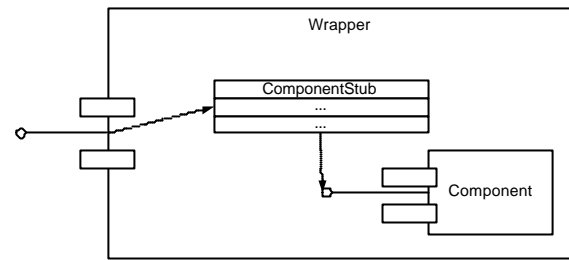


Figure 1 – Component Wrapper

## 4 CONCLUSIONS

While working on this overview of the state of the art of software components quality promoting and assurance techniques we have come across several quite promising research efforts. Unfortunately, in most cases little or no empirical evidence to back up the claims made on the potential benefits of each proposal was provided. In our opinion, there are two key reasons for this absence.

The first one is the recent emergence of this topic. Most initiatives have started very recently, so it is too soon for authors to have those empirical evaluations available.

A second motive would be the nature of empirical studies themselves. As we have seen in this overview, CBD presents some new challenges of its own, in what concerns the components quality evaluation. If we look at previous technological breakthroughs in software development, the same pattern can be found. First, a considerable effort is put into the development of the technology itself. The empirical evaluation of the claimed benefits only gains its momentum a few years after the technology is deployed. An example of this can be found with object oriented software development. Although object orientation had been around for a while, it was only during the 90's, particularly on the second half of the decade, that the focus on quality evaluation became more noticeable [27-29].

We expect the experience gathered by the empirical software engineering community will allow for a much quicker response to this new research topic. The editors of a recent book on CBSE have recognized some dismay caused by the lack of empirical studies to further the science and engineering practices of CBSE [30].

## 5 FUTURE WORK

Contributions are required in the field of components' quality and complexity evaluation. From a producer's point of view, the capacity to evaluate the quality of a component and its complexity may be determinant factors, not only in the human resources distribution, but also in the pricing of the produced components. From a consumer's point of view, our ability to make an informed buy is limited if no measure of the component's quality and complexity is

available, besides its price.

Many other questions are still unanswered. What sorts of learning difficulties are typical from the development with components? What gains of productivity are achieved through the usage of CBD, when compared to its object-oriented counterpart? Which are the benefits in terms of quality in the final systems that are attributable to this paradigm shift?

## 6 REFERENCES

- [1] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume I: Market Assessment of Component-Based Software Engineering," Software Engineering Institute, Technical Note CMU/SEI-2001-TN-007, May, 2000 2001.
- [2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press / Addison-Wesley, 1998.
- [3] R. M. Adler, "Emerging Standards for Component Software," *IEEE Computer*, vol. 28, pp. 68-77, 1995.
- [4] D. F. D'Souza and A. C. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*. Reading, Massachusetts: Addison Wesley Longman, 1998.
- [5] J. C. Seco and L. Caires, "A Basic Model of Typed Components," presented at 14th European Conference on Object-Oriented Programming (ECOOP'2000), Cannes, França, 2000.
- [6] O. Nierstrasz and D. Tsichritzis, "Object Oriented Software Composition," : Prentice Hall International, 1995.
- [7] J. D. Williams, "Raising Components," *Application Development Trends*, vol. 7, pp. 27-32, 2000.
- [8] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," IEEE, Standard IEEE Std 610.12-1990, 1990.
- [9] F. Bachman, L. Bass, C. Buhman, S. Cornella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering," Software Engineering Institute, Technical Report CMU/SEI-2000-TR-008, May 2000 2000.
- [10] M. Morisio, C. B. Seaman, A. T. Parra, and V. R. Basili, "Issues in COTS-based software development," presented at COTS Workshop - Continuing Collaborations for Successful COTS Development, Limerick, Ireland, 2000.
- [11] N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, vol. 20, pp. 199-206, 1994.
- [12] C. Abts, B. Boehm, and E. Bailey, "COCOTS: A COTS Software Integration Lifecycle Cost Model - Model Overview and Preliminary Data Collecting Findings," , Technical Report 2000.
- [13] B. W. Boehm, B. K. Clark, E. Horowitz, R. Maduchy, R. Selby, and C. Westland, "An overview of the COCOMO 2.0 software cost model," , 1995.
- [14] S. Yacoub, A. Mili, C. Kaveri, and M. Dehlin, "A Hierarchy of COTS Certification Criteria," presented at First Software Product Line Conference, Denver, Colorado, USA, 2000.
- [15] N. F. Schneidewind and A. P. Nikora, "Issues and Methods for Assessing COTS Reliability, Maintainability, and Availability," presented at Ensuring Successful COTS Development (ICSE'99), Los Angeles, California, USA, 1999.
- [16] B. Councill, "Third-Party Certification and Its Required Elements," presented at 4th ICSE Workshop on Component-Based Software Engineering, Toronto, Canada, 2001.
- [17] J. Voas, "Certifying Software for High-Assurance Environments," *IEEE Software*, pp. 48-54, 1999.
- [18] J.-M. Jézéquel and B. Meyer, "Put it in the Contract - The lessons of Ariane," *IEEE Computer*, vol. 30, pp. 129-130, 1997.
- [19] K. Garlington, "Critique of "Put it in the Contract: the Lessons of Ariane"," , 1998.
- [20] B. Meyer, "Contracts for components," in *Software Development Magazine*, 2000.
- [21] A. F. Ackerman, "Quality Grades for Software Component Source Code Packages - Informal Draft 0.1," Institute for Zero Defect Software 1997.
- [22] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability Maturity Model (Version 1,1)," *IEEE Software*, pp. 18-27, 1993.
- [23] G. Koch, *Process Assessment: The BOOTSTRAP Approach*: Butterworth-Heinemann Ltd, 1993.
- [24] M. Konrad, M. Paulk, and A. Graydon, "An Overview of SPICE's Model for Process Management," presented at Fifth International Conference on Software Quality, Austin, Texas, EUA, 1995.
- [25] J. S. Collofello, "Introduction to Software Verification and Validation," Software Engineering Institute - Carnegie Mellon University, SEI Curriculum Module SEI-CM-13-1.1, December 1988.
- [26] J. Voas, "Certifying Off-the-Shelf Software Components," *IEEE Software*, pp. 53-59, 1998.
- [27] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.
- [28] F. B. e. Abreu and W. M. U. o. Maryland), "Evaluating the Impact of Object-Oriented Design on Software Quality," presented at 3rd International Software Metrics Symposium (Metrics'96), Berlim, Alemanha, 1996.
- [29] B. Henderson-Sellers, *Object-Oriented Metrics - Measures of Complexity*. Upper Saddle River, NJ, EUA: Prentice Hall PTR, 1996.
- [30] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering - Putting the Pieces Together*. Boston, MA: Addison-Wesley, 2001.