Assessing Component-Based Systems

Alejandra Cechich[1], Mario Piattini[2], and Antonio Vallecillo[3]

[1] Departamento de Ciencias de la Computación
Universidad Nacional del Comahue, Argentina
acechich@uncoma.edu.ar

[2] Grupo Alarcos, Escuela Superior de Informática
Universidad de Castilla-La Mancha, Spain
Mario.Piattini@uclm.es

[3] Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
av@lcc.uma.es

Abstract. The last decade marked the first real attempt to turn software development into engineering through the concepts of Component-Based Software Development (CBSD) and Commercial Off-The-Shelf (COTS) components. The idea is to create high-quality parts and join them together to form a functioning system. The problem is that the combination of such parts does not necessarily result in a high-quality system. It is clear that CBSD affects software quality in several ways, ranging from introducing new methods for selecting COTS components, to defining a wide scope of testing principles and measurements. Today, software quality staff must rethink the way software is assessed, including all life-cycle phases—from requirements to evolution. Based on cumulated research efforts, the goal of this chapter is to introduce the best practices of current Component-Based Software Assessment (CBSA). We will develop and describe in detail the concepts involved in CBSA and its constituent elements, providing a basis for discussing the different approaches presented later in this book.

1 Introduction

The use of Commercial Off-The-Shelf (COTS) products as elements of larger systems is becoming increasingly commonplace. Component-Based Software Development (CBSD) is focused on assembling previously existing components (COTS or other non-developmental items) into larger software systems, and migrating existing applications towards component-based systems.

CBSD changes the focus of software engineering from one of traditional system specification and construction, to one requiring simultaneous consideration of the system's context and characteristics (such as user requirements, development costs and schedule, operating environments, etc.), the available products in the software marketplace, and viable architectures and designs. Furthermore, other engineering activities such as evaluation and acquisition processes, as well as contracting and licensing strategies, should be incorporated into the software development life-cycle.

The acceptance of CBSD carries along several challenges to the techniques and tools that have to provide support for it. Some of them are known problems, such as dealing with COTS production and integration, reducing complexity while improving reliability, and creating interaction models and supporting middleware. However, other challenges are new: the definition of metrics and quality attributes to measure software components and compositions, the definition of COTS selection processes, risk analysis, costs and effort estimation, etc. These are precisely the issues addressed by Component-Based Software Quality (CBSQ), which is emerging as a cornerstone discipline for a successful CBSD.

Within the last years, software researchers and practitioners have recognized the importance of quality in CBSD, and have started working on how quality has to be established, measured, and ensured both at the individual component level and at the system level, which constitute the two flips sides of the CBSQ coin. In the first case, the problems being addressed range from the precise definition, measurement, analysis, and evolution of COTS components, to the definition of tools, methods and processes for software component development [39, 40, 52, 53]. At the system level, software researchers and practitioners have started developing procedures, techniques and models for measuring and ensuring component-based systems quality [60].

During its relatively brief history, the field of CBSQ has experienced significant progress. However, CBSQ has not yet advanced to the point where there are standard measurement methods for any of the issues mentioned above, and few enterprises properly measure COTS component quality, if measured at all. Some efforts have started to define software metrics to guide quality and risk management in component-based systems, by identifying and quantifying various factors contributing to the overall quality of the system [9, 60, 61].

Based on cumulated research efforts, the goal of this chapter is to introduce the best practices of current CBSQ, in particular those that focus on Component-Based Software Assessment (CBSA). The objective is to provide an overview of the core elements of CBSA, and of its main issues.

The structure of this chapter is as follows. After this introduction, Section 2 introduces some of the issues that currently challenge CBSQ. Then, Section 3 deals with the core elements of CBSA at the individual component level, while Section 4 focuses on assessing component-based systems. Finally, Section 5 draws some conclusions.

2 CBSQ Issues

One can argue that ensuring the quality of component-based systems is much more difficult than is the case with manufactured goods. The problem is that the raw material—the software components—may well be of uncertain quality and their uses and behavior may be only partially known, hindering the effectiveness of possible quality assessment processes.

There are many potential problems that may affect the assessment of the quality properties of components and of component-based systems. One way of categorizing these problems is as follows:

- Issues related to components when considered as individual parts, that simply provide and require services (through their provided and required interfaces).
- Issues related to component interactions, which may have an explicit context dependencies about the operating systems or middleware platforms required, performance and other quality of service (QoS) requirements, or on other components [33].
- Issues related to component compositions (the combination of two or more software component yielding a new component).
- Other issues related to acquisition risks, such as vendor maturity level and trust, economic viability, legal issues, product evolution support, etc.

Let us discuss them more in detail. The first factor concerns uncertainty as to what constitutes the quality of a component when considered as an individual resource, i.e., a building block for constructing software systems. Unlike hardware components, for which there are catalogues and data-sheet available for describing their functional and extra-functional characteristics (to enable their proper re-use), the situation is not so bright when it comes to software components. First, there is no general consensus on the quality characteristics that need to be considered. Different authors (such as McCall [47] or Boehm [10]) and different organizations (such as ISO or IEEE) propose different (separate) classifications, but there is no clear agreement on which to use. The next issue is the lack of information about quality attributes provided by software vendors. The Web portals of the main COTS vendors show this fact—visit for instance Componentsource (`www.componentsource.com`) or Flashline (`www.flashline.com`).

In addition, there is an almost complete absence of any kind of metrics that could help measuring components' quality attributes objectively. Even worse, the international standards in charge of defining and dealing with the quality aspects of software products (e.g. ISO 9126 and ISO 14598) are currently under revision. The SquaRE project [3] has been created specifically to make them converge, trying to eliminate the gaps, conflicts, and ambiguities that they currently present. Furthermore, existing international standards provide very general quality models and guidelines, very difficult to apply in specific domains such as CBSD and COTS components.

The solution to the problem is not as obvious as counting with an agreed set of quality features that the component should exhibit, together with a set of associated metrics. External factors, such as users' requirements on the global system, architectural constraints, or context dependencies, may also have a strong influence on the component quality when assessed as a potential part of a larger system. This is why components may be completely satisfactory on most of their own attributes or features, but inadequate from a compositional viewpoint. Therefore, certain key quality features of a composition are also required.

The problems with the quality features of the composition rely both on the quality features of the individual components, the way they are interconnected, and on the quality features of the interaction media or middleware used. Thus, too many factors need to be considered, and in most cases the data may come from unreliable, external sources. Even if both the components and the middleware deliver the level of quality advertised by their vendors, what happens if higher levels of quality are required, or when the evolution of the system impose new levels of quality over time? Should the composer attempt to work together with the component or middleware vendors to improve the components' quality?

Another problem is due to the fact of re-using components. In CBSD, re-using not only means "using more than once", but also "using in different contexts", in order to achieve all the promised advantages of CBSD [34]. For example, most people would agree that using the components' clearly defined and documented interfaces is enough to accomplish error-free compositions. But this is only true if components remain confined to the domain of origin, and are never re-used across platforms or in other contexts. However, if the components are made available across platforms, then something needs to be done to resolve the inconsistencies and mismatches that happen at all levels: syntactically, semantically, and quality-wise.

The problem of ensuring component and component composition quality is exacerbated by the intrinsic nature of COTS components. The disadvantages associated with COTS-based design include the absence of source code and the lack of access to the software engineering artefacts originally used in the design of the components. Furthermore, whether you have built your system using COTS components from many vendors, or a single vendor has provided you with an integrated solution, many of the risks associated with system management and operation are not under your direct control [44].

Managing conflicting quality requirements is another issue specially interesting in the case of component-based systems. It is necessary to have an awareness of what could lead to inadequate component quality. "Fitness for use" implies that the appropriate level of system quality is dependent on the context. Determining the required quality is difficult when different users have different (even conflicting) needs. One might be tempted to state that the user requiring the highest component quality should determine the overall level of quality of the composed system. But in this way we may be over-imposing quality requirements to the system and to its individual components, which may be rarely required in most situations (e.g. in case of requirements imposed by occasional users, which do not represent the average system users). Thus, it is always necessary to properly balance conflicting requirements.

Finally, an important issue related to CBSA is about trust. Since we are going to incorporate to our system one part which has been developed elsewhere, our fears concerning quality could be reduced if we knew, for example, how the component was developed and who developed it. Furthermore, such a component would engender more confidence if someone other than the vendor or developer could certify, for example, that state-of-the-practice development and testing processes had been properly applied, that the code has no embedded malicious behavior [65].

As we can see, there are many issues related to establishing and assessing the quality of component-based systems. Most of them are general problems of quality assessment, common to all software systems, and known for years. But some of them are new problems, caused by the introduction of COTS components and CBSD. However, CBSD and COTS components not only have introduced new problems. They have greatly helped software development technology get mature enough to overcome many of the technical issues involved, putting us now in a position from where to start effectively tackling many of the quality aspects of software development. In this sense, CBSD offers an interesting technology from where many of the quality questions and problems can be formulated, and properly addressed.

3 Assessing COTS Components

In this Section we will discuss some of the main methods and techniques for the assessment of COTS components. In particular, we will cover COTS components evaluation, certification, and testing.

3.1 COTS Components Evaluation

Typically, the evaluation of COTS components consists of two phases: (1) COTS searching and screening, and (2) COTS analysis.

COTS component *search* is a process (i.e., a set of activities) that attempts to identify and find all potential candidate components that satisfy a set of given requirements, so they can be re-used and integrated into the application. The search is generally driven by a set of guidelines and selection criteria previously defined. Some methods propose a separate process for defining the selection criteria to be used, while others dynamically build a synergy of requirements, goals, and criteria [2, 13, 31, 37, 40, 42, 46, 52, 53, 54].

For example, the OTSO (Off-The-Shelf Option) method [39, 40] gradually defines the evaluation criteria as the selection process progresses, essentially decomposing the requirements for the COTS software into a hierarchical criteria set. Each branch in this hierarchy ends in an evaluation attribute: a well-defined measurement or a piece of information that will be determined during evaluation. This hierarchical decomposition principle has been derived from both Basili's GQM [5, 6] and the Analytic Hierarchy Process [59].

In the CAP (COTS Acquisition Process) proposal [52, 53], the first step is the identification of the criteria used for evaluating candidate alternatives. Requirements are translated into a taxonomy of evaluation criteria (called "Tailor & Weight Taxonomy"), and prioritized (or weighted) according to the Analytic Hierarchy Process, which also takes into account the stakeholders' interests.

COTS *screening* aims at deciding which alternatives should be selected for a more detailed evaluation. Decisions are driven by a variety of factors—foremost are several design constraints that help define the range of components. Some methods include *qualifying thresholds* for screening, i.e., defining and documenting the criteria and rationale for selecting alternatives. Some other methods estimate how much effort will be needed to actually apply all evaluation criteria to all COTS component candidates during the screening phase.

The results of the evaluation of the COTS alternatives is then used for making a decision. A COTS *analysis* phase starts, in general, from a set of ranked COTS software alternatives where the top-ranked alternative is measured against a set of final make-or-buy decision criteria, using for instance a *weighted scoring method*, which assigns weights to each criteria. However, measuring all applicable criteria for all COTS software alternatives can be expensive since: (*i*) there may be too many COTS software alternatives; (*ii*) the set of evaluation criteria could be quite large; and (*iii*) some of the criteria might be very difficult or expensive to measure (e.g., reliability).

The effectiveness of a COTS evaluation method depends on the expressiveness of the criteria selected for evaluation. A trade-off between the effectiveness of the evaluation criteria and the cost, time, and resource allocation of the criteria must be reached.

In COTS selection, both phases—search and screening, and COTS analysis—are based on statements about the requirements that need to be much more flexible than traditional ones, i.e. the specified requirements should not be so strict that either exclude all available COTS, or require large product modification in order to satisfy them. *Requirements Elicitation* is an activity by which a variety of stakeholders work together to discover, and increasingly define, requirements. Creative thinking is used by requirement elicitation teams to re-structure models, conceptualize, and solve problems. Some COTS evaluation proposals include processes to acquire and validate customer requirements [14, 42, 46], while others build a selection process based on iteratively defining and validating requirements [2, 43]. For instance, the proposal by Alves and Finkelstein [2], identifies first a set of high-level goals using traditional elicitation techniques, such as use-cases, which are then used for identifying possible COTS candidates in the marketplace. Then, new goals can be identified from these components, and the process starts again. The proposal by Franch *et al.* [14] uses a two-level selection process: the global level is responsible of combining the selection requirements from different areas in an organization, while the local level is responsible of carrying out the individual selection processes.

However, selecting COTS is not all about confronting component's services against technical and extra-functional requirements. As we mentioned earlier, there is also the issue of trust. Then, some selection methods also include a supplier selection process. The idea is to establish a supplier selection criteria, evaluate potential suppliers, rank them according to the agreed criteria, and select the best-fit supplier(s) [42, 44, 46, 71]. For instance, the V_RATE method [44] defines a taxonomy of vendor risk assessment, which produces a vendor-risk profile tied to real-world performance histories. This profile can be used to assess the risk associated with the use of a product in a particular environment, and to identify areas for additional risk-mitigation activities.

Finally, selection cannot be based on *exact* criteria in real applications. Usually, users' requirements are expressed in vague terms, such as "acceptable" performance, "small" size, or "high" adaptability. Measuring COTS components' quality attributes against such vague requirements is difficult. The QUESTA approach [31] addresses this issue by defining some mapping functions, that allow the assignments of values to such vague requirements.

3.2 Component Certification

Certification is the "procedure by which a third-party gives written assurance that a product, process, or service conforms to specified requirements" [17]. Current certification practices are basically process oriented, and usually require early life-cycle processes stressing the software product.

The particular nature of COTS components and component-based systems also present some challenges to traditional certification. Software certification must determine whether the software delivers all the functionality that is expected to deliver, and ensures that it does not exhibit any undesirable behavior. Ideally, the desirable behavior corresponds to specified requirements, although having to deal with missing requirements is unavoidable [70]. Therefore, certifying a software product, such as a component, is highly dependent on the quality of its requirements specification as well as on the quality of its documentation.

Documentation is one of the key issues in CBSD, specially at the component level. The black-box nature of components forces documentation to contain all details needed to understand the functionality provided by the component, describe how to deploy and install it, and how it should be connected to the rest of the components in the system. Furthermore, documentation should also describe the component's context dependencies and architectural constraints. Certification strongly relies on the component documentation, since it provides the "contract" with the requirements that the component behavior should conform to.

Another important problem related to component certification has to do with the application context. In practice, it is very difficult to certify that a component will behave in a correct manner *independently* from the context in which it will operate. However, certifying that a component works as expected within a given context or application can be more easily accomplished, since it is a matter of testing it in that particular environment.

Applying software testing techniques and conforming to standards on quality of components appear to be the most recommended approaches for certification. As a guidance, the ISO WD-12199-V4 standard [37] is directly applicable to COTS. It establishes: (1) requirements for COTS software products; (2) requirements for tests (including recommendations for the documentation of the tests); and (3) instructions on how to test a COTS against quality requirements (instructions for testing, in particular for third party testing).

Thus, in the context of CBSD, testing becomes the cornerstone process for certification.

3.3 Component Testing

Software component testing techniques focus on the expected behavior of the component, trying to ensure that its exhibited behavior is correct [8]. Black-box testing does not require any knowledge of the internals of the part being tested, and therefore is very appropriate for testing COTS components, whose source code is not available. Strategies for black-box testing include test cases generation by equivalence classes, error guessing and random tests. These three techniques rely only on some notion of the input space and expected functionality of the component being tested [8, 30, 50].

Component-based black-box testing techniques are also based on interface probing [41, 51]. A developer designs a set of test cases, executes a component on these test cases, and analyzes the outputs produced. There are a number of black-box methods for evaluating a component: manual evaluation, evaluation with a test suite, and automated interface probing. They will depend on how much the process can be automated, and the sort (and amount) of information available about the component. One of the major disadvantages of interface probing is that, frequently, a large number of test cases have to be created and analyzed. In addition, developers may frequently miss major component limitations and incorrectly assume certain component functionality. This may lead to incorrect use of the component when it is integrated with the final software system [51].

A different approach can be followed by automatically creating a system model from these individual requirements, and then automatically generating the test cases corresponding to them. In other words, from the requirements, a system model is automatically created with requirement information mapped to the model. This information can then be used to generate different tests [68].

Reflection and component metadata can also be effectively used for component testing. First, reflection enables a program to access its internal structure and behavior, providing a mechanism for interrogating and manipulating both. For instance, Sullivan has used reflection to load a class into a testing tool, extracting information about some its methods, and calling them with the appropriate parameters [55, 67]. Moreover, since the execution of methods in a class can create and use instances of other different classes, the approach is also helpful to test the integration of classes. Component metadata available in most existing component models can also be used to provide generic usage information about a component, which can be used for testing [15, 32].

CBSD also introduces some additional challenges to testing: components must fit into the new environment when they are re-used, often requiring real-time detection, diagnosis, and handling of software faults.

The BIT (Built-in Tests) technology developed within the Component+ European project [27] proposes embedding self-test functionality into software components, producing self-testable components that can help detect dynamic faults during run-time. This is a significant advance for improving software reliability and fault-tolerant capabilities in CBSD [36, 74]. There are some variations for building BIT-based COTS, such as including the complete test suite inside the components [75], or just embedding a minimal set of information, such as assertions, inside the components, which can be used at a later stage to define more complex tests [48]. In both cases there are some issues to consider. First, including complex tests means consuming space, which can be wasted if only a few number of tests are actually used. But if just a few information is included, or a small number of tests is embedded, additional external software need to be used to complement the provided tests, or to construct the tests from the component specifications. Current approaches try to show how built-in tests might be integrated by using flexible architectures, without requiring additional software, and hence minimizing the time spent on testing (see, e.g., [36]).

Finally, testing individual components is not enough for guaranteing the correctness of the final system. The way in which components are connected and interact introduce new properties to the system. Therefore, defining "good" stand-alone component tests may not be enough. Developing a foundation for testing component-based software is also a complicated issue, even though formal models of test adequacy for component-based software are being developed [58].

4 Assessment of Component-Based Systems

The assembly of an application based on third-party developed components does not necessarily assure a defect-free system, even if its constituent components haven been individually tested and certified. Furthermore, a component-based system needs to be tested on several aspects such as strategic risk analysis; risk-based process definition (for software components); risk-based design of component software (including initialization, fail-safe and fault-tolerant concepts, and built-in tests); risk-based component software analysis and design testing; software testing, including failure-mode and stress testing; documentation review; and assessment of hardware requirements [23].

In this Section we will concentrate on those quality issues than trespass the individual COTS components frontiers, and that arise only when the components are combined to form a functioning system.

4.1 COTS Integration and Software Architecture

In CBSD, most of the application developer's effort is spent integrating components, which provide their services through well-defined interfaces. It is extremely important that component services are provided through a standard, published interface to ensure interoperability [33].

Component architectures divide software components into requiring and providing services: some software components provide services, which are used by other components. The system's *Software Architecture* connects participating components, regulating component interactions and enforcing interaction rules. Software architectures and component technologies can be considered as complementary, and there is ample scope for their conceptual integration [73].

Some of the quality properties at the system level must be also enforced by the system's software architecture, which have to use the components' individual quality properties to deliver the required level of system quality. In addition, interaction patterns (usually encapsulated into *connectors* in the architecture) have also an strong effect on the global system quality, and their quality properties should also be incorporated into the quality assessment of the final system.

Another important issue when integrating components to build a functioning system deals with the mismatches that may occur when putting together pieces developed by different parties, usually unaware of each other. In this sense, Basili *et al.* [77] present a general classification of possible kinds of mismatches between COTS products and software systems, which includes architectural, functional, non-functional, and other issues. Incompatibilities are essentially failures of the components' interactions, so the authors claim that finding and classifying these interactions may help finding and classifying the incompatibilities.

Three aspects of inter-component interactions and incompatibilities can be considered: the kind of interacting components, the interoperability level required (syntactic, protocol, or semantic), and the number of components participating in the interaction. Different incompatibilities have different solutions, and the classes of problems are specific to the particular development phases. As another example, the work by Egyed *et al.* [28] combines architectural modelling with component-based development, showing how their mismatch-detection capabilities complement each other. The software architecture provides a high-level description of the components and their expected interactions. Architectural mismatches can be caused by inconsistencies between two or more constraints of different architectural parts being composed.

As important as determining architectural mismatches is calculating the integration effort. Decisions on component-based systems investments are strongly influenced by technological diversity: current technology is diverse and brings with it thousands of choices on components, with their opportunities and risks. Firms that avoid single technologies, and implement a diverse set of technologies and/or applications tend to focus their investments on innovative infrastructure, which might be the basis for new component-based system. Determining the effort required to integrate components into that infrastructure is essential to make a decision on migrating to CBSD. However, estimation is not straightforward. BASIS [4], for example, combines several factors in order to estimate the effort required to integrate each potential component into an existing architecture—architectural mismatches, complexity of the identified mismatch, and mismatch resolution. The final estimate describes the complexity of integrating a COTS component and is called the *difficulty of integration* factor. We should note that the BASIS approach also includes techniques for evaluating the vendor viability and the COTS component itself, determining a relative recommendation index for each product based on all factors.

Measuring complexity of the component's interactions also implies analyzing interfaces and messages. Some proposals in this direction propose to measure the complexity of the interactions—and their potential changes—by using metrics derived from information theory. For instance, the L-metric [18, 64] offers a static quantitative measure of the entropy caused by how the components interact as the system performs.

Finally, architectures used to build composite applications provide a design perspective for addressing interaction problems. Although little attention is paid to the evolvability of these architectures, it may also be necessary to estimate the effect on evolution in a system design in order to improve its robustness [24].

4.2 Cost Estimation

All CBSD projects require a cost estimation before the actual development activities can proceed. Most cost estimates for object-based systems are based on rules of thumb involving some size measure, like adapted lines of code, number of function points added/updated, or more recently, functional density [1, 26]. Using rules such as the system functional density, the percentage of the overall system functionality delivered per COTS component can also be determined, considering that the number of COTS components in a system should also be kept under a manageable threshold. However, in practical terms, rules such as functional density imply that (1) there must be a way of comparing one system design to another in terms of their functionality; (2) there must a way to split functionality delivered by COTS from that delivered from scratch; and (3) there must be a way to clearly identify different COTS functionalities.

Cost is not independent but a function of the enterprise itself, its particular software development process, the chosen solution, and the management and availability of the resources during the development project. The cost of updating and/or replacing components is highly situational, and depends on the organization's specific internal and external factors. Cost estimation should include the cost of project management, direct labor, and identification of affected software, affected data, and alternative solutions, testing and implementation. The accuracy of most cost estimates should improve during a project as the knowledge of the problem and the resources required for its solution increases. Some cost models are iterative, indicating change and re-evaluation throughout the solution stages. The most extended model on COTS integration cost—the COCOTS model [10, 22]—follows the general form of the COCOMO models, but with an alternative set of cost drivers addressing the problem of actually predicting the cost of performing a COTS integration task. Five groups of factors appear to influence COTS integration cost as indicated by the relatively large set of drivers: product and vendor maturity (including support services); customization and installation facilities; usability; portability; and previous product experience. In addition, COCOMO II model drivers such as architecture/risk resolution, development flexibility, team cohesion, database size, required reusability, etc., can be added to produce an integral COTS cost model. The goal is to provide the infrastructure for an easy integration of multiple COTS components. However, a cost model cannot easily put aside each of the problems encountered when integrating multiple COTS. In the migration to a component-based system, we would like to exercise the policy *Don't throw anything away, use and re-use as much as you can*. However, integrating existing software and COTS components as a whole is not a trivial task and may cause too many adaptation conflicts.

The cost model should provide mechanisms for dealing with the more common and more challenging integration problems, such as the following.

– Problems that happen when two or more components provide the same services with different representations.
– Problems that happen when a component has to be modelled by integrating parts of functionality from different components (or from other sources).

– Problems that happen when the resulting architecture does not cover the desired application requirements.
– Problems that happen when two or more integrated components with different implementations fail to interoperate.

While these problems might represent mainly one-time costs, the management-phase costs of CBSD recur throughout the system life-cycle, including during maintenance.

One of the most important problems of the maintenance process is the estimation and prediction of the related efforts. The different maintenance activities may present several related aspects: re-use, understanding, deletion of existing parts, development of new parts, re-documentation, etc. These aspects are relevant to all classes of systems, including object-oriented and component-based systems [29]. However, one of the most important distinguishing factors in CBSD is the separation of the interface and the implementation. The interface is usually realized by one or more components. Larger components may have more complex interfaces and represent more opportunity to be affected by change. Thus, components and component-based systems by nature are a source of changes stressed by potentially incompatible versions of components, which may compete with themselves.

When adapting a component-based system to changing requirements, invariant conditions are usually specified via constraint languages or specifically defined mechanisms. For instance, the context-based constraint (CoCon) mechanism [43] specifies one requirement for a group of indirectly associated components that share a context, which refers the state, situation or environment of each component. The mechanism requires that monitoring points (interception points between components) be determined for each invocation path. However, the main problem of this approach is precisely how to determine the context's property values of the components involved in this path.

In general, component technologies also impose some design constraints on software component suppliers and composers. These constraints are expressed as a component model that specifies required component interfaces and other development rules [33]. The focus of actual compositional rules is on syntactic aspects of composition, as well as on environmental and behavioral ones. For example, Ralf Reussner *et al.* [56] deal with the prediction of properties of compositions based on the properties of their basic components. Parameterized contracts that depend on the environment are defined, allowing timing behavior (and hence reliability) be analyzed by including timing behavior of the environmental services used by the component.

Reasoning abut behavioral composition is also as important as detecting interaction mismatches. The *Predictable Assembly from Certifiable Components* (PACC) [63] is an initiative at the Carnegie Mellon University–Software Engineering Institute (CMU-SEI) that tries to provide support for predicting behavioral properties of assemblies before the components are actually developed or purchased. *Prediction-Enabled Component Technology* (PECT) is a technology that supports composing systems from pre-compiled components in such a way that the quality of the system can be predicted before acquiring/building its constituents [35]. PECT makes analytic assumptions explicitly ensuring that a component technology satisfies the assumptions through a demonstration of "theoretical validity". It also ensures that predictions based on an analysis technology are repeatable through a demonstration of "empirical validity". Current research efforts focus on prediction of assembly properties such as latency and reliability.

Documentation is also another key to the success of the CBSD. In case of component-based systems, documentation should contain all the information about the system's structure and internal interconnections. For instance, *Ensembles* [72] is a conceptual language which enriches the system documentation by making explicit the links between component properties and component interactions. Ensembles also support graduated and incremental selection of technologies, products, and components.

In general, the composer does not need to understand all aspects of a component-based system, but should be able to identify its constituent components, and be able to search for components that may provide the functionality required by the architecture. In addition, it has been suggested that component documentation should be extensible, allowing the composer to insert details about component adaptations directly onto a system document library or annotating components. For example, annotations can be used to perform dependence analysis over these descriptions [66]. In addition, the system documentation should be closely tied to the composing tools, so the selected and tailored components are easily translated into reusable component libraries [16].

4.3 CBSD Risk Analysis

Software development is a quickly changing, knowledge-intensive business involving many people working in different phases and activities. Activities in software engineering are diverse, and the proportion of component-based systems is steadily growing. Organizations have problems identifying the content, location, and use of diverse components. Component composition requires access to tangible and intangible assets. Tangible assets, which correspond to documented, explicit information about the component, can vary from different vendors although usually include services, target platforms, information about vendors, and knowledge for adaptation. Intangible assets, which correspond to tacit and undocumented explicit information, consist of skills, experience, and knowledge of an organization's people.

Although the Risk Management Paradigm [76] continues to be useful as an overall guide to risk analysis, the majority of risk management studies deal with normative techniques for managing risk. Software risks may come from different dimensions: (1) environmental contingencies such as organizational environment, technologies, and individual characteristics; and (2) risk management practices such as methods, resources, and period of use [7, 57].

A few studies have classified software risk items, including CBSD risk analysis. These studies consider CBSD risks along several dimensions, providing some empirically-founded insights of typical cases and their variations. For example, the BASIS technique [4] focuses on reducing risks in CBSD by means of several integrated processes. One of them, the Component Evaluation Process [54], tries to reduce the risk of selecting inappropriate COTS components. Another process aims at reducing the risk of downstream integration problems through early assessment of the compatibility between a chosen COTS product and an existing system. Finally, a third process tries to reduce risks throughout the development life-cycle by defining built-in checkpoints and recommending measurement techniques. A vendor analysis is carried out along with these three processes.

Software project managers need to make a series of decisions at the beginning of and during projects. Because software development is such a complex and diverse process, predictive models should guide decision making for future projects. This requires having a metrics program in place, collecting project data with a well-defined goal in a metrics repository, and then analyzing and processing data to generate models. It has also been shown how metrics can guide risk and quality management, helping reduce risks encountered during planning and execution of CBSD [60, 61]. Risks can include performance issues, reliability, adaptability, and return on investment. Metrics in this case are used to quantify the concept of quality, aiming at investigating the tradeoffs between cost and quality, and using the information gained to guide quality management. The primary considerations are cost, time to market, and product quality.

4.4 Software Product Lines

In this last Section we will discuss the quality assessment issues of one of the most promising approaches for developing component-based systems, specially in some well-defined application domains: *software product lines* [11, 21, 69].

A software product line is a "set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market or mission and that are developed from a common set of core assets in a prescribed way" [21]. So, within the product line, it is necessary to determine what products are needed, how to develop those products, how to manage the evolution of them, and how to re-use them. The idea is to achieve high levels of intra-organizational re-use. Products become "components" in a product line approach, where product-line architectures are the basis for software component re-use.

Software product lines not only help developing software applications, but also help dealing with some quality issues in a controlled way. Of course, dealing with quality assessment and risk management in product lines involves a whole world of new methods and techniques—from requirement analysis to testing. However, the common nature of SPL facilitates the treatment of some of their quality issues.

For example, Chastek *et al.* [20] specify requirements for a product line using a model whose primary goal is to identify and analyze opportunities for re-use within requirements. The model has two main characteristics: (1) it specifies the functionality and quality attributes of the products in the product line;

and (2) its structure shows decisions about commonalities across the product line. The work products are based on object modelling, use-case modelling, and feature-modelling techniques. They form the basis of a systematic method for capturing and modelling the product line requirements, where five types of analysis are introduced: commonality and variability analysis, model consistency analysis, feature interaction analysis, model quality analysis, and requirements priority analysis.

As another example of mastering quality in software product lines, the framework published in [21] describes the essential practice areas for software engineering, technical management, and organizational management. A *practice area* is a body of work or a collection of activities that an organization must master to successfully carry out the essential work of a product line. The software engineering practice areas include those practices needed to apply the appropriate technology to create and evolve both core assets and products. The framework seems to have some similarities when compared to CMMI models [38, 62], where the major organizing element is the *process area*. Here, a process area is a group of related activities that are performed collectively to achieve a set of goals. Hence, the wide acceptance of the "capability models" has turned research into developing maturity models for product lines [12], which aims at providing guidelines for dealing with intra-organizational re-use.

On the other hand, testing in product lines environments involves testing assets as well as products. But a software product line organization must maintain a number of complex relationships. There are relationships between the group that develops the core assets and those who develop products, between the general product line architecture and specific product architectures, between versions of these architectures and versions of products. As a general rule, McGregor proposes structuring the relationships among test artifacts to mirror the structure of the relationships among production artifacts, so a more efficient test implementation can be produced [49]. In this case, the test architect carries the responsibility for achieving the quality properties defined in the test plan. The architect defines the structure of the test software and defines the basic test assets. This test software architecture can then be specialized to the specific environment, quality properties, and constraints of a given product.

Finally, the use of metrics has proven to be helpful in assessing particular situations to guide architectural decisions. For product line architectures (PLA), the work reported by Dincel *et al.* [25] focuses on PLA-level metrics for system evolution. In particular, they provide an incremental set of metrics that allows an architect to make informed decisions about the usage levels of architectural components, the cohesiveness of the components, and the validity of product family architectures. However, the experience with these metrics is limited, requiring further validation.

5 Conclusions

There are many benefits derived from CBSD, which has become one of the key technologies for the effective construction of large, complex software systems in timely and affordable manners. However, the adoption of component-based development carries along many changes that touch beliefs and ideas considered to be core to most organizations. These adjustments, and the approaches taken to resolve contention, can often be the difference between succeeding and failing in CBSD undertaking.

So far, most of the efforts from the Software Engineering community have focused on the technical and technological issues of CBSD. But once the situation starts to be stable at these levels, and component-based software commences to be effectively used in commercial applications and industrial environments, the focus is changing towards the quality aspects of CBSD.

In this chapter we have tried to provide an overview of the core concepts of component-based software quality (CBSQ), and the main issues related to component-based software assessment (CBSA). From here, the chapters in this book present some of the current initiatives for dealing with CBSQ issues, from COTS assessment to product lines maturity levels. However, much work still remain to be done in this emerging area in order to effectively address the many difficult challenges ahead.

Acknowledgments

This work is partially supported by research projects and grants CICYT TIC2002-04309-C02-02, CICYT TIC 2000-1673-C06-06 (DOLMEN/MEDEO), UNComa 04/E048, and by CYTED project RITOS2 (Red Iberoamericana de Tecnologías del Software para la década del 2000).

References

1. C. Abts. COTS-Based Systems (CBS) Functional density – A Heuristic for Better CBS Design. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 1–9, 2002.
2. C. Alves and A. Finkelstein. Challenges in COTS Decision-Making: A Goal-Driven Requirements Engineering Perspective. In *Proc. of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE'02*, 2002.
3. M. Azuma. SquaRE, the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality. In *ESCOM (European Software Control and Metrics conference)*, April 2001. `http://www.escom.co.uk/conference2001/papers/azuma.pdf`
4. K. Ballurio, B. Scalzo, and L. Rose. Risk Reduction in COTS Software Selection with BASIS. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 31–43, 2002.
5. V. Basili. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. Tech. Report CS-TR-2956, University of Maryland, 1992.
6. V. Basili and H. Rombach. Tailoring the Software Process to Project Goals and Environments. In *Proc. of ICSE'87*, IEEE CS Press, pp. 345–357, 1987.
7. V. R. Basili and B. Boehm. COTS-based systems top 10 list. *IEEE Software*, 34(5):91–93, 2001.

8. B. Beizer. *Black-Box Testing. Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.
9. M.F. Bertoa and A. Vallecillo. Quality Attributes for COTS Components. In *Proc. of ECOOP 2002 QAOOSE Workshop*, June 2002.
10. B. Boehm, C. Abts, and E. Bailey. COCOTS Software Integration Cost Model: an Overview. In *Proc. of the California Software Symposium*, 1998.
11. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.
12. J. Bosch. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *Proc. of the Second Software Product Line Conference*, pp. 257–271, 2002.
13. P. Bose. Scenario-Driven Analysis of Component-Based Software Architecture Models. In *Proc. of the First Working IFIP Conference on Software Architecture*, 1999.
14. X. Burgués, C. Estay, X. Franch, J.A. Pastor, and C Quer. Combined Selection of COTS Components. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 54–64, 2002.
15. A. Cechich and M. Polo. Black-box Evaluation of COTS Components using Aspects and Metadata. In *Proc. of the 4th International Conference on Product Focused Software Process Improvement*, Springer-Verlag, pp. 494–508, 2002.
16. A. Cechich and M. Prieto. Comparing Visual Component Composition Environments. In *Proc. of the XXII International Conference of the Chilean Computer Science Society*, IEEE Computer Society Press, 2002.
17. Brussels CEN. EN 45020:1993 General Terms and Definitions Concerning Standardization and re-lated activities, 1993.
18. N. Chapin. Entropy-Metric For Systems With COTS Software. In *Proc. of the 8th IEEE Symposium on Software Metrics*, IEEE Computer Society Press, 2002.
19. M. Charpentier. Reasoning about Composition: A Predicate Transformer Approach. In *Proc. of Specification and Verification of Component-Based Systems Workshop, OOPSLA 2001*, pp. 42–49, 2001.
20. G. Chastek, P. Donohoe, K. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Tech. Report CMU/SEI-2001-TR-001, Carnegie Mellon, Software Engineering Institute, 2001.
21. P. Clemens and L. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.
22. COCOTS. COnstructive COTS Model. `http://sunset.usc.edu/research/COCOTS/`, 2001.
23. W. T. Council. Third-Party Testing and the Quality of Software Components. *IEEE Software*, 16(4):55–57, 1999.
24. L. Davis and R. Gamble. Identifying Evolvability for Integration. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 65–75, 2002.
25. E. Dincel, N. Medvidovic, and A. van der Hoek. Measuring Product Line Architectures. In *Proc. of the International Workshop on Product Family Engineering (PFE-4)*, 2001.
26. J. Dolado. A Validation of the Component-Based Method for Software Size Estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.
27. EC. IST-1999-20162, Component+. `www.component-plus.org`, 2002.
28. A. Egyed, N. Medvidovic, and C. Gacek. Component-based perspective on software mismatch detection and resolution. *IEE Software Engineering*, 147(6):225–236, 2000.

29. F. Fioravanti and P. Nesi. Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 27(12):1062–1084, 2001.
30. J. Gao, K. Gupta, S. Gupta, and S. Shim. On Building Testable Software Components. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 108–121, 2002.
31. W. Hansen. A Generic Process and Terminology for Evaluating COTS Software - The QESTA Process. `http://www.sei.cmu.edu/staff/wjh/Qesta.html`.
32. M. Harrold. Using Component Metadata to Support the Regression Testing of Component-Based Software. Tech. Report GIT-CC-01-38, College of Computing, Georgia Institute of Technology, 2001.
33. G. Heineman and W. Council. *Component-Based Software Engineering - Putting the Pieces Together*. Addison-Wesley, 2001.
34. C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
35. S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging Predictable Assembly with Prediction-Enabled Component Technology. Tech. Report CMU/SEI-2001-TR-024, Carnegie Mellon, Software Engineering Institute, 2001.
36. J. H´ornestein and H. Edler. Test Reuse in CBSE Using Built-in Tests. In *Proc. of the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems. Workshop on Component-based Software Engineering*, 2002.
37. ISO International Standard ISO/WD 121199. ISO/WD 12199 - V4.: Software Engineering - Software product evaluation - Requirements for quality of Commercial Off The Shelf software products (COTS) and in-structions for testing, 2001.
38. L. Jones and A. Soule. Software Process Improvement and Product Line Practice: CMMI and the Framework for Product Line Practice. Tech. Report CMU/SEI-2002-TN-012, Carnegie Mellon, Software Engineering Institute, 2002.
39. J. Kontio. OTSO: A Systematic Process for Reusable Software Component Selection. Tech. Report UMIACS-TR-95-63, University of Maryland, 1995.
40. J. Kontio, S. Chen, and K. Limperos. A COTS Selection Method and Experiences of its Use. In *Proc. of the 20th Annual Software Engineering Workshop, NASA Software Engineering Laboratory*, 1995.
41. B. Korel. Black-Box Understanding of COTS Components. In *Proc. of the 7th International Workshop on Program Comprehension*, IEEE Press, pp. 92–99, 1999.
42. D. Kunda and L. Brooks. Applying Social-Technical Approach for COTS Selection. In *Proc. of the 4th UKAIS Conference*, University of York, 1999.
43. A. Leicher and F. B´ubl. External Requirements Validation for Component-Based Systems. In *Proc. of CAiSE 2002*, LNCS 2348, Springer-Verlag, pp. 404–419, 2002.
44. H. F. Lipson, N. R. Mead, and A. P. Moore. Can We Ever Build Survivable Systems from COTS Components? In *Proc. of CAiSE 2002*, LNCS 2348, Springer-Verlag, pp. 216–229, 2002.
45. J. Magee, N. Dualy, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proc. of the 5th European Software Engineering Conference*, LNCS 989, Springer-Verlag, pp. 137–153, 1995.
46. N. Maiden, H. Kim, and C. Ncube. Rethinking Process Guidance for Selecting Software Components. In *Proc. of the First International Conference on COTS-Based Software Systems*, Springer-Verlag, pp. 151–164, 2002.
47. J. McCall, P. Richards, and G. Walters. Factors in software quality, volume III: Preliminary handbook on software quality for an acquisition manager. Tech. Report RADC-TR-77-369, vol. III, Hanscom AFB, MA 01731, 1977.

48. E. Martins, C.M. Toyota, and R.L. Yanagawa. Constructing Self-Testable Software Components. In *Proc. of the 2001 International Conference on Dependable Systems and Networks*, pp. 151–160, 2001.
49. J. McGregor. Testing a Software Product Line. Tech. Report CMU/SEI-2001-TR-022, Carnegie Mellon, Software Engineering Institute, 2001.
50. De Millo. *Software Testing and Evaluation.* Benjamin/Cumming Publishing Co, 1987.
51. C. Mueller and B. Korel. Automated Black-Box Evaluation of COTS Components with Multiple-Interfaces. In *Proc. of the 2nd International Workshop on Automated Program Analysis, Testing, and Verification, ICSE 2001*, 2001.
52. M. Ochs, D. Pfahl, G. Chrobok-Diening, and Nothhelfer-Kolb. A COTS Acquisition Process: Definition and Application Experience. Tech. Report IESE-002.00/E, Fraunhofer Institut Experimentelles Software Engineering, 2000.
53. M. Ochs, D. Pfahl, G. Chrobok-Diening, and Nothhelfer-Kolb. A Method for Efficient Measurement-based COTS Assessment and Selection - Method Description and Evaluation Results. Tech. Report IESE-055.00/E, Fraunhofer Institut Experimentelles Software Engineering, 2000.
54. S. Polen, L. Rose, and B. Phillips. Component Evaluation Process. Tech. Report SPC-98091-CMC, Software Productivity Consortium, 1999.
55. M. Polo. Automating Testing of Java Programs using Reflection. In *Proc. of the ICSE 2001 Workshop WAPATV*, IEEE Press, 2001.
56. R. Reussner and H. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In *Proc. of the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems. Workshop on Component-based Software Engineering*, 2002.
57. J. Ropponen and K. Lyytinen. Components of Software Development Risk: How to Address Them? A Project Management Survey. *IEEE Transactions on Software Engineering*, 26(2):98–112, 2000.
58. R. Rosenblum. Adequate Testing of Component-Based Software. Tech. Report 97-34, Department of Information and Computer Science, University of California, Irvine, 1997.
59. T.L. Saaty. *The Analytic Hierarchy Process.* McGraw-Hill, 1990.
60. S. Sedigh-Ali, A. Ghafoor, and R. Paul. Metrics-Guided Quality Management for Component-Based Software Systems. In *Proc. of the 25th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, pp. 303–310, 2001.
61. S. Sedigh-Ali, A. Ghafoor, and R. Paul. Software Engineering Metrics for COTS-Based Systems. *IEEE Computer Magazine*, pp. 44–50, May 2001.
62. SEI. CMMI Product Suite. `http://www.sei.cmu.edu/cmmi/products`.
63. SEI. Predictable Assambly from Certifiable Components (PACC). `http://www.sei.cmu.edu/pacc/index.html`.
64. M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, and H. Ammar. Information Theoretic Metrics for Software Architectures. In *Proc. of the 25th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, 2001.
65. M. Sparling. Lessons Learned Through Six Years of Component-Based Development. *Communications of the ACM*, 43(10):47–53, 2000.
66. J. Stafford and L. Wolf. Annotating Components to Support Component-Based Static Analyses of Software Systems . Tech. Report CU-CS-896-99, University of Colorado at Boulder, 1999.

67. G. Sullivan. Aspect-Oriented Programming using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, 2001.
68. L. Tahat. Requirement-Based Automated Black-Box Test Generation. In *Proc. of the 25th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, pp. 489–495, 2001.
69. A. van der Hoek. Capturing Product Line Architectures. In *Proc. of the 4th International Software Architecture Workshop*, 2000.
70. J. Voas. Certifying Software for High-Assurance Environments. *IEEE Software*, 16(4):48–54, 1999.
71. K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Addison-Wesley, 2002.
72. K. Wallnau and J. Stafford. Ensembles: Abstractions for a New Class of Design Problem. In *Proc. of the 27th Euromicro Conference*, IEEE Computer Society Press, 2001.
73. K. Wallnau, J. Stafford, S. Hissam, and M. Klein. On the Relationship of Software Architecture to Software Component Technology. In *Proc. of the ECOOP 6th International Workshop on Component-Oriented Programming (WCOP6)*, 2001.
74. Y. Wang and G. King. A European COTS Architecture with Built-in Tests. In *Proc. of OOIS 2002*, LNCS 2425, Springer-Verlag, pp. 336–347, 2002.
75. Y. Wang, G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross. On Built-in Test Reuse in Object-Oriented Framework Design. *ACM Journal on Computing Surveys*, 32(1), 2000.
76. R. Williams, J. Walker, and A. Dorofee. Putting Risk Management into Practice. *IEEE Software*, 14(3):75–82, 1997.
77. D. Yakimovich, J. Bieman, and V. Basili. Software architecture classification for estimating the cost of COTS integration. In *Proc. of ICSE'99*, pp. 296–302, 1999.