UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org
Vol. IV, No. 4, August 2003

**UPGRADE** is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

**Publisher**
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by NOVÁTICA <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (Asociación de Técnicos de Informática <http://www.ati.es/>).
UPGRADE is also published in Spanish (full issue printed, some articles online) by NOVÁTICA, and in Italian (abstracts and some articles online) by the Italian CEPIS society ALSI <http://www.alsi.it> and the Italian IT portal Tecnoteca <http://www.tecnoteca.it/>.
UPGRADE was created in October 2000 by CEPIS and was first published by NOVÁTICA and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>).

**Editorial Team**
Chief Editor: Rafael Fernández Calvo, Spain <rfcalvo@ati.es>
Associate Editors:
• François Louis Nicolet, Switzerland, <nicolet@acm.org>
• Roberto Carniel, Italy, <carniel@dgt.uniud.it>

**Editorial Board**
Prof. Wolffried Stucky, CEPIS President
Fernando Piera Gómez and
Rafael Fernández Calvo, ATI (Spain)
François Louis Nicolet, SI (Switzerland)
Roberto Carniel, ALSI – Tecnoteca (Italy)

**English Editors:** Mike Andersson, Richard Butchart, David Cash, Arthur Cook, Tracey Darch, Laura Davies, Nick Dunn, Rodney Fennemore, Hilary Green, Roger Harris, Michael Hird, Jim Holder, Alasdair MacLeod, Pat Moody, Adam David Moss, Phil Parkin, Brian Robson.

**Cover page** designed by Antonio Crespo Foix, © ATI 2003

**Layout:** Pascale Schürmann

E-mail addresses for editorial correspondence: <nicolet@acm.org> and <rfcalvo@ati.es>

E-mail address for advertising correspondence: <novatica@ati.es>

**Upgrade Newslist** available at <http://www.upgrade-cepis.org/pages/editinfo.html#newslist>

**Copyright**
© NOVÀTICA 2003. All rights reserved. Abstracting is permitted with credit to the source. For copying, reprint, or republication permission, write to the editors.

The opinions expressed by the authors are their exclusive responsibility.

ISSN 1684-5285

Next issue (Oct. 2003):
**"e-Learning – Borderless Education"**

# Software Engineering – State of an Art

Guest Editor: **Luis Fernández-Sanz**

## Joint issue with NOVÁTICA

2 Presentation: Software Engineering. A Dream Coming True? – *Luis Fernández-Sanz*

*The guest editor presents the issue, that focuses on a really broad field like Software Engineering (SE) which has been driving the evolution of software development since the late sixties of the past century. The papers cover different areas of interest related to the application of engineering principles to software development and maintenance. As usual, a list of useful references is also included for those interested in knowing more about this subject.*

5 Software Project Management. Adding Stakeholder Metrics to Agile Projects – *Tom Gilb*

*In this article the author offers an analysis of the implications of the new agile methods in the field of software development.*

10 Model-Driven Development and UML 2.0. The End of Programming as We Know It? – *Morgan Björkander*

*This paper focuses on the idea of a truly model-driven software and analyses the influence that the new version of UML (Unified Modeling Language) is having on this process.*

15 Component-Based Software Engineering – *Alejandra Cechich and Mario Piattini-Velthuis*

*This paper studies the important role that components play in the field of Software Engineering.*

21 An Overview of Software Quality – *Margaret Ross*

*The author reviews important issues concerning quality in software development and also deals with the issues of users with disabilities and the influence of legislation regulating this aspect.*

26 Lessons Learned in Software Process Improvement – *José-Antonio Calvo-Manzano Villalón, Gonzalo Cuevas-Agustín, Tomás San Feliu-Gilabert, Antonio de Amescua-Seco, Mª Magdalena Arcilla-Cobián, and José-Antonio Cerrada-Somolinos*

*This paper describes the lessons learned by SOMEPRO, a Software Engineering R & D group in the Universidad Politécnica de Madrid, in more than ten software process improvement projects.*

30 A New Method for Simultaneous Application of ISO/IEC 15504 and ISO 9001:2000 in Software SME's – *Antònia Mas-Pichaco and Esperança Amengual-Alcover*

*The authors offer their view, based on practical experiences, of the always thorny problem of applying best software development practices to organizations where resources are especially limited.*

37 Empirically-based Software Engineering – *Martin Shepperd*

*This paper presents an overview of empirical Software Engineering and its implications for practitioners and researchers in four areas (object-orientation, inspections, formal specification and project failure factors.)*

42 Software Engineering Professionalism – *Luis Fernández-Sanz and María-José García-García*

*The aim of this paper is to provide a brief overview of what goes into making up our true perception of software engineers as specialised professionals within the field of Information Technologies.*

47 Searching for the Holy Grail of Software Engineering – *Robert L. Glass*

*In this article, the author defends eclecticism in development methods and the contribution that Software Engineering should make in this respect whenever the nature of a project demands flexible methods in order to be successful.*

49 Free Software Engineering: A Field to Explore – *Jesús M. González-Barahona and Gregorio Robles*

*This article analises the existing points of contact between Software Engineering and the development of free software, and puts forward a few future lines of research in this respect.*

# Component-Based Software Engineering

*Alejandra Cechich and Mario Piattini-Velthuis*

*As Component-Based Software Development (CBSD) starts to be used effectively, building systems requires new methodologies and processes not only for development and maintenance, but also for other lifecycle phases that are strongly affected. For example, some software vendors have begun to successfully sell and license commercial off-the-shelf (COTS) components, and this fact leads to a considerable number of components being available for use. Thus, requirements engineering techniques have to change to deal with more flexible requirements to provide a match between stakeholder requirements and COTS component's services. In addition to changes in activities such as composition and component specification, that are specific to Component-Based Software Engineering (CBSE), there are also a number of managerial issues that require change. Many of these issues are not yet established in practice or even developed. The main goal of this article is to present some characteristics of a CBSD and discuss some of the current issues associated with applying CBSE.*

**Keywords:** Component-based Software Development, component integration, component quality assessments, COTS selection.

## 1 Introduction

Software systems are becoming increasingly complex and providing more functionality. To deal with such systems, Component-Based Software Development (CBSD) advocates the use of pre-fabricated components, perhaps developed at different times, by different people, and possibly with different uses in mind [1][2]. The goal, once again, is the reduction of development times, costs, and efforts, while improving the quality of the final application due to the (re)use of software components already developed, tested and validated. The discipline of Component-Based Software Engineering (CBSE) provides methods, models and techniques for the developers and composers of component-based systems.

*Components*, the heart of CBSD, are defined in various ways and from different points of view. The definition most widely accepted considers a software component as "*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*" [3]. This definition demonstrates the important relationship between software components and a component model.

A *Component model* provides an abstract description giving component-based application developer a uniform view of how to build individual components as well as how components communicate and interact with each other. Thus, component models imply dealing with component compositions, i.e. "*the combination of two or more software components yielding a new component behaviour at a different level of abstraction*" [3].

The characteristics of the composition's behaviour are determined by the components being combined, and by the way they are combined. A component model enables composition by defining an interaction standard that promotes unambiguously specified interfaces. An interface is a collection of operations that are used to specify a service of a component. In an interface the semantics of each operation semantic are specified. This specification serves both to providers implementing the interface and to clients using the interface. The interface of a

***Alejandra Cechich*** is a PhD student at the Universidad de Castilla-La Mancha, Spain, having obtained a MSc from the Universidad Nacional del Sur, Argentina. She is an Associate Professor in the Department of Computer Science at the Universdad of Comahue, Neuquén, Argentina, and her research interests are software design, conceptual modelling, software quality, and information systems. <acechich@uncoma.edu.ar>

***Mario Piattini Velthuis*** obtained a MSc and a PhD in Computer Science from the Universidad Politécnica de Madrid, Spain, and a MSc in Psychology from the UNED, Spain. He is an Information System Audit and Control Association (ISACA) Certified Information System Auditor and Certified Information Security Manager, and is a Full Professor at the Department of Computer Science at the Universidad de Castilla-La Mancha, in Ciudad Real, Spain. The author of several books and papers on databases, Software Engineering and information systems, he leads the ALARCOS research group specializing in information system quality. He is member of the Board of ATI (Asociación de Técnicos de Informática, Spain) and co-editor of the Data Base section of its journal Novática. His research interests are software quality, advanced database design, metrics, software maintenance, information system audit and security.
<Mario.Piattini@uclm.es>

© Novática

component is important for the composition and customisation of components by users, allowing them to find suitable components and to understand their purpose, functionality, usage and restrictions [1][3].

The use of commercial off-the-shelf (COTS) products as elements of larger systems is becoming increasingly commonplace. CBSD is focused on assembling previously existing components (COTS and other non-developmental items) into larger software systems, and migrating existing systems towards a component based approach. The impact of this fundamental change is profound. Not only must engineering activities such as requirements specification change, but so must the acquisition processes and contracting strategies. Thus, the adoption of CBSD brings with it many changes that challenge beliefs and ideas considered core to most organisations. A COTS-Based Development comprises many facets, where the features of each activity implicitly define the development changes that impact the way a project team behave. The activities require developers to accept the importance of working with components as encapsulated black boxes and not attempt to repeatedly rebuild them. Conversely, developers building components have to balance the goal of reuse with application requirements looking at the changes to testing strategies brought about by CBSE.

In short, CBSE changes the focus of Software Engineering from the traditional requirement of system specification and construction to requiring simultaneous consideration of the system context (system characteristics such as requirements, cost, schedule, operating and support environments), capabilities of products in the marketplace, and viable architectures and designs.

In section 2 of this article we present a component-based development cycle discussing how COTS components affect development. Section 3 then introduces some open issues on CBSE. Conclusions are addressed in the final section.

## 2 Component-Based Software Development Cycle

The development cycle of a component-based system is different from the traditional ones, i.e. the waterfall, iterative, spiral and prototype based models. Figure 1 (from [4]) shows a comparison between the traditional waterfall model and the component-based development model. The different steps in the component development process are:

1. Find components (COTS and non-COTS),
2. Select the components that are most suitable to the system,
3. Create a composed solution that integrates the selected components,
4. Adapt the selected components so they suit the existing component model,
5. Compose or deploy the product, and
6. Replace old versions or maintain COTS and non-COTS parts of the system.

Gathering requirements and design in the waterfall model corresponds to finding and selecting components. Implementation, test and release correspond to create, adapt, deploy and replace. The first two steps – component search and selection – are the most crucial in the component-based cycle. Non-optimal component software used in the development of a system can become extremely costly for a software organisation. Here long-term decisions on which components will be used in a software system are made.

Apart from the general reuse problems (selection, integration, maintenance, etc.), COTS products have their own specific problems:

- Incompatibility: COTS components may not have the exact functionality required; moreover, a COTS product may not

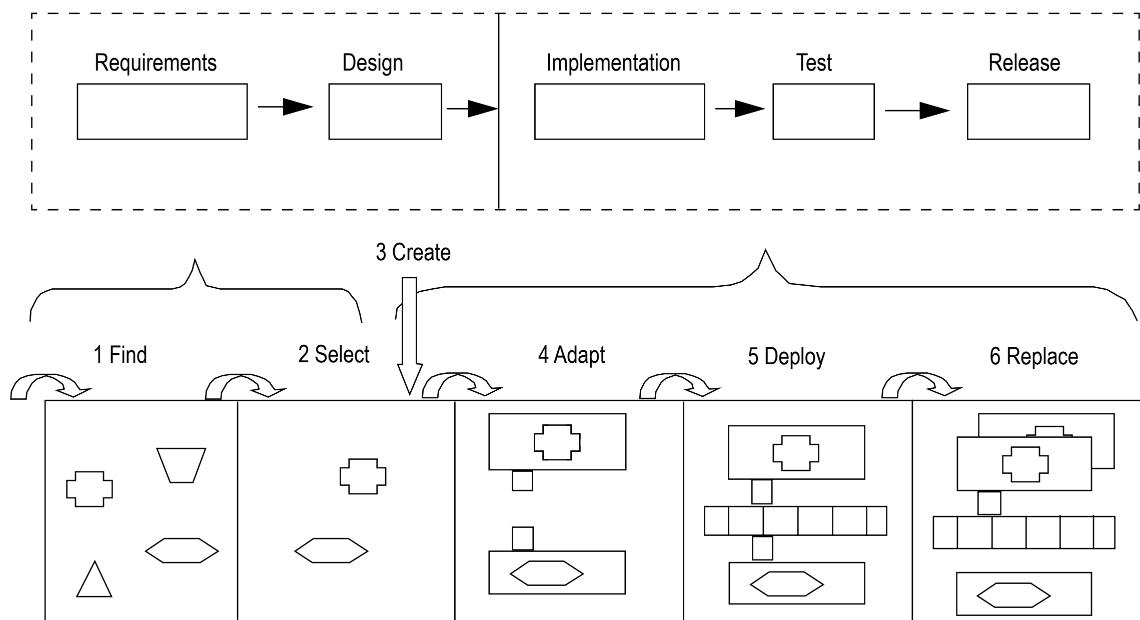


**Figure 1:** The CBSD Development Cycle Compared to the Waterfall Model (from[4])

 © Novática

be compatible with in-house software or other COTS products.

- Inflexibility: usually the source code of COTS software is not provided, so it cannot be modified.
- Complexity: COTS products can be too complex to learn and to use, imposing significant additional effort.
- Versioning: different versions of the same COTS product may not be compatible, causing more problems for developers.

Therefore, repeatable and systematic methods to assess and select COTS software are an important issue in COTS-based Software Engineering. The following discussion gives a brief characterisation of COTS selection and integration.

### 2.1 COTS Selection

Typically, COTS evaluations consist of two phases: COTS searching and screening (CS&S), and COTS analysis (CA).

1. CS&S. A COTS search is a set of activities that attempts to identify and find all potential candidate components for reuse. The search is generally driven by guidelines and criteria previously defined. Some methods propose a separate criteria definition process while others dynamically build a synergy of requirements, goals, and criteria [5]. For example, in the OTSO (Off-The-Shelf Option) method [6], the evaluation criteria are gradually defined as the selection process progresses, and are derived from reuse goals and factors that influence these goals. The evaluation criteria definition process essentially decomposes 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.
   COTS screening is the method used to decide which components should be selected for more detailed evaluation. Decisions are driven by a variety of factors – foremost are several design constraints that help define the range of components. So a balance is struck, depending upon the level of abstraction, complexity of the component, and goals and criteria.
2. CA. As the COTS alternatives have been evaluated, the evaluation data needs to be used for making a decision. A COTS analysis starts, in general, from a set of ranked COTS software alternatives where the top-ranked alternative is taken and exposed to measurement of a set of final make-or-buy decision criteria. For example analysis can be done by using an approach called *weighted scoring method*, in which criteria are defined and each criterion is assigned a weight or score.

In COTS selection, both phases – CS&S, and CA – are based on requirements statements that need to be much more flexible than traditional ones, i.e. the specified requirements should not be so strict as to either exclude the use of COTS or to require large product modification in order to satisfy them. Hence, some COTS evaluations include a process to acquire and validate customer requirements, while others build a selection process based on iteratively defining and validating requirements.

### 2.2 COTS Integration.

Component architectures divide software components into ‘requiring’ and ‘providing’: some software components can register the services they provide, while other components can subscribe to and consume these services. Components are plugged into a software architecture that connects participating components and enforces interaction rules. Thus, architectures mediate and regulate component interaction.

Interactions might potentially be characterised along many dimensions, depending on the architecture they are built into. Interactions, as part of a component model, require an accepted component vocabulary and a set of design standards. In general, the application composer should be able to search for patterns of interaction that reveal possible adaptation problems and allow calculation of the interaction effort. For example, the proposal in [7] presents a general classification of possible types of mismatches between COTS products and software systems, which includes architectural, functional, non-functional, and other issues. Three aspects of inter-component interactions and incompatibilities are considered in this approach: type of interacting component, layer (syntax or semantic-pragmatic), and number of components participating in the interaction. Different incompatibilities have different solutions, and the classes of problems are specific to the particular development phases.

### 2.3 Calculating Integration Effort

As important as determining architectural mismatching is calculating integration effort. Decisions on CBSD investments are strongly influenced by technological diversity: today technology is diverse and brings with it thousands of choices on components, with their opportunities and risks. Organisations 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 CBS. 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 [8], for example, combines several factors – architectural mismatching, complexity of an identified mismatch, and mismatch resolution – in order to estimate the effort required to integrate each potential component into an existing architecture. 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.

## 3 Some Open Issues

### 3.1 Measurement and Estimation

All CBS projects require a cost estimate before actual developments can proceed. Most cost estimates for CBS developments 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. Usually, the qualities of the desired COTS components are not directly measurable but are instead vague statements about like ‘acceptable performance’, ‘small size’, and ‘high reliability’.

© Novática

In spite of all that, researchers have started to define software metrics to guide quality and risk management in a CBS by identifying and quantifying various factors contributing to the overall quality [9]. Metrics let developers identify and quantify quality attributes in such a way that risks encountered during component selection and integration are reduced.

Cost is not independent but a function of the enterprise itself, its particular development, the solution it chooses, and the management and availability of resources during CBS projects. The cost of updating and/or replacing components is highly situational and depends on the specific internal and external factors affecting an organisation. Cost estimation should include the cost of project management, direct labour, and identification of affected software, affected data, and alternative solutions, testing and implementation. Some cost models are iterative, indicating change and re-evaluation throughout the solution stages. The most extended model on COTS integration cost – the COnstructive COTS (COCOTS) model [10] follows the general form of the Constructive Cost Model (COCOMO), 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); customisation and installation facilities; usability; portability; and previous product experience. Beside those factors, COCOMO II model drivers such as architecture/risk resolution, development flexibility, team cohesion, database size and required reusability are added to produce an integral COTS cost model. Therefore, it may provide the structure for easy integration of multiple COTS components. But a cost model cannot easily put aside each of the problems encountered when integrating multiple COTS. The cost model should provide mechanisms for dealing with the more common and more challenging problems. Some of them are as follows:

- In the migration to CBSD, we would like to exercise the policy "*Don't throw anything away, use and reuse as much as you can*". However, integrating existing software and COTS components as a whole is not a trivial task and may cause adaptation conflicts.
- Problems that occur when two or more components have the same services with different representations.
- Problems that occur when a component has to be modelled by integrating parts of functionality from different components (or from other sources).
- Problems that occur when integrating two or more components and the resulting architecture does not cover the desired application requirements.
- Problems that occur 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 a CBSD recur throughout a systems life cycle. The management phase consists mainly of maintaining, supporting, improving, and enhancing an organisation's product while using the components. One of the most important problems of the maintenance process is the estimation and prediction of related effort. The different maintenance activities may present several related aspects: reuse, understanding, deletion of parts, development of parts, re-documentation, etc. These aspects are relevant to all classes of systems, including object-oriented systems and CBS. However, one of the most important distinguishing factors in CBSD is the separation of the interface and the implementation and hence, component design decisions are driven by a variety of factors – including several design constraints that help define component's communications.

### 3.2 CBSD Risk Analysis

Although the Risk Management Paradigm continues to be useful as an overall guide to risk analysis, the majority of risk management studies deal with normative techniques of managing risk. Software risks may come from two different dimensions: (1) environmental contingencies such as organisational environment, technologies, and individual characteristics; and (2) risk management practices such as methods, resources, and period of use [11].

A few studies have classified software risk items, and for CBSD risk analysis is not the exception. These studies consider CBSD risks along several dimensions and have provided some empirically founded insights of typical cases and their variations. For example, the BASIS technique presented in [8] focuses on reducing risk in CBSD by means of several integrated processes. One of the processes reduces the risk of selecting inappropriate COTS components; another process reduces the risk of downstream integration problems through early assessment of the compatibility between a chosen COTS product and an existing system, and finally a third process reduces risk throughout the development lifecycle by defining built-in checkpoints and recommending measurement techniques. Also a vendor analysis is carried out along with the processes.

### 3.3 Standard Documentation

Documentation is also a key to the success of the CBSD. High quality documentation ensures that design and implementation standards are reflected in all of the application content built with the component. For example, a conceptual language would help documentation by making explicit the links between component properties and component interaction. In general, the composer need not understand all aspects of a CBS, but should be able to search for components that reveal functionality for solving a specific problem using the component. Component documentation might be extensible allowing the composer to insert details about component adaptations directly onto a system document library or annotating components. In addition, the system documentation should be closed tied to the composing tools, so the selected and tailored components are easily translated into reusable component libraries.

Documentation should contain the information necessary to understand all the functions stated in the specification description, and the user's documentation should completely describe all user-callable functions in the component. During the evaluation it is necessary to gather information about functional features, and other characteristics of the component; for example,

 © Novática

description of the system environment, previous product versions, customer's support strategies, etc. Hence, documentation's quality is crucial for a correct understanding and evaluation of COTS, which sometimes demand a translation of several types of vocabulary. Even some approaches suggest the use of particular notations such as use cases or scenarios, including essential customer requirements and customer standards, or the extension of UML (Unified Modelling Language) diagrams with ADLs (Architecture Description Languages), few of them particularly address documentation quality as a main element to understand COTS components.

### 3.4 Testing CBSD

Black box testing techniques are characterised by focusing tests on the expected behaviour of a software component and ensuring that the resulting functionality is correct. In many cases this behavioural testing is done without the aid of source code or design documents, and it is possible to provide input test cases and only test the outputs of the object for correctness. This type of black box testing does not require knowledge of the source code and so is possible to do with COTS components. Strategies for black box testing that may be effective for COTS components include test case 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. With COTS components, the user should know these facts and be able to create test cases around them. To guide test case selection, the operational profile of the COTS component can be used. The operational profile identifies the criticality of components and their duration and frequency of use. Both the operational profile of the COTS component and the whole system must be taken into account.

CBSD also introduces some additional challenges to testing: components must fit into the new environment when they are reused, often requiring real-time detection, diagnosis, and handling of software faults. The Built-in Tests (BIT)-based COTS technology developed within the European project Component+ [12] makes it possible to reuse tests by applying BIT to software components. A BIT-based COTS is a software component whose source code includes tests as special functions. This approach enables tests be embedded and reused in the same way as that of code in COTS and bespoke components is used, allowing producing self-testable components so that dynamic faults may be detected at run-time.

In summary, the problem of testing component-based software is complicated by a number of characteristics of component-based Software Engineering. Distributed component-based systems of course exhibit all the well-known problems that make traditional testing difficult. But testing of component-based software is further complicated by the fact that different functionality is provided inside a component depending on its interfaces and state.

## 4 Conclusions

A component-based development alone does not guarantee the success of a system, which depends on how methodologies, processes and techniques of a CBSE are defined and applied. However, there are many open issues that should be solved to provide developers with a full range of methods and processes [5]. Some of the issues have been briefly discussed in this article – measurement and estimation, risk analysis, documentation, and testing. Other issues are listed here:

- Component certification: Certification is the procedure by which a third-party gives written assurance that a product, process, or service conforms to specified requirements. Current certification practices are highly process oriented, and usually require early life-cycle processes stressing the resulting software product. In this context, the assembly of an application based on third-party developed components does not necessarily assure defect-free software.
- Component configuration: One of the basic problems when developing component-based systems is that it is difficult to keep track of components and their interrelationships. This problem is more severe during design and deployment phases due to run-time dependencies, although requirement specification and all other phases are also affected. Identifying COTS components and their versions is now out of developer's direct control.
- Composition predictability: Predictability implies developing a technology that supports composing systems from precompiled components in such a way that the quality of the system can be predicted before acquiring/building its constituents. CBSE should provide support for predicting behavioural properties of assemblies before the components are actually developed or purchased.

Of course, a successful component-based solution might be produced due to the experience and skill of developers and composers. But what we actually need is successful component-base endeavours supported by CBSE, in which best practices can be identified. The crucial point here is how the open issues can be met. This is still an ongoing concern.

**References**

[1] C. Szyperski. Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[2] K. Wallnau, S. Hissam, and R. Seacord. Building Systems from Commercial Components. Addison-Wesley, 2002.

[3] G. Heineman and W. Council. Component-Based Software Engineering – Putting the Pieces Together. Addison-Wesley, 2001.

[4] Crnkovic I., Larsson M. Building Component-based Reliable Software Systems, Artech House, 2002

[5] A. Cechich, M. Piattini, and A. Vallecillo. Assessing Component-Based Systems. In Component-Based Software Quality: Methods and Techniques. A. Cechich et al. (eds.), Springer Verlag, LNCS 2693, 2003.

[6] J. Kontio. OTSO: A Systematic Process for Reusable Software Component Selection. Technical Report UMIACS-TR-95-63, University of Maryland, 1995.

© Novática

[7]
D. Yakimovich, J. Bieman, and V. Basili. Software architecture classification for estimating the cost of COTS integration. In Proceedings of ICSE'99, pages 296–302, 1999.

[8]
K. Ballurio, B. Scalzo, and L. Rose. Risk Reduction in COTS Software Selection with BASIS. In Proceedings of the First International Conference on COTS-Based Software Systems, Springer-Verlag, pages 31–43, 2002.

[9]
J. Martín-Albo, M. Bertoa, C. Calero, A. Vallecillo, A. Cechich, and M. Piattini, GQM: A Software Component Metric Classification Model 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, QAOOSE 2003, Darmstadt, Germany, July 2003.

[10]
COCOTS. COnstructive COTS Model.
<http://sunset.usc.edu/research/COCOTS/>, 2001.

[11]
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.

[12]
EC. IST-1999-20162, Component+.
<http://www.component-plus.org>, 2002.

 © Novática