Component-based Software Engineering — New Challenges in Software Development

The primary role of component-based software engineering is to address the development of systems as an assembly of parts (components), the development of parts as reusable entities, and the maintenance and upgrading of systems by customising and replacing such parts. This requires established methodologies and tool support covering the entire component and system lifecycle including technological, organisational, marketing, legal, and other aspects. The traditional disciplines from software engineering need new methodologies to support component-based development. **IVICA CRNKOVIC** assesses the challenges of this emerging technology and discusses its implications for the software development process.

Ivica Crnkovic
Mälardalen University
Department of Computer Engineering
Västerås, Sweden
ivica.crnkovic@mdh.se
http://www.idt.mdh.se/~icc

Software Development Challenges

We are witnessing an enormous expansion in the use of software in business, industry, administration and research. Software is no longer marginal in technical systems but has now become a central factor in many fields. System features based on software functionality, rather than other characteristics, are becoming the most important factor in competing on the market, for example in the car industry, the service sector and in schools. Increasing numbers of software users are non-experts. These trends place new demands on software. Usability, robustness, simple installation and integration become the most important features of software. As a consequence of the wider area of software utilisation, the demand for the integration of different areas has increased. We distinguish between *vertical integration* in which data and processes at different levels are integrated, and *horizontal integration* in which similar types of data and processes from different domains are integrated. For example, in industrial process automation, at the lowest levels of management (*Field Management*), data collected from the process and controlled directly, is integrated on the plant level (*Process Management*), then is further processed for analysis and combination with data provided from the market and finally published on the Web (*Business Management*).

A consequence of all this is that software is becoming increasingly large and complex. Traditionally, software development addressed challenges of increasing complexity and dependence on external software by focussing on one system at a time and on delivery deadlines and budgets, while ignoring the evolutionary needs of the system. This has led to a number of problems: the failure of the majority of projects to meet their deadline, budget, and quality requirements and the continued increase in the costs associated with software maintenance. To meet these challenges, software development must be able to cope with complexity and to adapt

Non-experts...place new demands on software.

quickly to changes. If new software products are each time to be developed from scratch, these goals cannot be achieved. The key to the solution to this problem is reusability. From this perspective *Component-based Development* (CBD) appears to be the right approach. In CBD software systems are built by assembling components already developed and prepared for integration. CBD has many advantages. These include more effective management of complexity, reduced time to market, increased productivity, improved quality, a greater degree of consistency, and a wider range of usability [1].

However, there are several disadvantages and risks in using CBD which can jeopardise its success.

- ***Time and effort required for development of components.*** **Among the factors which can discourage the development of reusable components is the increased time and effort required, the building of a reusable unit requires three to five times the effort required to develop a unit for one specific purpose. (B. Spencer, Microsoft, Presentation at 22nd ICSE, 1999, also an interesting observation about efficient reuse of real-time components, made by engineers at Siemens [2] that, as a rule of thumb, the overhead cost of developing a reusable component, including design plus documentation, is recovered after the *fifth* reuse. Similar experience at ABB [3] shows that reusable components are exposed to changes more often than non-reusable parts of software at the beginning of their lives, until they reach a stable state.)**
- ***Unclear and ambiguous requirements.*** **In general, requirements management is an important part of the development process, its main objective being to define consistent and complete component requirements. Reusable components are, by definition, to be used in different applications, some of which may yet be unknown and the requirements of which cannot be predicted. This applies to both functional and non-functional requirements.**
- ***Conflict between usability and reusability.*** **To be widely reusable, a component must be sufficiently general, scalable and adaptable and therefore more complex (and thus more complicated to use), and more demanding of computing resources (and thus more expensive to use). A requirement for reusability may lead to another development approach, for example building a new, more abstract, level which gives less flexibility and fine tuning, but achieves better simplicity [3,4].**
- ***Component maintenance costs.*** **While application maintenance costs can decrease, component maintenance costs can be very high since the component must respond to the different requirements of different applications running in different environments, with different reliability requirements and perhaps requiring a different level of maintenance support.**

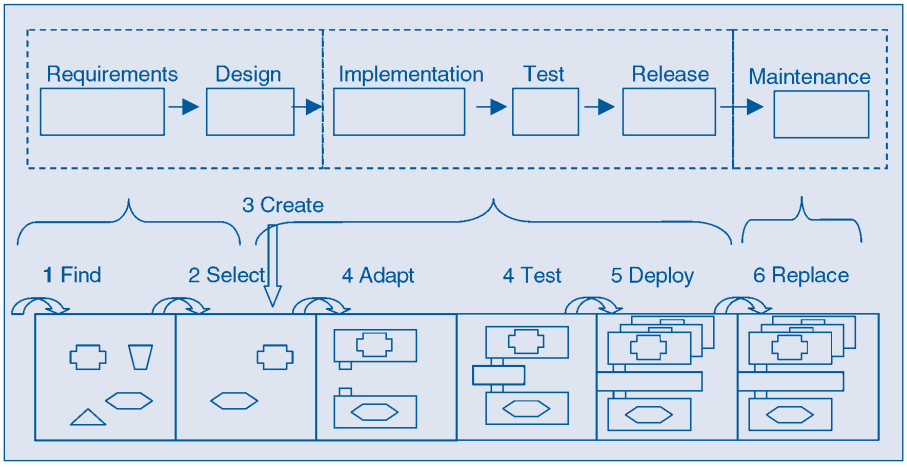

Figure 1 The development cycle compared with the waterfall model

- ***Reliability and sensitivity to changes.*** **As components and applications have separate lifecycles and different kinds of requirements, there is some risk that a component will not completely satisfy the application requirements or that it may include concealed characteristics not known to application developers. When introducing changes on the application level (changes such as updating of operating system, updating of other components, changes in the application, etc.), there is a risk that the change introduced will cause system failure. To enjoy the advantages and avoid the problems and risks, we need a systematic approach to component-based development at the process and technology levels.**

Component-Based Software Engineering

The concept of building software from components is not new. A 'classical' design of complex software systems always begins with the identification of system parts designated subsystems or blocks, and on a lower level modules, classes, procedures and so on. The reuse approach to software development has been used for many years. However, the recent emergence of new technologies has significantly increased the possibilities of building systems and applications from reusable components. Both customers and suppliers have had great expectations from CBD, but their expectations have not always been satisfied. Experience has shown that component-based development requires a systematic approach to and focus on the component aspects of software development [3]. Traditional software engineering disciplines must be adjusted to the new approach, and new procedures must be developed. *Component-based Software Engineering* (CBSE) has become recognised as such a new sub-discipline of Software Engineering.

The major goals of CBSE are the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customising and replacing their components [5]. The building of systems from components and the building of components for different systems requires established methodologies and processes not only in relation to the development/maintenance aspects, but also to the entire component and system lifecycle including organisational, marketing, legal, and other aspects. In addition to specific CBSE objectives such as component specification or composition and technologies, there are a number of software engineering disciplines and processes that require specific methodologies for application in component-based development. Many of these methodologies are not yet established in practice, some are not even developed. The progress of software development in the near future will depend very much on the successful

Software engineering disciplines must be adjusted to the new approach, and new procedures must be developed.

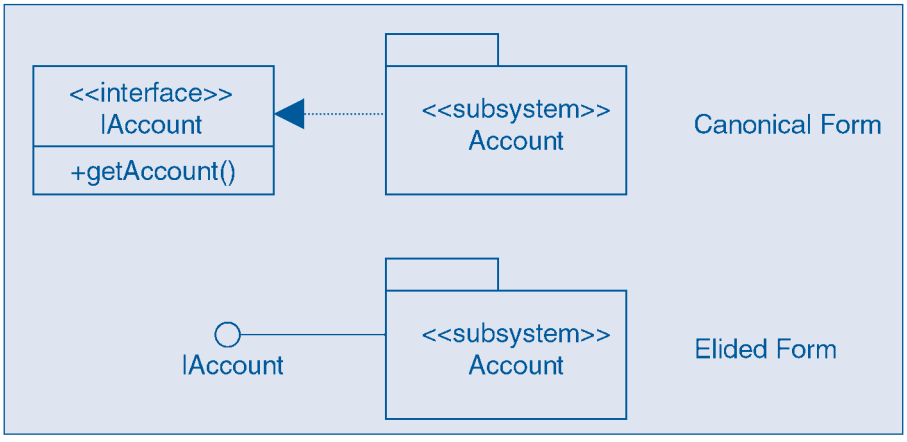

Figure 2 UML component

establishment of CBSE and this is recognized by both industry and academia. All major software engineering conferences now include sessions related to CBSE, and CBSE workshops are held frequently [6–10]. According to the Gartner Group [11] 'By 2002, 70% of all new applications will be deployed using component-based application building blocks.'

Overviews of certain CBSE disciplines and some of the relevant trends and challenges in the near future are presented below.

Component Specification

For a common understanding of component-based development, the starting point is an agreement of what a component is and what it is not. As a generic term the concept is pretty clear — a component is a part of something — but this is too vague to be useful. The definition of a component has been widely discussed [13,14]. However, we shall adopt Szyperski's definition [4], which is the most frequently used today:

> *A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parts.*

The most important feature of a component is the separation of its interface from its implementation. This separation is different from those that we can find in many programming languages (such as ADA or Modula-2), in which declaration is separated from implementation, or those in object-oriented programming languages in which class definitions are separated from class implementations. We require that the integration of a component into an application should be independent of the component development lifecycle and that there should be no need to recompile or re-link the application when updating with a new component. Another important characteristic of the separation is that the component implementation is only visible through its interface. This is especially significant for components delivered by a third party. An implication of this is the requirement for a complete specification of a component including its functional interface, non-functional characteristics (performance, resources required, etc.), use cases, tests, etc. While current component-based technologies successfully manage functional interfaces, there is no satisfactory support for managing other parts of a component specification.

The component definition adopted above is focused on the use of components. It says little about how to design, implement and specify a component. There are however, other definitions that point to other aspects of component-based development. For example there is a strong relation between object-oriented programming (OOP) and components. Component models (also called component standards) COM/DCOM [15,16], .NET [17], Enterprise Java Beans (EJB) [15,17], and CORBA Component Model (CCM) [20] relate Component Interface to Class Interface. Components adopt object principles of unification of functions and data encapsulation. Cheesman and Daniels [21] consider that a component can exist in several forms during its lifecycle: *Component Specification* (component characteristics, component function), *Component Interface* (a part of its specification, a definition of a component's behaviour), *Component Implementation* (a realisation of a Component Specification), *Installed Component* (deployed instance of a Component Implementation) and *Component Object* (an instance of Installed Object). Not all researchers agree that components are extensions of OOP. On the contrary, they consider that the difference between components and objects lies in the fact that an object has state and is a unit of instantiation, while a component is stateless and is a unit of deployment [4].

There are also different understandings of CBD in academia and industry [22]. While researchers in academia define components as well-defined entities (often small, and with easily understood functional and non-functional features), industry sees components as parts of a system which can be reused, but are not necessarily well defined with explicit interfaces and with slight or no conformance with component models. A component can be an amorphous part of a system, the adaptation of which may require much effort. Such components (or rather reusable entities) are of extreme importance, as the larger the components are, the greater the productivity that can be achieved by their reuse.

Component specification remains a topic of research. Component standards are mostly concentrated on the interface definition, while non-functional properties are specified (if specified at all) informally in separate documentation. Some improvements in that direction, by gathering both functional characteristics and design characteristics, have been made in the new Microsoft Component Model .NET.

Component-Based System Development Lifecycle

CBSE addresses the requirements, challenges and problems similar to others encountered elsewhere in software engineering. Many of the methods, tools and principles of software engineering can be used in the same or in a similar way as in other types of applications or systems but there is one distinction, CBSE covers both component development and system development with components. There is a slight difference in the requirements and

business ideas in the two cases and different approaches are necessary. Of course, when developing components, other components can be (and often must be) incorporated but the main emphasis is on reusability: Components are built to be used and reused in many applications, some not yet existing. A component must be well specified, easy to understand, sufficiently general, easy to adapt, easy to deliver and deploy and easy to replace. The component interface must be as simple as possible and strictly separated (both physically and logically) from its implementation. Marketing factors play an important role as development costs must be recovered from future earnings, this being especially true for COTS. However, the main problem in developing components is in the acquisition and elicitation of requirements in combination with COTS selection [23] because the process includes multi-criteria decisions. If the process begins with requirements selection, it is highly probable that a COTS meeting all the requirements will not be found. If components are selected too early in the process, the system obtained may not meet all the requirements.

Development with components is focused on the identification of reusable entities and relations between them, starting from the system requirements. The early design process includes two essential steps: Firstly, specification of a system architecture in terms of functional components and their interaction, this giving a logical view of the systems and secondly, specification of a system architecture consisting of physical components.

Different lifecycle models, established in software engineering, can be used in CBD. These models will be modified to emphasise component-centric activities. Let us consider, for example, the waterfall model using an extreme component-based approach. Figure 1 shows the waterfall model and the meaning of the phases. Identifying requirements and a design in the waterfall process is combined with finding and selecting components. The design includes the system architecture design and component identification/selection.

The different steps in the component-based systems development process are:

- ***Find components that may be used in the system*. All possible components are listed here for further investigation. To successfully perform this procedure, a vast number of possible candidates must be available as well as tools for finding them. This is an issue not only of technology, but also of business.**
- ***Select the components that meet the requirements of the system*. Often the requirements cannot be fulfilled completely, and a trade-off analysis is needed to adjust the system architecture and to reformulate the requirements to make it possible to use the existing components.**
- ***Alternatively, create a proprietary component to be used in the system.* In a component-based development process this procedure is less attractive as it requires more efforts and lead-time. However, the components that include core-functionality of the product are likely to be developed internally as they should provide the competitive advantage of the product.**
- ***Adapt the selected components so that they suit the existing component model or requirement specification*. Some components would be possible to be directly integrated in to the system, some would be modified through a parameterisation process, some would need wrapping code for adaptation, etc.**
- ***Compose and deploy the components using a framework for components*. Typically component models would provide that functionality.**
- ***Replace earlier with later versions of components.* This corresponds with system maintenance. Bugs may have been eliminated or new functionality added.**

There are many other aspects of CBD that require specific methods, technologies and management. For example, development environment tools [24,25], component models and support for their use, software configuration management [26], testing, software metrics, legal issues, project management, development process, standardisation and certification issues, etc. Discussion of these is beyond the scope of this article and the relation between software architecture and CBD is discussed in the following.

Software Architecture and Component-Based Development

Software architecture and components are closely related. All software systems have an architecture that can be viewed in terms of the decomposition of the system into components and their relations. A commonly used definition of Software architecture is 0: '*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.*' Traditionally, software architecture is at the focus in the early design phase when the overall structure of the system is designed to satisfy functional and non-functional requirements. In monolith applications, the architecture specified in the design process is concealed at execution time in one block of executable code. Component technologies focus on composition and deployment, closer to or at execution time. In a component-based system, the architecture remains recognisable during the application or system execution, the system still consisting of clearly separated components. The system architecture thus remains an important factor during the execution phase. Component-based Software Engineering embraces the total lifecycles of components and component-based systems and all the procedures involved in such lifecycles.

In a 'classical' approach, the design of software begins with determining its architecture, structuring the system in smaller parts, as independent as possible. The first phase of this structuring is functionality-based architectural design. The second phase is software architecture assessment during which the software architecture is evaluated with respect to the driving quality requirements. Once the software architecture has been defined, the components that are to constitute the system must be developed or selected. We can distinguish different categories of components in relation to the requirements of the system: special purpose components, developed specifically for the system, reused components, internally developed for multiple usage, and final commercial components (COTS). Pre-existing components typically need to be integrated into the system using glue code or a modification of the components themselves. This top-down approach ensures the fulfilment of the requirements, or at least a better control of requirements fulfilment. However, this approach does not encourage the reuse of pre-existing components, especially not commercial components, since there is a high degree

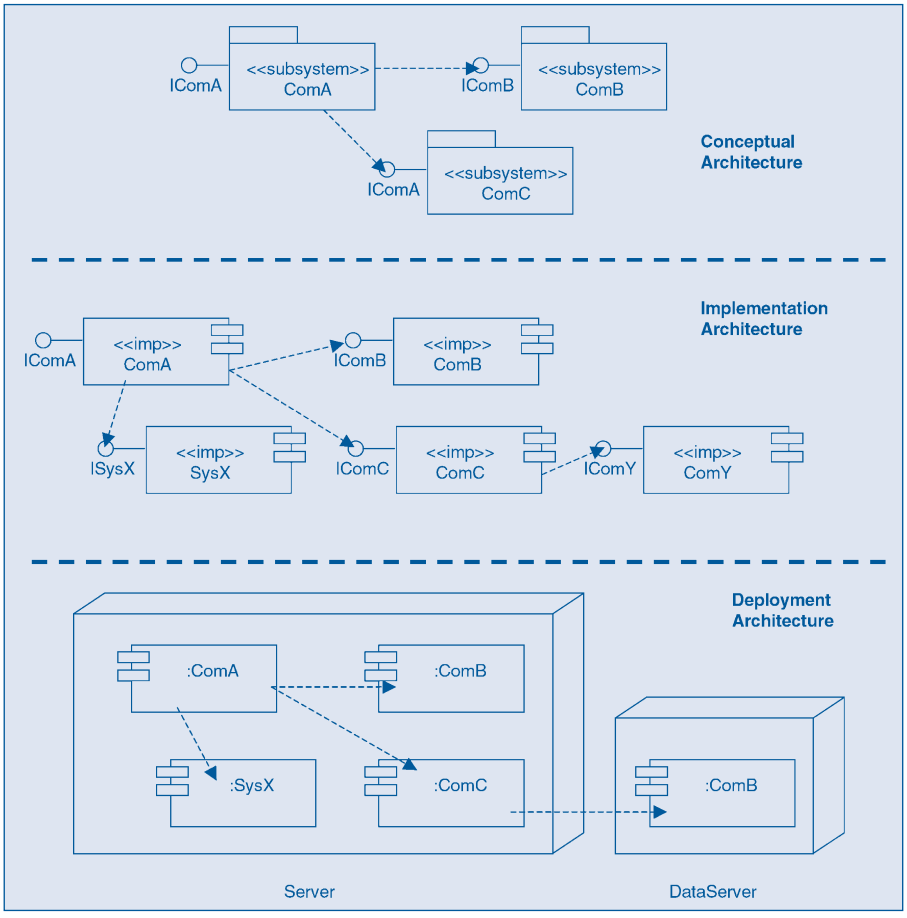

Figure 3 Examples of different aspects of component-based architecture

of probability that the pre-existing components do not exactly fit into the system. Another approach, a mix of bottom-up and top-down approaches begins with the system requirements and the analysis of possible candidate components. The component specification and selection have an impact on the final requirements and the system architecture. In this case, software architecting is primarily concerned with identifying means of optimising the interactions between the given components. Since basic artefacts for both software architecture and component technologies are components and their composition is natural that they will merge, i.e. use common techniques, methods and tools. Architectural definition languages (ADLs), for example ACME [29], can be used for designing component-based-systems and implemented for the existing component models.

Software architecture is often related to a process of tradeoff analysis. Experience has shown that the many attributes of large software systems live principally in the system's software architecture. In such systems the achievement of qualities attributes depends more on the overall software architecture than on code-level practices such as language choice, detailed design, algorithms, data structures, testing, and so on. There exist several methods for such analysis, for example SAAM (Software Architecture Analysis Method) [30] and ATAM (Architecture Tradeoff Analysis Method) [31]. Both ATAM and SAAM are a scenario-based method. However, unlike the SAAM, the ATAM focuses on multiple quality attributes (modifiability, availability, security, and performance) and is aimed at locating and analysing trade-offs in a software architecture. For component-based systems a modified approach in these analyses is required. The components have pre-determined attributes, some of them immanent only to the component, but some of them emerging in composition with other components. A trade-off analysis helps in selecting the proper components and in predicting the attributes of component compositions. At the same time inclusion of the pre-existing components sets the boundaries in which the analysis can be performed. For example one characteristic of a candidate component can be a high reusability but a poor performance, while of the other candidate a better performance but a lower reusability. The architectural analysis will help in making a decision in component selection.

Software architecture and CBD are successfully used in the development of software product lines [22,27] from which many variants of a product are delivered. Typical product variants contain a set of core-components and a number of additional components. The component-based approach and architectural design play an important role in product configuration management.

UML and Component-Based Systems Modelling

UML (Unified Modelling Language) can be used for both component and system modelling, as shown in [21]. Component-driven design concentrates on interface definitions and collaboration between the components through the interfaces. The design process continues with the modelling of the system with physical components, which do not necessarily match the logical structure. These may be pre-existing components, with interface already specified and possibly in need of wrappers. One logical component, identified in the first phase of design, may consist of several physical components. Finally, there is a deployment aspect, the components being executed on different computers in a distributed application. In a non-component-base approach the first, the design phase is important, while mapping between the conceptual and implementation level is a direct mapping, and the deployment phase is the same for the whole application. In principle, UML [32] can be utilised to provide support for designing component-based systems covering all these aspects [1,5]. Interfaces are presented as multiple subsystems (also multiple interfaces may be realised by a subsystem), which indicate the possibility of changing the implementation without replacing the interface. An interface can be presented in two ways (see Figure 2), the second alternative being the more common presentation.

Figure 3 shows the three aspects of system architecture. The conceptual architecture is a result of a top-down system analysis and design and in at least the first step is not different from a 'non-component-based' design. In the conceptual part the components are expressed by UML packages with the <<subsystems>> stereotype. In the implementation architecture part, the physical components are represented by UML components and the <<imp>> stereotype. Note that the implementation part is not necessarily the only refinement of the conceptual level, but also the structure can be changed. For example, different packages can include the same physical components. It may also happen that the component selection requires modifications of the conceptual architecture.

UML is however not specialised for CBD and certain extensions to standard UML (such as naming convention, or stereotypes) are required. The component interfaces cannot be described by UML at such a detailed level that they can be used directly. For this reason there exist extensions to UML, for example Catalysis [33]. Further work on UML related to CBSE is expected. The next major version of UML (UML 2.0) [34] includes proposals for extensions for describing Enterprise Java Beans, data modelling entities, real-time components, XML components, etc. Many of these are related directly or indirectly to CBSE.

Future of Component-Based Software Engineering

It is obvious that CBD and CBSE are in the very first phase of their lives. CBD is recognised as a new, powerful approach that will, if not revolutionise, at least significantly change the development of software and software use in general. We can expect that components and component-based services will be widely used by non-programmers for building their applications. Tools for building such applications by component assembly will be developed. Automatic component update over the Internet, already present today in many applications, will be a standard means of application improvement. Another trend we can see is the standardisation of domain-specific components on the interface level. This will make it possible to build applications and systems from components purchased from

Components and component-based services will be widely used by non-programmers for building their applications.

different vendors. The standardisation of domain-specific components requires the standardisation of domain-specific processes. Widespread work on standardisation in different domains is already in progress, (a typical example is OPC Foundation [35], working on a standard interface to make possible interoperability between automation/control applications, field systems/devices and business/office applications). Support for the exchange of information between components, applications, and systems distributed over the Internet will be further developed. Works related to XML [36] will be further expanded.

CBSE is facing many challenges today, some of these are summarised in the following.

- *Trusted components* — Because the trend is to deliver components in binary form and the component development process is outside the control of component users, questions related to component trustworthiness become of great importance. The meaning of 'trustworthiness' is, however, not precisely defined. Although there are formal definitions of many attributes associated with the concept 'trustworthiness' (reliability and robustness, for example), there is no formal definition and understanding of 'trustworthy', no standardised measurement or trustworthiness. What are the effects of different degrees of trustworthiness on system attributes is not known.

- *Component certification* — One way of classifying components is to certificate them. In spite of the common belief that certification means absolute trustworthiness, it in fact only gives the results of tests performed and a description of the environment in which the tests were performed. While certification is a standard procedure in many domains, it is not yet established in software in general and especially not for software components [37,38].

- *Composition predictability* — Even if we assume that we can specify all the relevant attributes of components, it is not known how these attributes determine the corresponding attributes of systems of which they are composed. The ideal approach to derive system attributes from component attributes is still a subject of research. A question remains — 'Is such derivation at all possible? Or should we not concentrate on the measurement of the attributes of component composites?' [39].

- *Requirements management and component selection* — Requirements management is a complex process. A problem of requirements management is that requirements in general are incomplete, imprecise and contradictory. In an in-house development, the main objective is to implement a system that will satisfy the requirements as far as possible within a specified framework of different constraints. In component-based development, the fundamental approach is the reuse of existing components. The process of engineering requirements is much more complex as the possible candidate components usually lack one or more features which meet the system requirements exactly. In addition, even if some components are individually well suited to the system, it is not necessary that they do not function optimally in combination with others in the system — or perhaps not at all. These constraints may require another approach in requirements engineering — an analysis of the feasibility of requirements in relation to the components available and the consequent modification of requirements. As there are many uncertainties in the process of component selection there is a need for a strategy for managing risks in the components selection and evolution process [5,40].

- *Long-term management of component-based systems* — As component-based systems include sub-systems and components with independent lifecycles, the problem of system evolution becomes significantly more complex. There are many questions of different types: technical issues (can a system be updated technically by replacing components?), administrative and organisational issues (which components can be updated, which components should be or must be updated?), legal issues (who is responsible for a system failure, the producer of the system or of the component?), etc. CBSE is a new

approach and there is little experience as yet of the maintainability of such systems. There is a risk that many such systems will be troublesome to maintain.

- ***Development models*** **— Although existing development models demonstrate powerful technologies, they have many ambiguous characteristics, they are incomplete, and they are difficult to use.**
- ***Component configurations*** **— Complex systems may include many components which, in turn, include other components. In many cases compositions of components will be treated as components. As soon as we begin to work with complex structures, the problems involved with structure configuration popup. For example, two compositions may include the same component. Will these components be treated as two different entities or will they be assumed to be one identical entity? What happens if these components are of different versions, which version will be selected? What happens if these versions are not compatible? The problems of the dynamic updating of components are already known, but their solutions are still the subject of research [41].**
- ***Dependable systems and CBSE*** **— The use of CBD in safety-critical domains, real-time systems, and different process-control systems, in which the reliability requirements are more rigorous, is particularly challenging. A major problem with CBD is the limited possibility of ensuring the quality and other non-functional attributes of the components and thus our inability to guarantee specific system attributes.**
- ***Tool support*** **— The purpose of Software Engineering is to provide practical solutions to practical problems, and the existence of appropriate tools is essential for a successful CBSE performance. Development tools, such as Visual Basic, have proved to be extremely successful, but many other tools are yet to appear — component selection and evaluation tools, component repositories and tools for managing the repositories, component test tools, component-based design tools, run-time system analysis tools, component configuration tools, etc. The objective of CBSE is to build systems from components simply and efficiently, and this can only be achieved with extensive tool support.**

These are some of the many challenges facing CBSE today. The goal of CBSE is to standardise and formalise all disciplines supporting activities related to CBD. The success of the CBD approach depends directly on further research and the implementation of CBSE.

References

[1] Brown A. *Large-Scale Component-Based Development*. Prentice Hall, 2000.

[2] Mrva M. *Reuse Factors in Embedded Systems Design*. High-Level Design Techniques Dept. at Siemens AG, Munich, Germany, 1997.

[3] Crnkovic I, Larsson M. A Case Study: Demands on Component-based Development. *Proceedings 22nd International Conference on Software Engineering*, ACM Press, 2000.

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

[5] Heineman G, Councill W. *Component-based Software Engineering, Putting the Pieces Together*. Addison Wesley, 2001.

[6] ICSE 2000, Workshop on Component-Based Software Engineering (CBSE 3), http://www.sei.cmu.edu/cbs/cbse2000/CFP2000.html, Access date 2001-07-14.

[7] ICSE 2001, Workshop on Component-Based Software Engineering (CBSE 4), http://www.sei.cmu.edu/pacc/CBSE4-Proceedings.html, Access date 2001-07-14.

[8] ECOOP 2000, Workshop on Component-Oriented Programming, http://www.ipd.hk-r.se/bosch/WCOP2000/, Access date 2001-07-14.

[9] Euromicro 2001, Workshop on Component-Based Software Engineering, http://www.idt.mdh.se/ecbse/, Access date 2001-07-14.

[10] Workshop on CBSE – ABB Corporate Research Centre, Switzerland, 2000, http://icawww2.epfl.ch/~opreiss/CBSE_Conference2000/, Access date 2001-07-14.

[11] Gartner Group, http://www.gartner.com, Access date 2001-07-14.

[12] Brown A, Wallnau K. The current state of CBSE, *IEEE Software*, 1998.

[13] Szyiperski C, Pfister C. Workshop on Component-Oriented Programming, Summary. In Mühlhäuser M. (ed.) *Special Issues in Object-Oriented Programming — ECOOP96 Workshop Reader*, Springer, 1997.

[14] ICSE 1999, Workshop on Component-Based Software Engineering (CBSE 2), http://www.sei.cmu.edu/cbs/icse99/cbsewkshp.html, Access date 2001-07-14.

[15] Box D. *Essential COM.* Addison-Wesley, 1998.

[16] Microsoft Component Object Model, http://www.microsoft.com/com/, Access date 2001-07-14.

[17] Microsoft .NET Component Model, http://www.microsoft.com/net, Access date 2001-07-14.

[18] Enterprise Javabeans technology, http://java.sun.com/products/ejb/, Access date 2001-07-14.

[19] Matena V, Stearns B. *Applying Enterprise JavaBeans(TM): Component-Based Development for the J2EE(TM) Platform.* Addison-Wesley, 2000.

[20] OMG, CORBA, http://www.omg.org/technology/documents/spec_catalog.htm, Access date 2001-07-14.

[21] Cheesman J, Daniels J. UML *Components — a Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[22] Bosch J. *Design & Use of Software Architecture.* Addison Wesley, 2000.

[23] Maiden N, Ncube C. Acquiring Requirements for Commercial Off-The Shelf Package Selection, *IEEE Software* 1998; **15**(2).

[24] Microsoft Visual Studio, http://msdn.microsoft.com/vstudio/, Access date 2001-07-14.

[25] Development tools — Forte ™ tools, http://www.sun.com/forte/, Access date 2001-07-14.

[26] Larsson M, Crnkovic I. New challenges for configuration Management, *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, Springer, 1999.

[27] Bosch J. Software Product Lines: Organisational alternatives, *ICSE 2000 Proceedings*, ACM Press, 2001, 91-100.

[28] Bass L, Clements P, Kazman R, *Software Architecture In Practice.* Addison Wesley, 1998.

[29] ACME architecture definition language, http://www.cs.cmu.edu/~acme/, Access dated 2001-07-14.

[30] Kazman R, Abowd G, Bass L, Clements P. Scenario-based analysis of software architecture. *IEEE Software* 1996: 47–55.

[31] Kazman R, Barbacci M, Klein M, Carrière SJ. Experience with Performing Architecture Tradeoff Analysis, *ICSE 1999 Proceedings*, ACM Press, 1999, 54–63.

[32] Booch G, Jacobson I, Rumbaugh J. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[33] D'Souza D, Wills A. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[34] OMG UML, http://www.omg.org/technology/uml, Access date 2001-07-14.

[35] OPC Foundation, http://www.opcfoundation.org/ , Access date 2001-07-14.

[36] Extensible Markup Language (XML) http://www.w3.org/XML, Access date 2001-07-14.

[37] Voas J, Payne J. Dependability certification of software components. *Journal of Systems and Software* 2000; **52**: 165–172.

[38] Morris J, Lee G, Parker K, Bundell G, Peng Lam C. Software component certification. *IEEE Computer* 2001.

[39] Wallnau K, Stafford J. Ensembles: Abstractions for a New Class of Design Problem, *27th Euromicro Conference 2001 Proceedings*, IEEE Computer Society, 2001: 48–55

[40] Kotonya G, Rashid A. A strategy for Managing Risks in Component-based Software Development, *27th Euromicro Conference 2001 Proceedings*, IEEE Computer Society, 2001: 12–21.

[41] Crnkovic I, Larsson M, Küster Filipe JK, Lau K. *Databases and Information Systems, Fourth International Baltic Workshop, Baltic DB&IS*, Selected papers. Kluwer Academic Publishers, 2001: 237–252.