

Workshop on Component-Based Software Engineering Processes

Kingsley C. Nwosu
Lucent Technologies, Inc.
600-700 Mountain Ave
Room 3D-435
Murray Hill, NJ 07974
+1 908-582-7131

Robert C. Seacord
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+1 412-268-3265

Abstract

Component-based software engineering (CBSE) spans a range of technologies and engineering practices. Engineering practices for component-based systems (e.g., design, integrate, test, deploy and sustain) are emerging, but in isolated settings rather than at a community level. The goal of this workshop is to provide a baseline understanding of the broad aspects of CBSE processes.

1 Introduction

Over the years, it has been the ambition and goal of the Software Engineering community to have the ability to quickly assemble or build a software system from compatible and cooperating entities or components. At last, the long cherished ambition of Software System Developers and Integrators is coming to fruition. This has been largely motivated by similar abilities in the companion hardware areas and also by the other foreseeable by products such as shortened time to market and increased productivity.

The Component-Based Software Engineering (CBSE) process departs drastically from the conventional software development process in that it's integration-centric as opposed to development-centric. And a true CBSE depends mostly on selection, acquisition, and integration of components from external vendors, which raises a lot of issues and problems. There are numerous issues, questions, and risks involved in component selection, acquisition, and integration that must be adequately addressed and resolved in a CBSE project. And all these, and other necessary tasks, such as requirements engineering, domain engineering, etc., must be addressed in the context of a formal Component-Based Software System Development (CBSE) Process.

2 Workshop goals

The goal of this workshop is to address issues involved and surrounding the development of Component-Based Software System Development (CBSE) processes. As a result, this workshop will target research and industrial works and experiences that deal with the issues as they related to CBSE:

Requirements engineering and analysis

Inadequate, incomplete, erroneous, and ambiguous system and software requirements are a major and ongoing source of problems in systems development. These problems manifest themselves in missed schedules, budget excesses, and systems that are to varying degrees unresponsive to the true needs of the sponsor. These difficulties are often attributed to the poorly defined and ill-understood processes used to elicit, specify, analyze, and validate requirements [2].

Domain engineering and analysis

Domain engineering is the process of defining the scope (i.e., domain definition) analyzing the domain (i.e., domain analysis) specifying the structure (i.e., domain architecture development) building the components (e.g., requirements, designs, software code, documentation) for a class of subsystems that will support reuse. Domain analysis focuses on supporting systematic and large-scale reuse (as opposed to opportunistic reuse, which suffers from the difficulty of adapting assets to fit new contexts) by capturing both the commonalties and the variabilities of systems within a domain to improve the efficiency of development and maintenance of those systems. The results of the analysis, collectively referred to as a domain model, are captured for reuse in future development of similar systems and in maintenance planning of legacy systems (i.e., migration strategy) [1]

Interface specification languages and tools

Interface languages are used to specify the interfaces between program components. Each specification provides the information needed to use an interface. A critical part of each interface is how components communicate across the interface. Communication mechanisms differ from programming language to programming language. For example, some languages have mechanisms for signaling exceptional conditions, other do not. More subtle differences arise from the various parameter passing and storage allocation mechanisms used by different languages.

Product and technology evaluation

To ensure that systems meet immediate user requirements, component qualification practices must be developed. Typically, we describe products in terms of interfaces that provide access to functionality. Here, standards may provide a frame of reference for comparing the product to generally accepted capabilities. Various approaches have been developed for evaluating products in terms of their interfaces. For example, [5] describes a rigorous process for selecting between competing COTS products.

However, a broader interpretation of the interface to a product includes far more than its functionality. To make use of a product one must also understand aspects of performance, reliability, flexibility, etc., as well as the implicit assumptions made by the product about the operating environment. For example, while examination of the published interface of a product may suggest that it can interoperate with a second product, interoperation may be limited by each product's assumption that it has primary responsibility for handling incoming events. Much of this sort of information is not addressed by standards and is unavailable from product suppliers. Thus, hands-on evaluation to identify such mismatches (alternately called architectural mismatches [6], and interface mismatches by [7]) must be a primary option.

Integration

In CBSD, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centerpiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because of this, integrability is a key consideration in the decision whether to acquire, reuse, or build the components.

Modeling

A model is a representation of a system whose representations are used to answer questions about the system. The reason for creating a model, rather than examining the system as a whole is that the model can abstract away from unimportant details and allow us to more easily investigate the relevant questions.

Software models are created to examine the function, performance, safety, security, and availability of a system or a component. Functional models are created to ensure that a system will behave as expected.

System behavior can be predicted by creating situations and using the model to test them. If the model's behavior is satisfactory we gain some confidence that in the same situation the system will perform as desired. One reason for modeling is that we expect the creation of a model to be cheaper than the creation of a system. In such cases, the model is a predictor of the future system and development requires that the system conform to the model. Sometimes we model a system or component after the fact, we may already have the component and, through testing or other investigative techniques we uncover facts about the component which we build into our model. Again, we use the model to predict the behavior of the component when placed in certain situations. We might choose this approach if we wish to create a system using a particular component where the failure of the component could cause catastrophic failure of the system. Safety or security related systems might take this approach.

Design and development

A fundamental change necessary in developing component-based systems is the simultaneous definition and tradeoffs among the system context, the architecture and design, and the availability of products in the marketplace, as shown in Figure 1 Component-based development

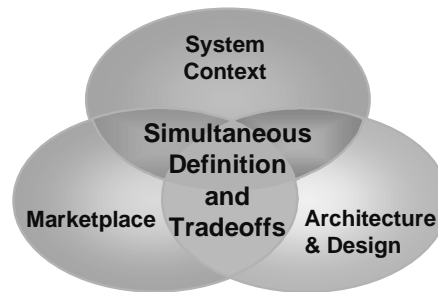


Figure 1 Component-based development

Risk analysis and management

Risk management is a practice with processes, methods, and tools for managing risks in a project. It provides a disciplined environment for proactive decision making to assess continuously what could go wrong (risks) determine which risks are important to deal with and implement strategies to deal with those risks.

Process models

A process model is a relatively detailed, formal or semi-formal representation of a process. The primary users of process models are process engineers, who analyze, assess, design and monitor processes for continuous process improvement and process automation, and process participants, who perform the processes or are interested in their performance (e.g., project managers). Process models can support a wide range of uses, and some of the most common are:

- ◆ as a mechanism to help people understand and visualize a process (especially graphical models),
- ◆ as a basis for engineering (i.e., developing, evaluating, improving, etc.) a process, and
- ◆ as the means of formalizing a process for machine-assisted enactment by a process-centered software engineering environment, workflow engine, or the like.

Numerous process-modeling notations have been proposed and applied in practice. Process models have been used to describe existing (as-is) processes and prescribed processes (e.g., standards, regulations), evaluate them for desirable characteristics and improvement opportunities, and develop and analyze new (to-be) processes. They have also been used to quantitatively simulate and analyze processes in support of management planning and control, and process improvement [3].

System frameworks and architectures

Components implement two kinds of interface: a functional interface that reflects the role of a component in the system, and another extra-functional interface that reflects the component model imposed by some underlying system framework [8]. These extra-functional interfaces express the architectural constraints that enable composeability and other desirable properties of component-based systems. Therefore, our understanding of what makes a component a component is inextricably linked to our understanding of the architectural constraints imposed on components by a system framework.

System maintenance

The one constant aspect of component-based software development is change [4]. Constituent components are constantly evolving – new products are emerging while existing products become dated or obsolete. Keeping existing systems “current” is a difficult but necessary maintenance task. In maintaining a best-of-breed solution, the maintainer needs to continually monitor the marketplace to evaluate new technologies, new products and new releases of component products and upgrade the system when appropriate. These skills are not very different from the skills required to initially develop these systems.

Repository management

The benefits of developing an effective component library are readily apparent: by allowing system integrators to fabricate software systems from pre-existing components rather than laboriously develop each system from scratch, enormous time and energy can be saved in the development of new software systems.

However beneficial a component library might be, a useful and effective repository has turned out to be an elusive goal. Traditional software libraries have been conceived as large central databases containing information about components and, often, the components themselves. Examples of such systems include the Center for Computer Systems Engineering’s Defense System Repository, the JavaBeans Directory, and the Gamelan Java directory.

While the JavaBeans and Gamelan directories are still going concerns, similar systems have failed in the past largely as a result of their conception as centralized systems. Problems with this approach include limited accessibility and scalability of the repository, exclusive control over cataloged components, oppressive bureaucracy, and poor economy of scale (few users, low per-user benefits, and high cost of repository mechanisms and operations).

3 Publishing workshop results

A workshop summary will be published on the WWW via the workshop homepage www.sei.cmu.edu/cbs/tools99.

References

1. Foreman, John. Product Line Based Software Development- Significant Results, Future Challenges. Software Technology Conference, Salt Lake City, UT, April 23, 1996.
2. Requirements Engineering and Analysis Workshop Proceedings, CMU/SEI-91-TR-030, December 1991.
3. Marc I. Kellner, Ulrike Becker-Kornstaedt, William E. Riddle, Jennifer Tomal, Martin Verlage, Process Guides: Effective Guidance for Process Participants, Published in the Proceedings of the 5th International Conference on the Software Process: Computer Supported Organizational Work, International Software Process Association, New Jersey, 1998.
4. Robert C. Seacord, Kurt Wallnau, John Robert, Santiago Comella Dorda, Scott A. Hissam, Custom vs. Vendor-Integrated COTS Software, CMU/SEI-99-TN-003, May 1999.
5. Kontio, J., "A Case Study in Applying a Systematic Method for COTS Selection," Proceedings of the International Conference on Software Engineering, Berlin, 1996.
6. Garlan, D., Allen, R. and Ockerbloom, J., "Architectural Mismatch: or Why It’s Hard to Build Systems Out of Existing Parts," Proceedings of the International Conference on Software Engineering, Seattle, 1995.
7. Wallnau, K., Clements, P. and Zaremski, A. "Correcting, Identifying, and Avoiding Interface Mismatch: Theory and Practice", Draft Paper, Software Engineering Institute, Carnegie Mellon University.
8. Alan Brown, Kurt Wallnau, The Current State of Component-Based Software Engineering (CBSE) IEEE Software, September 1998, pg. 37-47.