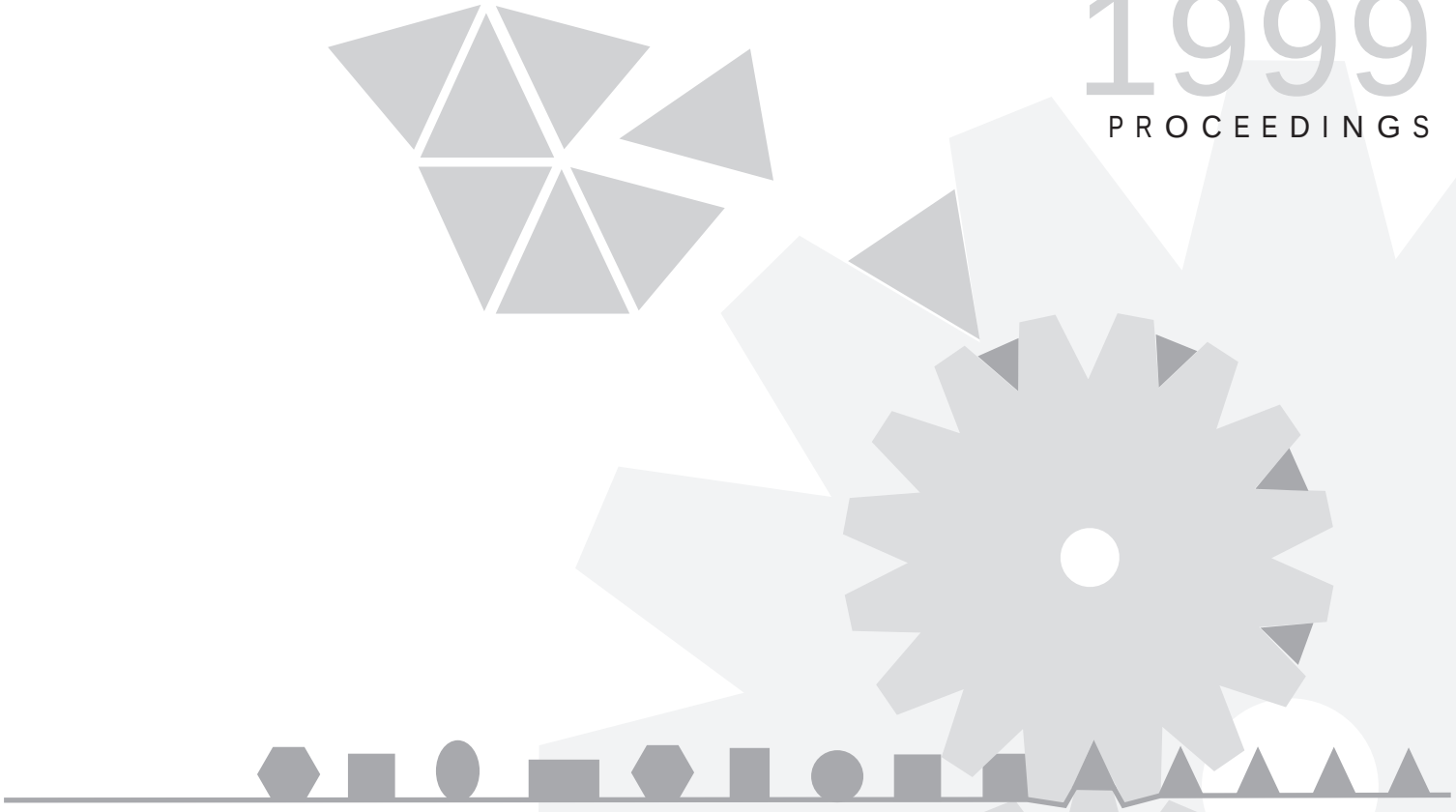
International Workshop on Component-Based Software Engineering

1999

PROCEEDINGS



MAY 17-18, 1999

Table of Contents

PORE: Procurement-Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm

Cornelius Ncube & Neil.A.M. Maiden

Centre for Human-Computer Interface Design
City University, London, UK

Tel: +44 171 477 8166
Fax: +44 171 477 8859
E-Mail: [C.Ncube@soi, N.A.M.Maiden@]city.ac.uk
PORE URL: http://www.soi.city.ac.uk/pore/welcome.html

Abstract

Most current research in Component-Based Systems Engineering (CBSE) focuses on design and integration processes. There is little interest in the requirements engineering and product evaluation/selection processes that must precede design and integration. Also most current methods and tools support systems design and integration but neglect the requirements engineering and product evaluation/selection processes. However, in spite of this lack of focus on requirements engineering, a consensus seems to be emerging that the CBSE development process should be an iterative one of requirements engineering, systems design, product evaluation/selection and systems integration. This paper proposes a new method, PORE, to address the lack of requirements engineering methods and product evaluation/selection process guidance for the CBSE process. The paper ends with a 'vision' for future research directions for component-based systems engineering development process.

Keywords: PORE, systems procurement, requirements engineering and acquisition, COTS software products, product evaluation and selection, process guidance.

1: Introduction

As the next century approaches organisations are increasingly shifting the development processes of their complex software-intensive systems away from bespoke systems development to Component-Based Systems Engineering. Commercial software components that can be procured off-the-shelf (COTS) are now available to perform most of the functions that in the past required bespoke development. This use of COTS components has the potential for reducing the cost and time to develop software intensive systems. However, given the complexities of today's software intensive systems, the cost and risk of procuring/purchasing wrong package(s)/component(s) due to inadequate requirements acquisition and product selection is large. The success of a Component-Based Systems Engineering, (CBSE) development process largely depends on the successful selection of COTS software component that meet core essential customer requirements.

However, the problem is that when building systems from COTS products, new and different types of requirements, (e.g. contractual and supplier requirements) need to be defined and new methods and techniques of acquiring these requirements and selecting candidate COTS products and suppliers also need to be defined. We are developing a new method, PORE, (Procurement-Oriented Requirements Engineering), (e.g. Ncube & Maiden 1997) which supports the requirements engineering and product evaluation/selection processes for CBSE development process. PORE uses an iterative process of requirements acquisition and product evaluation/selection as its main novel approach. The PORE approach has three main components:

- a process model that identifies four essential goals that should be achieved by any CBSE process and prescribes generic processes to achieve each of these goals as well as a sequence in which these goals should be achieved;

- a method box that includes methods, techniques and tools that are available to help undertake and achieve each of the processes;
- a product model that provides semantics and syntax for modelling software products

The three components are integrated into an approach that provides a requirements engineering team with a coherent process guidance for CBSE development process.

Section 2 outlines PORE's major components and in particular, its life-cycle processes. Section 3 describes PORE's iterative process of requirements acquisition and product evaluation/selection. Also described in section 3 are the methods, techniques and tools used in PORE for requirements acquisition and product identification, evaluation and selection. Section 4 describes PORE's situated process and how the requirement and software product models inform the iterative process of requirements acquisition and product evaluation/selection, therefore guiding the CBSE process. The paper ends with an outline and a vision for future research directions.

2: PORE: A Requirements Engineering Method For the CBSE Process

The basic PORE life-cycle process model has six generic processes. The process model describes the most fundamental processes undertaken during COTS product procurement. The processes are defined at 3 levels according to Humphrey's (1989) process model:

- the universal (U) level processes that describe general guidance for the actors in the process. Each describes a uniform sequence of processes;
- the worldly (W) level processes that are relevant to iterative process of requirements acquisition and product evaluation and selection. Each guides the sequence of tasks during product procurement.
- the atomic (A) level processes that are specific to individual methods, procedures, techniques and tools which enable the W-level processes.

Figure 1 depicts the six generic U-level PORE processes. It shows the processes which are often undertaken, although not all the processes are performed during each product procurement. For example, contract production does not take place if the component is procured internally. Management takes place throughout the procurement process. Software package selection and requirements acquisition are performed iteratively. Supplier selection and package selection often take place at the same time. Each processes has sub-processes and their main objectives are described below.

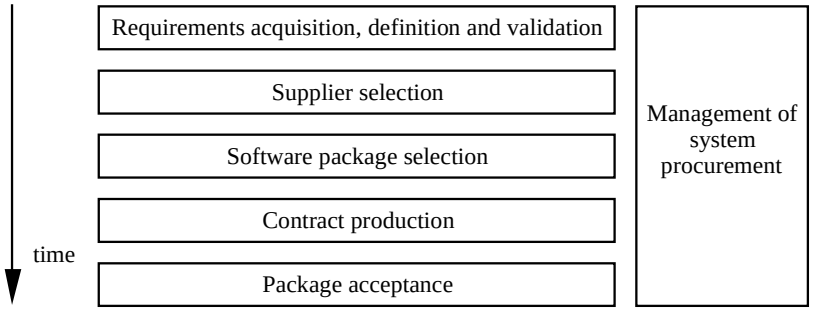


Figure 1: The Basic Overview of the PORE Process Model

- Management of System Procurement process – the objectives of this process are to plan and control the procurement process in order to fulfil the needs of the procurer in time and at a reasonable cost.
- Requirements Acquisition process – this process acquires and validates customer requirements. It also determines the current system architecture so that new component(s) can be integrated with it.

- Supplier Selection process – this process's objectives is to establish supplier selection criteria, evaluate suppliers, rank them according to the criteria and select the best-fit supplier(s).
- Software Package Selection – this process's objective is to identify candidate packages, establish selection criteria using customer requirements, evaluate the identified packages, rank them according to the criteria and select one or more package(s) which best meet the core essential customer requirements.
- Contract Production – this process's objective is to negotiate the legal contract with package suppliers and resolve an legal issues pertaining to the purchasing of the package and licensing.
- Package Acceptance – the objective of this process is to check the delivered package or system against the customer's original core essential requirements.

The main focus of this paper is the two processes of requirements acquisition and package selection. We aim to report work on the remaining processes in the near future.

3: Iterative Requirements Acquisition and Product Evaluation/Selection

In the CBSE development process, requirements are the cornerstone for any effective COTS procurement. Requirements become criteria for evaluating and selecting of candidate components and are embedded in the legal contract (contractual requirements). Requirements even provide acceptance criteria to check when the system or package is delivered that it meets customer's expectations. In CBSE the consequences of inadequate requirements engineering can be greater because COTS products are living systems and evolve over a long time. In a COTS intensive system, the integrated components will be developed and updated by different vendors at different development, update and evolution cycles hence requirements changes often will incur great additional costs to the customer.

However, the importance of requirements engineering to CBSE is not reflected in the current range and focus of research activities and in the available commercial methods, techniques and tools. Most current methods and tools support systems design (e.g. Garlan 1995) and integration (e.g. Vigdar et al 1996, Brown et al 1995) but neglect the requirements engineering and product evaluation/selection processes that must precede design and integration. In spite of this lack of focus on requirements engineering, there is an emerging consensus that the process of developing systems from COTS components should be an iterative one of requirements engineering, systems design, product evaluation/selection and product/systems integration (Fox et al 1997, Tran & Liu 1997).

Hence at the heart of PORE is the iterative and parallel process of requirements acquisition and product evaluation and selection. PORE's iterative process selects products by rejection, (i.e. the products that do not meet core customer requirements are selectively and iteratively rejected and removed from the candidate list). At the same time the products are selectively rejected, therefore resulting in a decreasing number of candidate products, the number and detail of customer requirements will be increasing. The result is an iterative process whereby the requirements acquisition process enables product selection and the product selection process informs requirements acquisition. This process is depicted Figure 2 below.

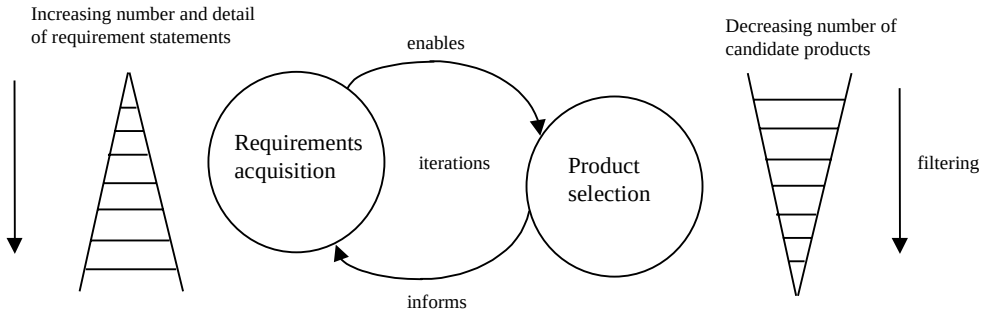


Figure 2: Overview of the PORE's iterative process. Requirements acquisition enables product selection and product selection informs requirements acquisition. As the number and detail of requirements increases, the number of candidate products decreases.

3.1: Methods, Techniques and Tools Integration

Within this iterative process, the PORE method integrates different methods, techniques and tools for requirements acquisition and product identification and evaluation/selection with process guidance for choosing and using each technique. Some of the techniques, methods and tools integrated within the iterative PORE process are indicated below:

- Knowledge engineering techniques such as card sorting and laddering (e.g. Rugg and McGeorge 1995) which are useful when acquiring information about categories of products, suppliers, contracts and hierarchical information about product properties and customer requirements.
- Feature analysis techniques (Kitchenham & Jones 1997) to aid when scoring the compliance of each product feature to each customer requirement;
- MCDM (Multi-Criteria Decision Making) techniques such as AHP (e.g. Saaty 1990) and Out Ranking method (e.g. Fenton 1994) to aid in the decision making process during the complex product ranking and selection process;
- Argumentation techniques (e.g. Buckingham-Shum 1994) to record and aid the decision-making process;
- Requirements engineering methods such as Volere (e.g. Robertson 1997) for aiding the requirements engineering process;
- Requirements acquisition techniques such as ACRE (e.g. Maiden and Rugg, 1996) for acquiring customer requirements;
- Product (or component) identification tools such as the internet or Agora (e.g. Robert et al 1998) for identifying products or components available in the market;
- ATA (Architecture Tradeoff Analysis) for analysing architectures and SAAM (Software Architecture Analysis Method, SEI 1998) for evaluating software product architectures.

As well as integrating these techniques, PORE also provides guidelines for designing product evaluation test cases and for organising evaluation sessions. The guidelines are provided using different templates (e.g. Maiden & Ncube 1998) for requirements acquisition and product selection. The process is structured into stages and the first three templates provided are:

- Template-1, to guide the requirements engineer when acquiring essential customer requirements and product information sufficient to select and reject products as a result of supplier-given information;
- Template-2, to guide the requirements engineer when acquiring customer requirements and product information sufficient to select and reject products from supplier-led demonstrations using test-cases for individual requirements;
- Template-3, to guide the requirements engineer to acquire customer requirements and product information sufficient to select and reject products as a result of customer-led product exploration

Each template defines the types of product information (e.g. supplier, contract, architecture), types of requirements to acquire (e.g. functional, contractual, architectural, non-functional), requirements acquisition techniques to use and the best decision-making techniques to use. Figure 3 below shows each template within the iterative process stages.

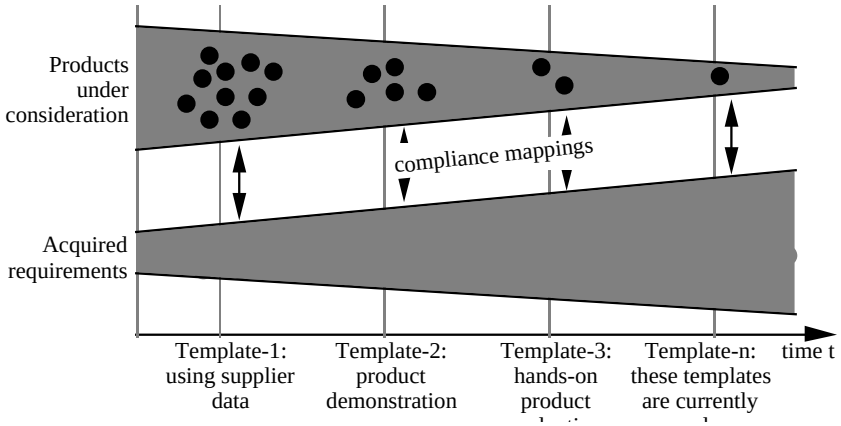


Figure 3: PORE's templates within the iterative process stages. Because of this iterative nature of the process, this means that each template can be used several times. Also shown in the figure process goals that each template must achieve and these are explained below.

3.2: Goal-Based Process Guidance

Another essential feature of PORE is that it prescribes four essential goals for selecting/rejecting candidate products. The four essential goals are integrated with templates as shown in Fig. 3. PORE rejects products according to compliance with:

- essential atomic customer requirements, (Goal 1);
- non-essential atomic customer requirements, (Goal 2);
- complex non-atomic customer requirements, (Goal 3);
- customer user requirements, (Goal 4).

The requirements engineering team must achieve these goals in a sequence. For this PORE prescribes four generic processes to achieve each of the four essential goals. The four generic processes are depicted in Figure 4 and are:

- acquire information about software products, customer requirements, suppliers requirements and contractual requirements from stakeholders;
- analyse acquired information
- use decision-making techniques to analyse and determine product-requirement compliance;
- reject one or more non-compliant candidate products

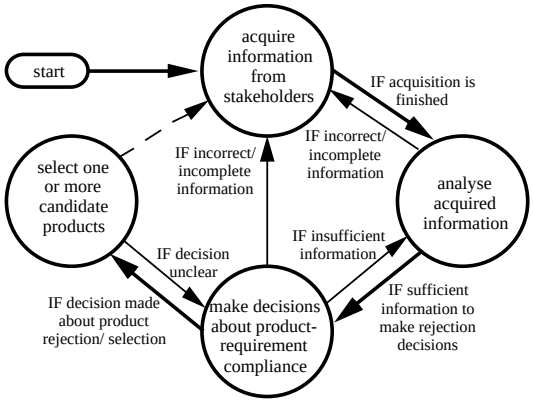


Figure 4: A route map showing PORE's high-level generic processes for achieving each essential goal

The order in which the four processes are undertaken is context-driven and is determined by the current state of the situation model which is depicted in Figure 5. PORE's situations are as defined in Suchman (1987) which states that "every course of action depends in essential ways upon its material and social circumstances" (p50). The CBSE process is a very complex, so a large number of "situations" are possible at any point in the process. In PORE, our solution is to enable the RE (Requirements Engineering) team to model the "material circumstances" as situations that inform process guidance.

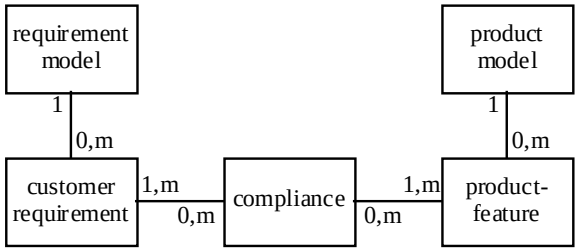


Figure 5: The structure of the situation model, and the relationships between the requirement, product and compliance sub-models.

As Figure 5 shows, a PORE situation is modelled in three parts. The first part models the current state of the customer's requirements. The second part models the degree of compliance between customer requirements model and product model, (product model and requirements model are discussed in more detail in section 4). Compliance is modelled as set of relationships between one or more customer requirements and one or more product features. The third part of the situations is a model of a software product.

PORE's context-driven process is made more complex by the large number of situations which may arise at any point in the process and many techniques from different disciplines (e.g. see section 2) which are available to achieve each process. As a result, PORE provides a multi-layered process guidance. As depicted in Figure 6, at any point in the process, three levels of guidance are provided. The process model level provides guidance to achieve the essential process goals. The situation model provides guidance at the other two levels. At the second level, it recommends techniques(s) to use to undertake a process by inferring general properties about the requirements, product and compliance models. At the third level, it recommends the content focus for applying each technique based on inferences about the current contents of the requirements, product and compliance sub-models. The process guidance is therefore given in the form of a triplet:

{process-goal, techniques-to-use, content-focus}

Of the three triplet contents, the process-goal part changes least and the content-focus changes most during each instance of a process as new information is added to the three sub-models and new inferences about the properties of the model are being made.

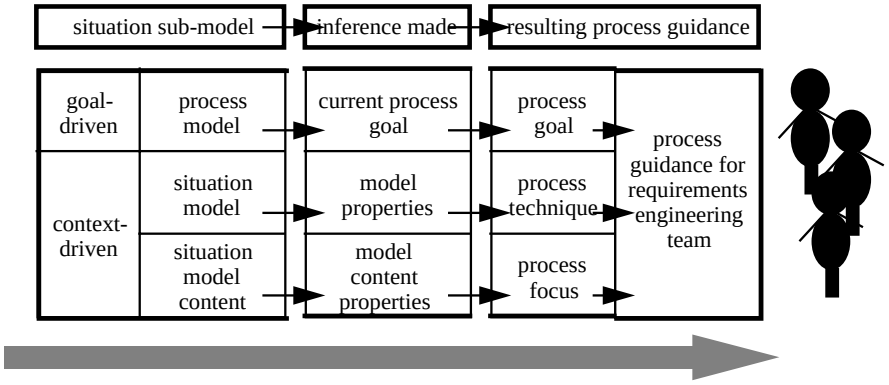


Figure 6: The three levels of process guidance which form the PORE process triplet. The current process goal is inferred from the process model. The technique to achieve this process is inferred from properties of the situation model. The focus of this technique's application is inferred from properties of the situation model content

4: Models for Guiding the CBSE Process

At the heart of the PORE method are three models, the product model, the requirements model and the product-requirement compliance model, see Figure 5. These models are used an as instrument for comparisons and evaluation of candidate software products and for driving and guiding the requirements acquisition and product selection processes.

4.1: Product Model

To enable effective COTS product evaluation and selection, there is a need to understand software products. PORE's product model (Ncube and Maiden 1998) enables each software product to be modelled in three different ways:

- Firstly, the product model models the observable behaviour of the product and in particular how the user interacts with the product. To achieve this, we draw on the existing use case modelling approaches, in particular the CREWS (Co-operative Requirements Engineering With Scenarios, Maiden et al, 1998).
- Secondly, the product model also models the product's articulated goals using goal-based requirements methods (e.g. Anton 1997).
- Thirdly, it also models the product's architecture using architecture modelling techniques such as those reported in Shaw (1996), Garlan et al. (1995) and SEI (1998).

To precisely model the properties of the COTS software product, the PORE approach uses a variety of abstract meta concepts such as goals to be achieved, objects to be used, actions taking place, functions to achieve actions and relationships between the meta-concepts, (Ncube & Maiden 1998). Figure 7 depicts the PORE's software product meta-model and its primitive concepts (e.g. agent, action, software component) and the meta-relationships linking the meta-concepts (e.g. performs, achieves, depends). The meta-concepts provide specific ways and strategies for traversing the product meta-model to instantiates its instances during requirement-product compliance mapping. Each step in the product-requirement mapping becomes a validation of the meta-model concepts against essential customer requirements.

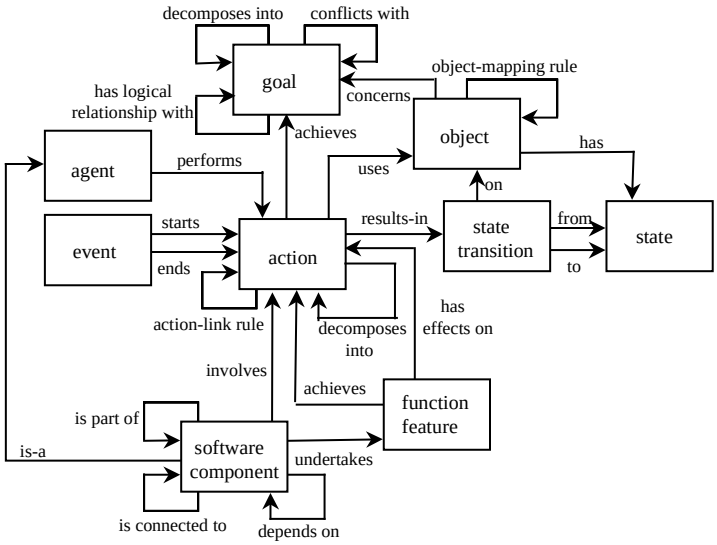


Figure 7: A meta-model for the software product showing the primitive concepts and relationships with which to model each software product.

4.2: The Requirement Model

The PORE method uses the requirement model to both acquire and elaborate the requirements statements and to check requirement-product compliance during the product evaluation and selection process. A critical factor in a successful requirements acquisition is to understand not only what the system under consideration should do (functional requirements), but also the way in which it should provide its services (non-functional requirements). A broader view of requirement acquisition, therefore should go beyond the description of what the system is expected to do (the system's functionality) and include system properties and constraints under which the system must operate, e.g. (architecture requirements). In the CBSE development process, this view is even taken further to include information about product suppliers such as their technical capabilities, application domain experience, ISO standard certification, CMM level, etc. and legal issues involved in product procurement such as negotiating contract terms and conditions, licensing arrangements, etc. When taken in this context requirements represent both a model of what is needed and statements of the

problem under consideration in various degrees of abstraction. Figure 8 depicts the meta-concepts of the requirement model.

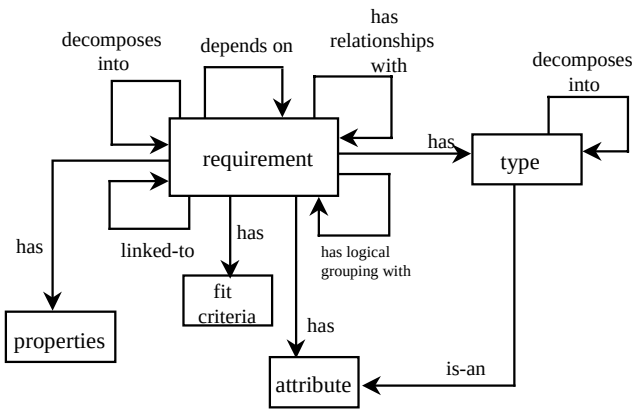


Figure 8: The requirement model and its meta-concepts.

5: Discussion and Further Development of PORE.

We believe that PORE is a novel approach to guiding the CBSE development process. In particular, it addresses the lack of process and method guidance for requirements acquisition and component/product evaluation/selection processes which must take place before system design. However, the PORE approach still has significant limitations that we aim to overcome in future developments. One of the problems, as mentioned above, is that the iterative process of requirements acquisition and product evaluation/selection is very complex. At any point in this complex process, a large number of possible situations can arise. For example, stakeholders may define a large number of requirements, so the requirements sub-model can give rise to a large number of different situations. The requirements engineering team can also evaluate a large number of software products, so the product and compliance sub-models as well, can give rise to a very large number of situations. In addition, PORE can sometimes recommend a large number of processes and techniques to use in a single situation. To handle this scale of complexity, we believe that a software tool is needed that has a computational model to detect situations and generate process guidance.

As a result, a prototype tool known as PORE Process Advisor is being developed to support the PORE approach. The main components of the tool are a process engine which will analyse the current set of goals to be achieved (stored in the goal agenda), model properties (inferred by the situation inference engine) and instructions from the requirements engineering team to recommend process advise in the form of the process triplet, i.e. <process-goal, technique, focus>. The PORE Process Advisor tool is being developed to integrate with existing software tools like Microsoft Access, Visual Basic, and Kappa-PC. It is also linked to Rational's RequisitePro requirements management tool (for requirements management) and CREWS-SAVRE tool (e.g. Maiden et al. 1998) for generating scenarios. We shall be reporting usage trials of PORE Process Advisor tool in the near future Further details of PORE in general can be found in the dedicated web site: http://www.soi.city.ac.uk/pore/welcome.html. We welcome any feedback, suggestions and participation in the development of PORE.

6: Future Research Directions for the CBSE Paradigm

We believe that the vision for the future research directions will be in the 'shared knowledge' development process and in new directions in the training and education of the systems developer of the future as well as the organisations which will be developing these COTS-based systems and the user organisations. A whole new set of skills in both project management and systems development will be required for the CBSE process. A brief outline of the two research directions is given below.

6.1. Shared Knowledge Development Process

As the shift to CBSE development will rapidly gather speed in the near future, we envision a 'vision' of a process whereby shared knowledge of products, development skills and experiences, new techniques and methods as the way forward. This shared knowledge will result in a supply chain of components/products, skills and experiences and personnel. Also organisations will have access to each other's development infrastructure along the supply chain and share business strategies and objectives, expertise, ideas, risks and information. Joint technology development programmes will be possible through shared knowledge and close relationships. This vision of a shared knowledge process is depicted in Figure 9 below.

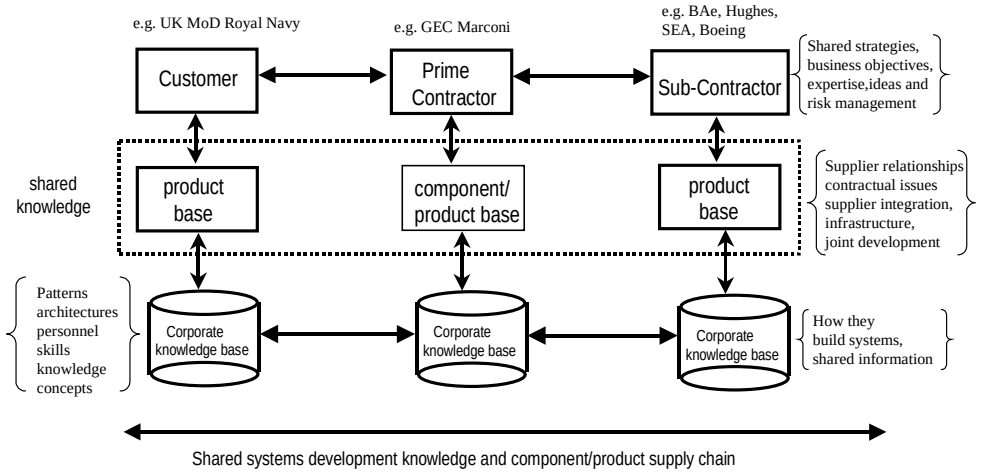


Figure 9: A vision of shared knowledge as the cornerstone for the future success of the CBSE paradigm.

6.2: The 'soft' Issues: Training and Education

The number of techniques and knowledge from different disciplines required in the CBSE process for it to be a success will mean that it will be impossible for any individual to possess all the necessary skills. As a result the development team of the future will be composed of team members from many backgrounds therefore forming 'smart teams'. The project management and development skills that are required for the CBSE process are significantly different from those required for traditional systems development. Even the project manager of the future will be required to do significantly different task as opposed to what a traditional project manager has to do. Organisations themselves will probably have to change the way they do their business. For example, selecting a product to be included in the integrated systems largely results in the selection of the product supplier. Therefore, in a COTS-

intensive system, the integration of the different products results in the integration of product suppliers as well. As such, the development organisation will need to manage not only its relationships with the individual suppliers but also the relationships between the integrated suppliers. As a result, we have identified personal or ‘soft’ issues as major research area. A different sets of skills will be required and therefore CBSE principles need to be incorporated into the training and education of systems developers of the future, be it in universities, colleges or organisation training programmes.

7: References

Anton A. I., 1997, 'Goal Identification and Refinement in the Specification of Software Based Information Systems, PhD Thesis, Georgia Institute of Technology, June 1997.

Fenton N., 1994, 'Out Ranking Method for Multi-Criteria Decision Aid: with emphasis on its role in systems dependability assessment, Centre for Software Reliability, City University, London, UK.

Fox G., Marcom S. & Lantner K, 1997, A Software Development Process for COTS-based Information System Infrastructure, Proceedings of the 5th International Symposium on Assessment of Software Tools and Technologies (SAST'97), pp133-143

Garlan D., Allen R. & Ockerbloom X., 1995, 'Architectural Mismatch or Why it's hard to build systems out of existing parts', Proceedings 17th International Conference on Software Engineering, IEEE Computer Society Press.

Humphrey W.S., 1989, 'Managing the Software Process', Addison-Wesley

Kitchenham B & Jones L., 1997, 'Evaluating Software Engineering Methods and Tools: Part 5, The Influence of Human Factors', Software Engineering Notes 22(1).

Maiden N.A.M. & Rugg G., 1996, 'ACRE: Selecting Methods for Requirements Acquisition, Software Engineering Journal 11(5), 281-292.

Maiden N.A.M., & Ncube C., 1998, 'Acquiring COTS Software Selection Requirements', IEEE Software, March/April 1998 Issue, 46-56

Maiden N.A.M., Minocha S., Manning K. & Ryan M., 1998, 'CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements', Proceedings 4th International on Requirements Engineering (ICRE98), IEEE Computer Society Press.

Ncube C & Maiden N.A.M., 1997, 'Procuring Software Systems: Current Problems and Solutions', Proceedings REFSQ97 workshop, CaiSE97, Barcelona, Spain, June 16-17

Ncube C & Maiden N.A.M., 1998, 'Why Model Software Products: Why, Guiding the CBSE Process, of Course!', Proceedings of the CBISE98 workshop for CAiSE' 98, Pisa, Italy.

Robert C. S., Scott A. Hissam, & Kurt C.W, 1998, 'AGORA: A Search Engine for Software Components, Software Engineering Institute, Carnagie Mellon University, USA.

Robertson S., 1998 'Volere: Requirements Specification Templates, Edition 6' , Atlantic Systems Guild, http://www.atlsysguild.com/GuildSite/Robs/Template.html,

Rugg G. & McGeorge P., 1995, 'Laddering', Expert Systems 12(4), 339-346

Saaty T. L., 1990, 'The Analytic Hierarchy Process', New York, McGraw-Hill.

SEI, 1998, 'Architecture Tradeoff Analysis (ATA) and Software Architecture Analysis Method (SAAM)', Software Engineering Institute, Carnegie Mellon University, USA, http://www.sei.cmu.edu/ata/ata_init.html

Shaw M., 1996, 'Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does', Proceedings of the 8th International Workshop on Software Specification and Design, IEEE Computer Society Press, March 1996.

Suchman L., 1987, ‘Plans and Situated Actions: The Problems of Human-Machine Communication’, Cambridge University Press.

Tran V. & Liu D, A Procurement-centric Model for Engineering Component-based Software Systems, Proceedings of the 5th International Symposium on Assessment of Software Tools and Technologies (SAST’97), pp70-80

Vigdar M.R., Gentleman, M. W. & Dean J., 1996, ‘COTS Software Integration: State of the art’, National Research Council, Canada, NCR-CNRC Report, January 1996.

A Component Model Proposal

Jim Q. Ning

Andersen Consulting

3773 Willow Road

Northbrook, IL 60062, U.S.A.

+1 847 714 2537

jning@cstar.ac.com

Abstract

This position paper describes a conceptual model for Component-Based Software Engineering (CBSE). The model is an attempt to define what CBSE is essentially about and help answer critical questions concerning how CBSE relates to and distinguishes itself from other software development paradigms/concepts such as object-orientation.

1. Motivation

As clearly stated in the opening statement of this year's CBSE Workshop's theme description:

> "There is growing interest in the notion of software development through the planned integration of pre-existing software components. This is often called component-based development (CBD), component-based software engineering (CBSE), or simply componentware. While the broad concepts of CBSE are well known and easily stated, a closer look reveals that the term CBSE is used in a diverse set of situations, encompasses a variety of characteristics, and is often given many different interpretations."

To clarify the misconception and confusion, this position paper proposes a model or framework which, hopefully, will be simple and easy to explain to people on one hand and yet rich enough to capture all the key component-related concepts. In the next section, a concept diagram is presented to depict the key concepts and their relationships. Then, a glossary is provided to further define/describe the concepts. Finally, I will explain why such a component model is important and beneficial to the CBSE community.

During the workshop, I anticipate to receive comments from the participants to improve the proposed model. The objective is for this young community to reach some consensus on what CBSE is essentially about and help answer critical questions concerning how CBSE relates to and distinguishes itself from other software development disciplines such as object-orientation.

2. Concept Diagram

The concept diagram is shown on the last page of this paper (Appendix). It is meant to illustrate the important CBSE concepts, the relationships among themselves and with other software engineering concepts. The component concepts are shown in plain (white) boxes, and other concepts in gray boxes. The following link types are used to describe the relationships between the concepts:

- **Aggregation**. This is shown as a solid-line path with a hollow diamond at one end. The concept that is connected by the diamond end is the aggregate.

- **Association**. A binary association is shown as a solid-line path that connects two concepts. A ternary

association is shown as a diamond with a path from the diamond to each participant concept. The cardinality of a participant concept is optionally shown at the end of an association path.

- **Generalization**. This is shown as a solid-line path from the more specific concept to the more general one, with a hollow triangle at the end where the path meets the general concept.

The two vertical dotted lines in the diagram group the concepts into *component concepts* (those in the left column), *interface concepts* (those in the middle column), and *connector concepts* (those in the right column), the three cornerstones of CBSE. The two horizontal dotted lines further categorize the concepts in terms of when they are created during the software lifecycle: the what I call *Type concepts* at the top row are developed at the design/specification time, *the Instance concepts* at the bottom row are created at runtime, and the concepts in the middle row are typically generated during system construction. More precise definitions of these concepts can be found in the next section. It should be noted that no Type concept equivalent is shown for a component concept. It is my belief that a component type at the specification level will be too abstract to be useful.

The gray boxes around the borders of the diagram show how more conventional software artifacts relate to component concepts. Note that this diagram is not meant to be exhaustive. The gray boxes are selectively drawn to exemplify what the component concepts are (e.g., a Component Instance is an Executable), what they are composed of (e.g., a Class is a part of a Component), how they interact with other concepts (e.g., a Connector Instance receives services from an Object Request Broker), etc.

3. Glossary

This section provides definitions/descriptions of the component concepts shown in the concept diagram.

- **Component** – An encapsulated, distributable, and executable piece of software that provides and receives services through well-defined interfaces.

- **Component Instance** – The runtime manifestation of a component. It is typically a runtime image or a piece of code run within some runtime image.

- **Interface Type** – An abstract specification of a set of behaviors without the concern of how to implement the behaviors.

- **Interface** – The association of an interface type to a component to make the services provided or required by the component externally visible.

- **Provided Interface** – An interface representing the services supported by a component to the external world.

- **Required Interface** – An interface representing the services that must be received by a component from outside in order for the component to perform its own operations.

- **Interface Instance** – The runtime manifestation of a interface. It is typically the proxy, stub, and marshaling code packaged and run within some component instances.

- **Connector Type** – An abstract specification of a style of interaction among components.

- **Connector** – The association of a connector type between the required and provided interfaces of components.

- **Connector Instance** – The runtime manifestation of a connector. It can be an independent runtime image or be packaged as part of the proxy and stub code within some component instances.

4. Benefits

What can we gain by having a component model that will be commonly accepted by the CBSE community? First, we

will have a common framework to talk about components in particular and CBSE in general. How many times have we seen a meeting or workshop getting into endless discussions on what is and is not a component? Amazingly, however, no consensus has been reached after all these discussions – a sign of an emerging but immature discipline. The outcome of this work will hopefully become a starting point for the community to brainstorm on and evolve with to build a common understanding and a solid foundation for CBSE.

Another benefit of having a well-defined component model is to help distinguish component concepts from other related concepts. For example, using the concept diagram given in this paper, we can easily state that a program module may form part of a component implementation. But a module by itself is not a component - a module in the conventional sense does not include explicit descriptions of provided and required interfaces. Similarly, we can decide that an object in the object orientation sense is not a component either. An object can be an identifiable entity that lives within some component instance at runtime. But it does not fit our general definition of a component.

Yet another benefit is that such a model helps us understand how the component concepts exist within the context of known technology pieces. For example, the diagram tells us that the concept of interfaces is not entirely new, ports and application programming interfaces (APIs) are all special cases of component interfaces. The diagram also shows that a connector instance does not exist on its own; it must be supported by some runtime infrastructure services such as transaction monitors.

There is no question that a good component model is long overdue for the CBSE community.

References

1. Bronsard, F., Bryan, D., Kozaczynski, W., Liongosari, E., Ning, J., Ólafsson, Á, and Wetterstrand, J., "Toward Software Plug-and-Play," in *Proceedings of the Symposium on Software Reusability*, 1997.

1. D'souza, D. F., and Wills, A. C., *Objects, Components, and Frameworks with UML – The Catalysis*^{TM} *Approach*, Addison-Wesley, 1998.

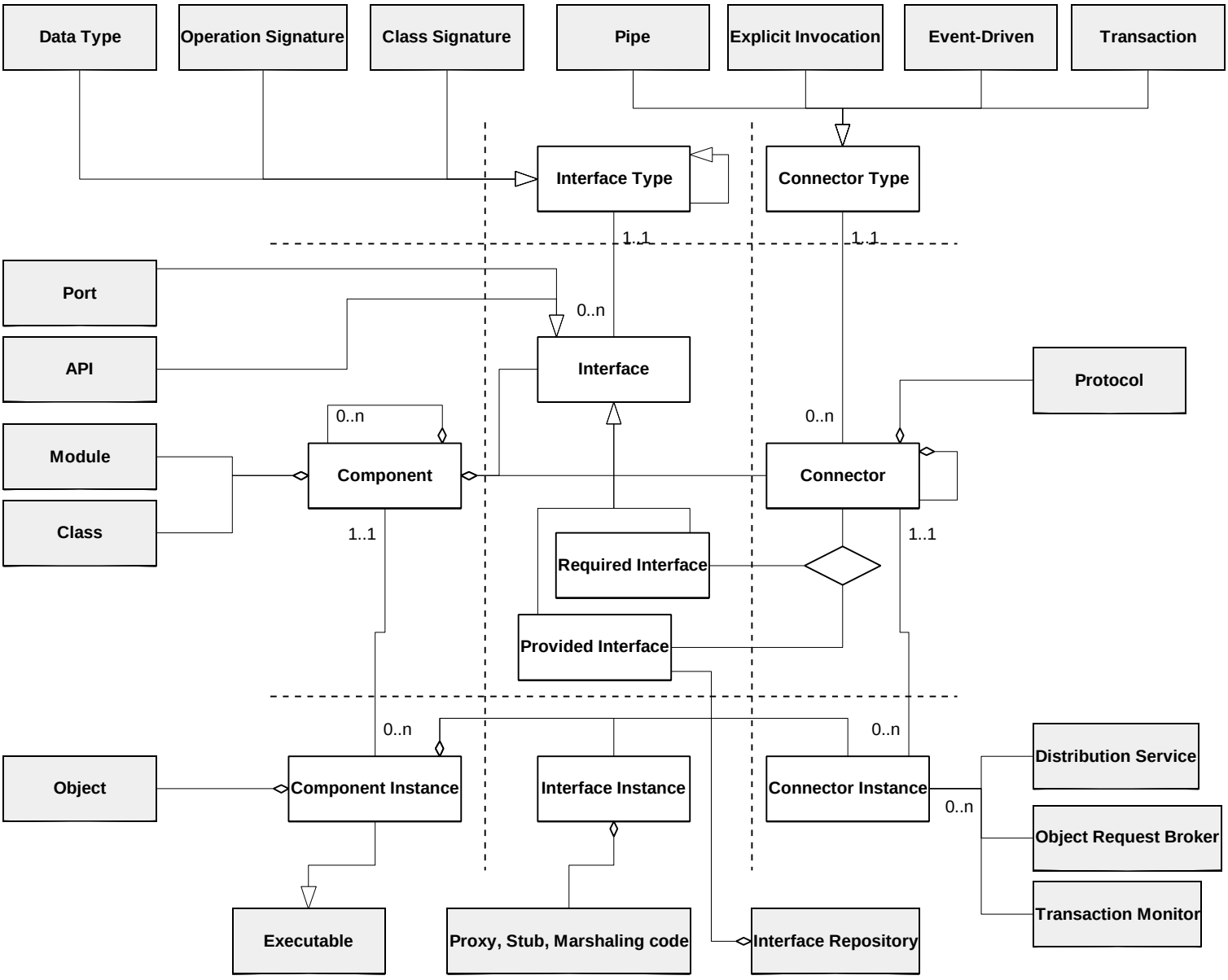
2. Hofmeister, C., Nord, R., and Dikip Soni, *Applied Software Architecture – Draft*, 1998.

3. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1995.

1. Rational Software, *UML Notation Guide*, Version 1.1, September 1997.

1. Shaw, M., and Garlan, D., *Software Architecture – Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

Appendix – Component Concept Diagram



An Evaluation of Component Adaptation Techniques

George T. Heineman
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609, USA
heineman@cs.wpi.edu
WPI-CS-TR-99-04

Abstract:

One of the many difficulties in making Component-Based Software Engineering (CBSE) a reality is that software components may require adaptation when constructing applications from COTS components. We survey the literature to discover various approaches to component adaptation and evaluated these approaches against a set of requirements for component adaptation mechanisms. We also discuss differences between adaptation of software components and extension of object-oriented classes.

---

1. Introduction

The closing sentence of a recent report on the current state of CBSE states that the growing use of external components will demand improvements in how components are documented, assembled, adapted, and customized [3]. This position paper addresses the issue of adaptation.

We have argued in [8, 6, 7, 9] that a true component marketplace will only exist when application builders can *adapt* software components to work within their application. For this position paper, we surveyed the literature for different approaches to adapting software components. Our primary contribution is to show that component adaptation is a highly relevant problem to CBSE. Component adaptation is sufficiently different from software evolution that it requires new techniques and certainly new understanding to solve its challenges.

We first motivate the need to adapt third-party COTS components after they have been designed, implemented, and made available for purchase. We then discuss the differences between adaptation of components and adaptation of object-oriented programs. We then evaluate various approaches to component adaptation against a set of requirements for adaptation mechanisms.

2. Motivation

An application builder has designed and partially implemented a software system using several reusable in-house software components. The builder finds an externally available third-party software component that satisfies some desired functionality or behavior. Because there are such difficulties in accurately specifying software, however, the builder is not totally sure that the component will completely perform all the desired tasks; in fact, the component may contain additional unneeded features that are incompatible with the original system. There is enough evidence, however, to install the component and try to use it, so the builder proceeds.

The application builder must then integrate the component into the original system; this task may be complicated by syntactic incompatibilities between the interfaces that need to communicate with one another. The builder can either a) modify the original system to overcome these incompatibilities; b) modify the component; or c) introduce a component adaptor [19] or some other *wrapper* between the system and the component. As Hölzle shows, however, there are complications when multiple components must communicate with each other while they are contained within some form of wrapper object [17].

Once all syntactic problems are overcome, however, there will likely still be situations where the functionality or behavior of the component needs to be modified according to the needs of the application builder. Component designers cannot, of course, foresee every possible use of their component, and they cannot respond to every modification request from their users. We need to create mechanisms, therefore, whereby application-builders can easily adapt third-party components without requiring knowledge of the source code.

As more and more third-party components are added to the application - or when an application is constructed entirely from such components - the only solution that will scale is one that minimizes the effort to make modifications to the original application and to adapt the software components.

2.1 Adaptation, Evolution, Customization

The players in this drama are the component designer and the application builder. We make the

distinction between *software evolution*, where component designers modify the software component they designed, and *adaptation*, where an application builder adapts a third-party component for a (possibly radically) different use. If the component designer were requested to adapt a component, the designer would likely select a minimal set of changes because of direct knowledge of the component. The application builder does not have this advantage, nor will the builder be able to acquire this knowledge simply from the source code and documentation. The application builder, thus, needs help to successfully adapt components. We also differentiate adaptation from *customization*; an end-user customized a software component by choosing from a fixed set of options (such as OIA/D [11]). An end-user adapts a software component by writing new code to alter existing functionality or behavior.

2.2 Differences between adapting components and classes

Object-Oriented Design (OOD) embodies the principle of *design for change*, a design principle first stated by Parnas [15] that encourages Software Engineers to modularize code to minimize the impact of future changes. OOD has two mechanisms that serve this purpose. First by designing classes with a public interface and private implementation, a class supports *information hiding*. The class designer can insulate the clients of the class from the internal implementation, which usually changes more frequently than the interface definition. Second, *inheritance* is a mechanism by which an object acquires characteristics from one or more other objects [1]. Inheritance can be classified as *essential*, referring to the inheritance of behavior or an externally visible characteristic, or *incidental*, referring to the inheritance of part, or all, of an underlying implementation of a more general object. Object-oriented designers learn early on that incidental inheritance, done strictly for the purpose of reusing existing code, leads to poor design.

In the Software Architecture literature, inheritance is a modeling vehicle used by various Architectural Description Languages (ADLs), such as ACME [4] to specify when *interface inheritance* occurs (there are exceptions, notably the use of object-oriented typing as seen in [18]). We argue, however, that inheritance should not be used to create new components from parts of old components.

One of the major differences between CBSE and OO is that engineers wishing to adapt an existing object-oriented program must perform the difficult task of understanding (often complex) class hierarchies. There is a tacit assumption with object-oriented technology that the designer of the system and the maintainer/adapter are one and the same. If this is not the case, however, the adapter must determine the set of classes to modify to make the change such that the original integrity is not broken. Often, additional leaf classes are added to a class hierarchy to avoid changing the original class structure when it would have been better to make modifications to existing classes. We seek to find ways for an application builder to adapt a component with only knowledge of its documented interface.

3. Requirements for Component Adaptation Techniques

To set the context for our comparison, consider an application builder that acquires a component *C* from a third-party. The application builder employs an adaptation technique to construct a new component C_A from the original component *C*. The technique may rely only on ad-hoc solutions or it may provide some specific adaptation mechanism. C_A is then used as a component within the target application. If *C* already exists as a component in an application, we classify the situation as *adaptive evolution*. Contrast this with a standard integration problem where the application builder must modify the application so that component *C* can be used as is.

We compiled a list of requirements from [2, 8, 10]. We considered three additional requirements for this paper and have consolidated the total list to a set of eleven possible requirements which we have divided into requirements on *C* and C_A , requirements on the adaptation technique, and requirements on the adaptation mechanism.

3.1 Adapted component C_A and original component *C*

1. Homogeneous - the code that uses C_A should use C_A in the same manner as it would have used *C* ([8], was *transparent* in [2]).
2. Conservative - aspects of *C* there were not adapted should be accessible without explicit effort by C_A (was included as *transparent* in [2]).
3. Ignorant - *C* should have no knowledge of its adaptations (was included as *transparent* in [2]).
4. Identity - *C* should continue to retain its own identity as a separate entity; this eases the way in which future updates of the component will be handled [10].
5. Composable - C_A should itself be open to future adaptations; it should be straightforward to compose together a set of desired adaptations [2].

3.2 Adaptation technique

6. Configurable - the adaptation technique should be able to parameterize and apply a particular adaptation (the *generic part*) to many different components (the *specific part*) [2].
7. Black-box - the adaptation technique should have no knowledge of the internal implementation of *C* [2, 10].
8. Architectural focus - There should be a global description of the architecture of the target application together with a specification of *C* and a modified description of C_A [6]; the specifications of *C* and C_A must be different. This will enable the application builder to specify the adaptation(s) at an architectural level.
9. Framework independent - the adaptation technique must not be dependent upon the component framework to which *C* belongs. For example, the technique must function equally well on COM [12], CORBA [5], and JavaBeans [13] components.

3.3 Adaptation mechanism

10. Embedded - the adaptation mechanism must exist within *C* before *C* can be adapted into C_A [8].
11. Language independent - the adaptation mechanism must not be dependent upon the language used to implement *C* [8]; this requirement also pertains to the adaptation technique.

It may not be possible for an adaptation mechanism to satisfy each requirement, since these requirements are drawn from disparate sources. There is no clear indication on how to prioritize these requirements. Note that some of the requirements in Figure 1 are partly contradictory: **R3** and **R10**, for example. Others are strongly related, such as **R1-R3**. By evaluating component adaptation mechanisms against these requirements, we can determine those requirements that are the most useful.

3.4 Adaptation as a facet of Integration

Incorporating third-party software components will always require integration, but there is not enough emphasis on the necessary adaptation that must take place. Again, we differentiate adaptation from *customization* whereby the customer simply selects from a pre-determined set of options. Some have proposed wrapping or mediation as integration mechanisms, but these only partially satisfy the integration aspects, and do not solve the problems of adaptation.

3.5 Architectural evolution

Figure 1 lists only those approaches that adapt a software component to create a new component. There are several research efforts concerned with *Architectural Evolution*, namely the addition, removal, or replacement of components, connectors, or changes to the configuration of components and connectors. Some examples are ArchStudio [14] and Simplex [16]. There are also different efforts towards creating software systems whose architecture can change dynamically at run-time to adjust as needed to changing circumstances; these are dynamic versions of architectural evolution.

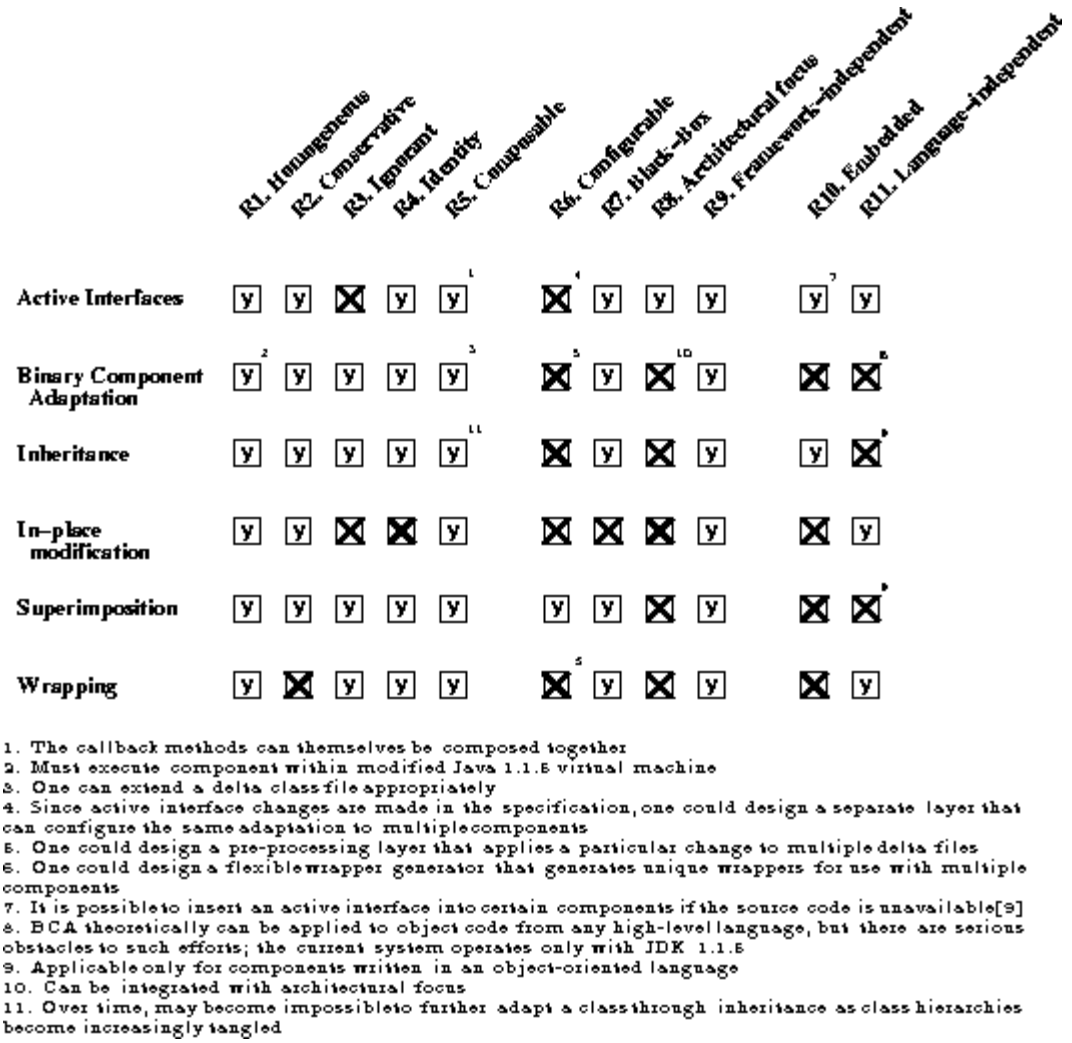


Figure 1: Comparison matrix

4. Discussion

The comparison matrix in Figure 1 reveals various correlations between the requirements and mechanisms. There is strong agreement that requirements **R1,R5,R9** are suitable for adaptation mechanisms. This reflects, perhaps, the fact that these requirements relate to structural issues. Requirements **R6** and **R8** only have one proponent each, namely Superimposition [2] and Active Interfaces [8], but this is simply a way these techniques differentiate themselves from others in the literature.

The sharpest division on these requirements (where at least two techniques vote positive and two vote negative) are embedded (**R10**) and language-independence (**R11**). Note that these requirements both refer to the type of adaptation mechanism employed by the adaptation technique. Most component designers select a programming language with only a passing attention to future adaptation needs. Object-oriented programming languages, however,

automatically enable adaptation because inheritance is built-in. The Active Interface approach [8] only requires small hooks embedded into a component and thus is a viable adaptation technique. Currently, component implementation is driven by the selection of a particular component framework, such as JavaBeans [13], Component Object Model (COM) [12], or CORBA [5]. A component belonging to one of these frameworks must embed the appropriate mechanisms to belong to the framework, so in a sense embedding need not be a controversial topic.

When considering the adaptation techniques themselves, in-place modification clearly fails; with small differences, however, the other techniques are more related than one might realize at first. We have carried out a small experiment, described in [9], in evaluating the use of each of these techniques (except Superimposition) in solving an adaptation problem. We are currently designing more rigorous experiments to understand how to better aid application builders when they need to adaptation existing components.

5. Conclusion

This evaluation survey provides an overview of existing component adaptation techniques and provides a good starting point for discussing the nature of component adaptation mechanisms. This material belongs in various sections of the proposed Strawman outline for the workshop. Under the Technology supporting CBSE (Section 3), reusable components must be discussed within the framework of how application builders will adapt them. Integration technologies should not be limited to Run Time support; rather it should include such static mechanisms as discussed in this paper. Finally, from a philosophical perspective, it is important to differentiate software reuse (which traditionally has been a means of reusing functional code libraries or frameworks) from reusable components (which brings in the notion of adapting behavior).

References

1 Edward V. Berard. *Essays on Object-Oriented Software Engineering.* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

2 Jan Bosch. Superimposition: A component adaptation technique. Technical Report TR, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, September 1997.

3 Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37-46, September 1990.

4 David Garlan, Robert T. Monroe, and David Wile. ACME: An architectural description interchange language. In *1997 CASCON Conference*, pages 169-183, Toronto, Ontario, November 1997.

5 Object Management Group. CORBA standard. Internet site (`http://www.omg.org`).

6 George T. Heineman. Adaptation and Software Architecture. In *3rd International*

Workshop on Software Architecture, pages 61-64, Orlando, FL, November 1998.

7 George T. Heineman. Composing software systems from adaptable software components. In *DARPA/OMG Workshop on Compositional Software Architectures*, Monterey, CA, January 1998. `http://www.objs.com/workshops/ws9801/report.html`.

8 George T. Heineman. A Model for Designing Adaptable Software Components. In *22nd Annual International Computer Software and Applications Conference*, pages 121-127, Vienna, Austria, August 1998.

9 George T. Heineman and Helgo Ohlenbusch. An Evaluation of Component Adaptation Techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

10 Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California, Santa Barbara, December 1997.

11 Gregor Kiczales, John Lamping, Cristina Lopes, Chris Maeda, Anurag Mendherkar, and Gail Murphy. Open Implementation Design Guidelines. In *19th International Conference on Software Engineering*, pages 481-490, May 1997.

12 Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification: Draft Version 0.9, October 24, 1995.
Internet publication (`http://www.objs.com/workshops/ws9801/report.html`).

13 Sun Microsystems, Inc. JavaBeans 1.0 API Specification.
Internet site (`http://www.javasoft.com/beans`), December 4, 1996.

14 P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering*, Kyoto, Japan, April 1998.

15 David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(6):310-320, March 1979.

16 L. Sha, R. Rajkumar, and M. Gagliardi. Evolving dependable real-time systems. In *IEEE Aerospace Applications Conference*, pages 335-346, New York, NY, 1996.

17 Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In O. Nierstrasz, editor, *ECOOP '93 Conference Proceedings*, LNCS 707, pages 36-56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

18 Ian Welch and Robert Stroud. Adaptation of connectors in software architectures. In *Third International Workshop on Component-Oriented Programming (WCOP'98)*, Brussels, Belgium, July 1998.

19 Daniel M. Yellin and Robert E. Strom. Protocol Specification and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292-333, March 1997.

Practical Software Engineering Support for Component-Based Control Systems.

Francois Bronsard, Gilberto Matos, Dilip Soni
Siemens Corporate Research
755 College rd. East
Princeton, NJ 08540

INTRODUCTION

This position paper presents our current work on development tools for component-based systems. Although some of our efforts are specificly aimed at the field of control systems, we believe that the development tools suggested here can be generalized to other fields in which component-based systems are becoming common.

The field of control systems is experiencing a shift from stand-alone applications to distributed component-based systems. In this new environment, a major challenge for control system developers will be the correct and effective integration of COTS components and custom-made components. Our goal is to produce tools to help developers verify whether their own components make correct and effective use of the environment provided by the COTS components and the underlying infrastructure.

COMPONENT-BASED CONTROL SYSTEMS

Traditionally, small and medium sized control systems are stand-alone programs residing on Programmable Logic Controllers. However, current trends render this simple architecture obsolete. First, at the bottom layer of this architecture, the device layer, the devices controlled by the PLCs are becoming "smarter" and able to take over part of the control logic previously residing on the PLC. Making use of these capabilities would improve control systems. Similarly, in modern systems the top layer of this architecture, the presentation layer, does not have as its sole client a single human-machine interface running on a monitor residing in a centralized control room. Today, many business applications want to access the information present on the PLC and even to exert some high-level control capabilities. Finally, the industry would like more flexibility in setting up and evolving the plants being controlled. One would want to be able to replace a crane in a plant by a newer, improved model without needing a completely new control system. By encapsulating the control of the crane in a separate component, such flexibility becomes possible.

To address these trends, control systems are moving toward a component-based architecture with systems built on top of a standard distributed-enabled execution environment, namely Microsoft DCOM. Thus, we see a future where most components needed for a control system will be provided by various commercial entities. The manufacturers of the devices used in a plant will provide software components to control these devices. Another set of components will be provided by control and optimization specialists. Finally, the plant engineers will add custom

components to address plant-specific issues and to provide the high-level control mechanisms. In this context, the major challenge faced by control system developers is to insure that these components will work together correctly and effectively.

THE INTEGRATION PROBLEM

Our concern is to integrate the COTS components with the custom-made components, and to do so rapidly and reliably. However, the barriers to integration are numerous. Often an in-depth knowledge of the components being used and of the infrastructure supporting the system is necessary to build a truly optimal system. Therefore, the long learning curve for complex components and infrastructure impedes their adoption. To address this problem, we are proposing a tool able to check *usage rules*. We envision that complex components would be accompanied by a collection of usage rules describing how best to use the component. Our tool would then test whether the custom-made components adhere to these rules.

As components and the infrastructure become more complex, the interactions between them also become complex. This leads to a need to document these interactions, and to test whether the client objects of these components use the components correctly. The second tool that we are developing is designed to help test such complex interactions. We envision that components will be accompanied by a description of how the component interacts with its environment. Our tool would generate a run-time mechanism to monitor how the rest of the system interacts with that component and check if this interaction satisfies the specification of the component.

CHECKING USAGE RULES

The COTS components that will form the core of a control system, as well as the underlying infrastructure, form a collection of complex frameworks that plant engineers must master to build a performing control system. However, using complex frameworks is challenging even for experienced developers. Without knowledge of the intricacies of each component and of the infrastructure, many common programming practices either fail to take advantage of the framework, or worse, introduce errors. For widely used frameworks, this leads to the creation of user groups and the publication of FAQs, or even books, dedicated to cataloging "usage tips" for using the framework. For example, to use the Microsoft COM infrastructure more effectively, one might read the recent book "Effective COM: 50 ways to improve your COM and MTS-based applications" by Box, Brown, Ewald, and Sells.

Many of the tips for using a complex component or a framework can be captured by programming guidelines, or usage rules, that can be checked automatically. Guidelines identify those code patterns that are inefficient or incompatible with the framework, and suggest alternative design choices that make better use of it. Automatic guideline checking improves software quality and performance by finding programming errors or inefficiencies. It also reduces the amount of effort needed to develop a system since full mastery of the framework is not required.

We have developed a generic Code Inspector which provides a powerful facility for ensuring that source code adheres to programming guidelines and usage rules. This code inspector has been

used in projects to detect common mistakes and test adherence to project specific guidelines. For example, a version of the code inspector targeted to C/C++ programmers checks guidelines such as "Do not use delete[] to delete a non-array and delete to delete an array." This C/C++ inspector currently checks for 150 guidelines.

In our current project, we are combining the Code Inspector with usage rules derived from Box et al.'s book to develop a small COM Inspector to help developers produce more effective COM programs. For example, one of the guidelines explains how to make use of the facilities of the COM layer to protect clients and servers against communication failure: for a client, it is recommended always to check the return value of remote procedure calls since this value will indicate communication failure; for a server, it is recommended to use the worker pattern since this enables one to use the COM garbage collector to be warned of client or communication failure.

The COM inspector demonstrates the feasibility of building a code inspector targeted to a specific complex framework. We propose to generalize this approach and suggest that components be accompanied by collections of usage rules that could be checked automatically. This would significantly ease the use of complex components.

TESTING

It is often hard to debug a complex application to find out what caused an error, since the application consists of a number of concurrent processes or threads. One way to quickly identify when something goes wrong is to add runtime verification code to the components and detect anomalies as they happen. It is rarely acceptable to use this code in real products due to its overhead, and it is even less practical to require designers to implement and maintain multiple versions of their components. Fortunately, code that verifies expected behaviors can be automatically generated and integrated with the original components, creating temporary objects that log and analyze some aspects of system interaction to detect errors.

The component designer can specify its expectations in the form of a finite state machine whose transitions are enabled by the occurrence of specific events, such as method invocations. This state machine contains specific states that are only reachable as a result of interaction patterns, usually incorrect ones. Logging and monitoring of a component's environment allows the integrators to identify the first time an unexpected event affects a given component, thus guiding them faster toward the original cause of the error. Interaction logging provides information on the coverage of the expected behaviors, and incomplete coverage indicates that a component is not being used to its full potential, suggesting possible efficiency improvements.

We are developing a tool that accepts a UML specification of a component with a state machine description of its environment expectations, and produces monitoring code that can be used as a wrapper for the original component without modifying its functionality. Our tool is specifically directed at COM-based systems, and relies on the interception of interfaces provided by a given server component. The designer of a COTS server component should provide a description of its environment expectations, either in the form of monitoring code or UML state diagrams that can be used to generate the code. The instrumented component monitors the incoming events and

signals to the user if the assumptions are violated. At the end of the execution, the log can show whether all the states and transitions of the expected behavior were exercised by the test drivers.

We are working on broadening the range of events within COM that can be used to monitor expected system behaviors, e.g. by capturing outgoing interfaces and monitoring dynamic object creation. We are also studying the application of this technology to more complex interaction patterns, such as groups of components involved in a use case. We envision that tools like this will be used by COTS components developers to create executable objects that can monitor their environment when necessary, and cooperate with other similar components to verify the correctness of their execution.

CBSE in the Realm of Computing / Information System Life Cycle

Lana Kuzmanov , kuzmanov@sympatico.ca

A position paper for Second International Workshop on Component-Based Software Engineering in the category "Practices for Adopting CBSE".

1. Introduction

In the never-ending quest for efficient and high quality solutions to the IT problems that industry organizations experience in a wide range from strategic to day-to-day, they tend to gladly embrace new techniques. For its known features and benefits, CBSE falls into the category of emerging, promising approaches. However, in the effort to position itself for its adoption, organization inevitably faces questions such as: How to fit CBSE into its business process life cycle? How should the CBSE process itself work? How to enable and support CBSE process execution? How to approach CBSE process implementation? This paper is addressing the above questions.

2. Positioning the CBSE in an organization's process life cycle

Position Statement 1. An organization must position the CBSE process within the framework of its business process life cycle by defining its relations with other processes (Figure 1). The rationale for this is implied by the fact that success of an organization is directly proportional to the quality of business processes design and management of those processes. For example, the quality of results (products and/or services) and the efficiency of execution (least time with least resources) are crucial measurement parameters of a particular process. Applying the approach where activities are executed concurrently and providing utilities for information assets reuse throughout the process, contribute to high rating of these parameters.

Process Architecture.

1.) Business Process (BP) life cycle: *Scope / Problem Context*. Includes business drivers definition, BP modeling, BP planning, BP management, and BP evolution. Results of business drivers definition activities are used in the architecture life cycle at the strategic design level. Relevant deliverables of BP modeling activities for our consideration are data requirements, computing/ information services requirements and computing/ communication requirements. These requirements represent a base (necessary input) for Component-Based system design as a part of architecture life cycle process.

2.) Architecture life cycle: *Solution / Design Context*. Component of Computing / Information System life cycle. System engineering activities, such as component-based analysis, design, and development (i.e. CBSE), are embodied within this context. Detailed in Section3 (Figure 2).

3.) System life cycle: *Deployment/Operations Context*. Component of Computing / Information System life cycle. This is the context of actual system operations comprising the activities of system management, component (data, software, computing and communication) distribution, production implementation, system administration and support. Detailed in Section 3, Figure 2.

4.) Supporting systems life cycle: *Support / Enabling Context*. Our scope (this paper) only covers Computing / Information Models Process Life Cycle details of which are addressed in Section 4.

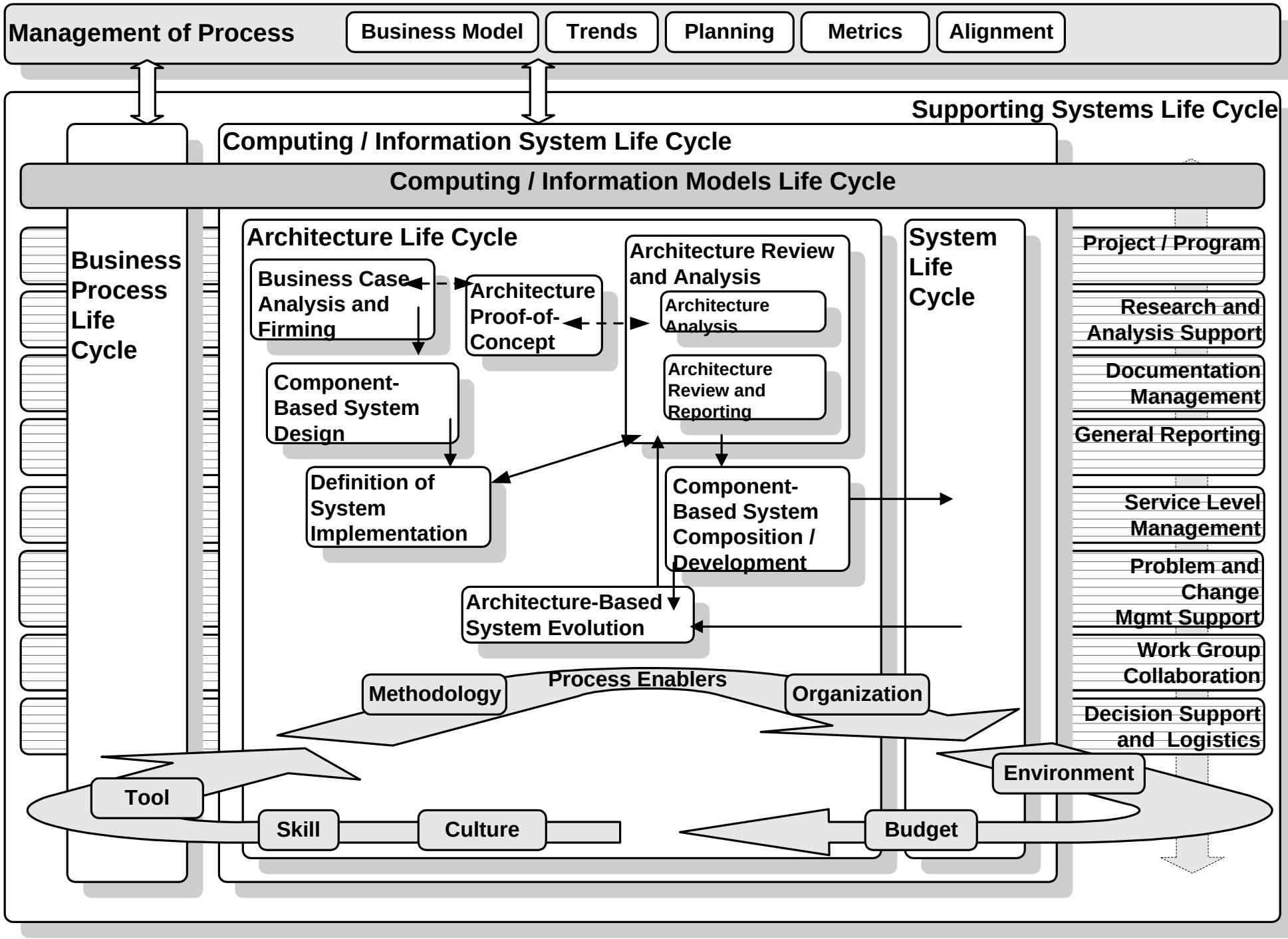
Process Enablers.

In order to execute and deliver product or service, process requires support of enabling components. These components are methodologies, organization, environment, budget, culture, skills/competence and tools and are called process enablers.

Management of Process.

Business process requires management in all of its segments. It is critical that the management of process be synchronized across Business, Computing / Information System and Supporting Systems. Management of process includes models, trends, planning, metrics and alignment.

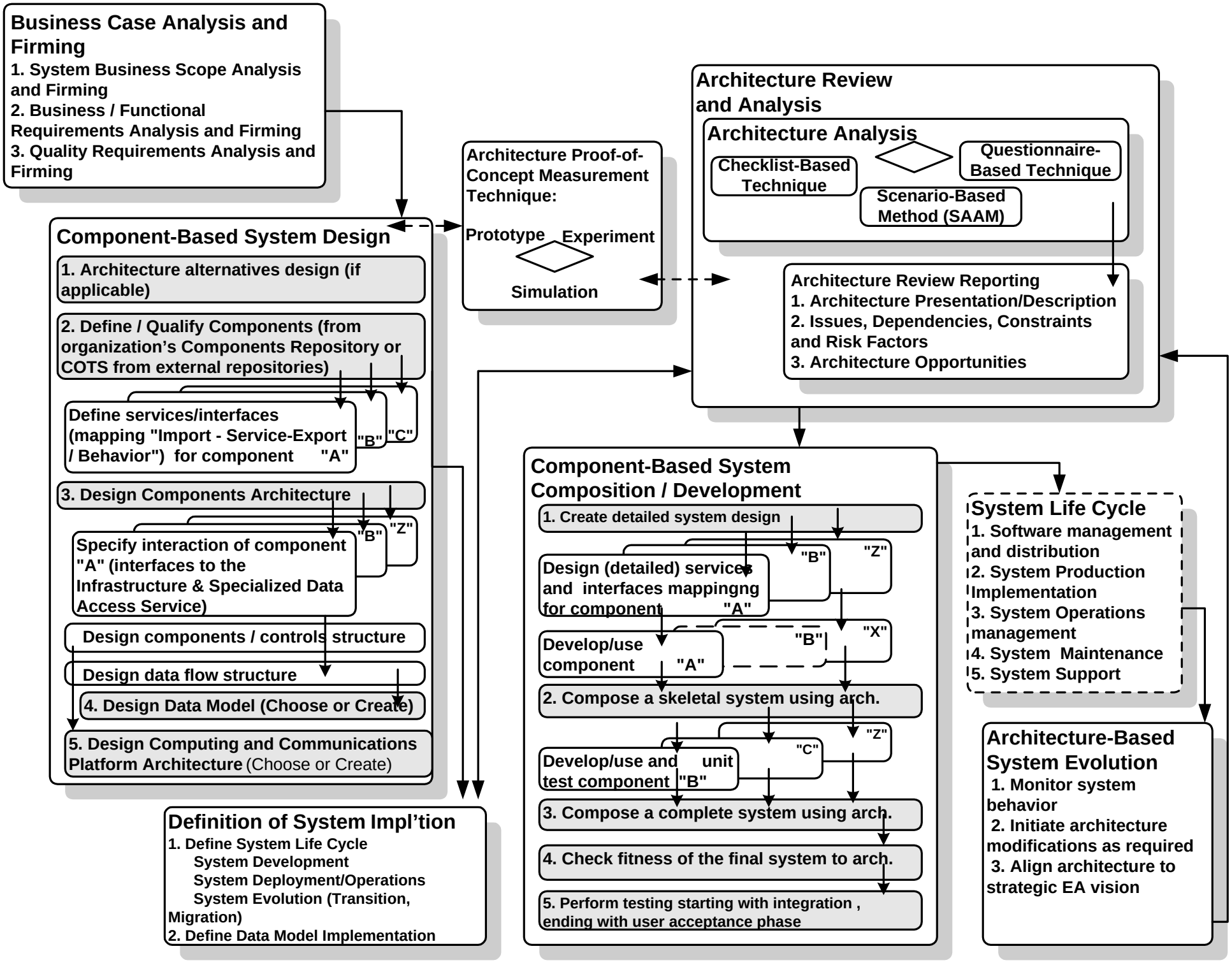
Figure 1. System's Process Life Cycle ***(figure appears on next page)***


Management of Process
Business Model
Trends
Planning
Metrics
Alignment
Supporting Systems Life Cycle
Computing / Information System Life Cycle
Computing / Information Models Life Cycle
Business Process Life Cycle
Architecture Life Cycle
Business Case Analysis and Firming
Architecture Proof-of-Concept
Architecture Review and Analysis
Architecture Analysis
Architecture Review and Reporting
Component-Based System Design
Definition of System Implementation
Component-Based System Composition / Development
Architecture-Based System Evolution
System Life Cycle
Project / Program
Research and Analysis Support
Documentation Management
General Reporting
Service Level Management
Problem and Change Mgmt Support
Work Group Collaboration
Decision Support and Logistics
Process Enablers
Methodology
Organization
Tool
Skill
Culture
Budget
Environment

3. Defining CBSE Process

Position Statement 2. An organization must adopt (choose, choose and customize or create) CBSE process that suits its needs (Figure 2) focusing on the fulfilling the following: 1.) Applying an architecture-centered approach [BCK]; 2.) Compromising between requiring rigorous procedures (that introduce process complexity, require over-documenting, make achievement of process phase exit criteria difficult, etc.) and manageable, user-friendly procedures; 3.) Requiring concurrent execution of activities throughout analysis, design and development processes; 4.) Standardizing process supporting and enabling framework (computing / information models base, toolkit, etc.).

Figure 2. CBSE Process Diagram ***(figure appears on next page))***


Business Case Analysis and Firming
1. System Business Scope Analysis and Firming
2. Business / Functional Requirements Analysis and Firming
3. Quality Requirements Analysis and Firming
Architecture Proof-of-Concept Measurement Technique:
Prototype
Experiment
Simulation
Architecture Review and Analysis
Architecture Analysis
Checklist-Based Technique
Questionnaire-Based Technique
Scenario-Based Method (SAAM)
Architecture Review Reporting
1. Architecture Presentation/Description
2. Issues, Dependencies, Constraints and Risk Factors
3. Architecture Opportunities
Component-Based System Design
1. Architecture alternatives design (if applicable)
2. Define / Qualify Components (from organization's Components Repository or COTS from external repositories)
Define services/interfaces (mapping "Import - Service-Export / Behavior") for component "A"
"B"
"C"
3. Design Components Architecture
Specify interaction of component "A" (interfaces to the Infrastructure & Specialized Data Access Service)
"B"
"Z"
Design components / controls structure
Design data flow structure
4. Design Data Model (Choose or Create)
5. Design Computing and Communications Platform Architecture (Choose or Create)
Component-Based System Composition / Development
1. Create detailed system design
Design (detailed) services and interfaces mappingng for component "A"
"B"
"Z"
Develop/use component "A"
"B"
"X"
2. Compose a skeletal system using arch.
Develop/use and unit test component "B"
"C"
"Z"
3. Compose a complete system using arch.
4. Check fitness of the final system to arch.
5. Perform testing starting with integration , ending with user acceptance phase
System Life Cycle
1. Software management and distribution
2. System Production Implementation
3. System Operations management
4. System Maintenance
5. System Support
Architecture-Based System Evolution
1. Monitor system behavior
2. Initiate architecture modifications as required
3. Align architecture to strategic EA vision
Definition of System Impl'tion
1. Define System Life Cycle
System Development
System Deployment/Operations
System Evolution (Transition, Migration)
2. Define Data Model Implementation

4. Managing the "Building Blocks"

Position Statement 3. An organization should implement Computing / Information Models life cycle approach as supporting process to CBSE (Figure 3). The process is data-centered i.e. its main objective is managing data about organization's information assets (also enabling links to information about COTS). The underlying database is comprised of repositories containing "building blocks" for requirements engineering (Business Models Repository), component-based designing and developing (Architecture Models Repository), and operating and managing system that performs computing and/or information processing (Operational Models Repository). This is achieved by providing management of information about components of corresponding nature available for reuse in all relevant contexts:

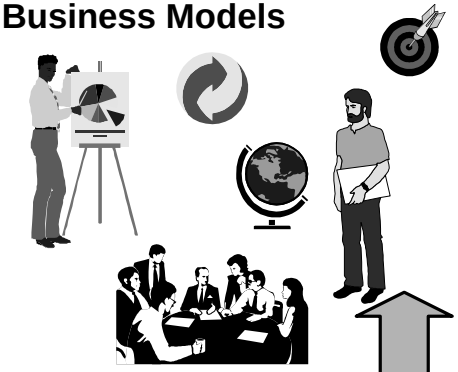
1. Business Models Repositories. Data, Services and Computing and Communication Requirements Repositories, provide necessary support for those processes completion in efficient manner. The principle is: "Specification of problem defined once, reused whenever alike problem identified."

2. Architecture Models Repositories. Data Structures, Components / Applications and Computing and Communications Platforms Repositories are used to retrieve, analyze and qualify components for reuse in particular system solution specification. The principle is: "Solution for the problem developed once, reused for alike problem."

3. Operational Models Repositories. Data Structures Deployment, Component / Application Operation, and Computing & Communication Operation Repositories, provide necessary information about various components configurations and setups. The principle is: " Configured and setup onetime, operated and managed anytime."

Figure 3. Information Models Base ***(figure appears on next page)***

Business
Scope / Problem Context:
Business Models

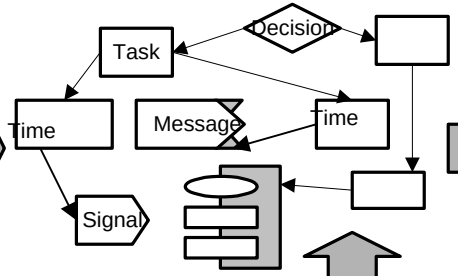


Data Requirements

Service Requirements

Computing & Communication Requirements

Architecture
Solution / Design Context:
Architecture Models

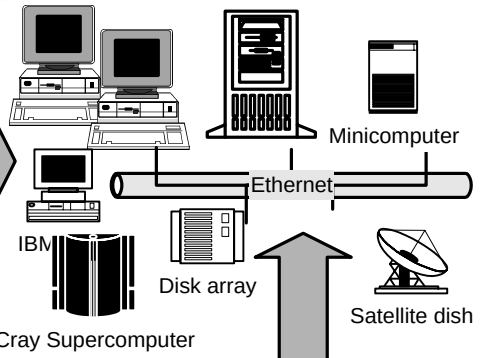


Data Structure (Logical and Physical)

Component / Application Structure

Computing & Communication Structure

System
Deployment/Operation Context:
Operational Models



Data Structure Deployment

Component / Application Operation

Computing & Communication Operation

Computing & Information Models Base

Business Models Repository

- **Data Requirements Repository**
- **Service Requirements Repository**
- **Computing & Communication Requirements Repository**

Architecture Models Repository

- **Data Structures Repository**
- **Components / Applications Repository**
- **Computing & Communication Platforms Repository**

Operational Models Repository

- **Data Structures Deployment Repository**
- **Component / Application Operation Repository**
- **Computing & Communication Operation Repository**

5. Components-Based Architecture

Position Statement 4. An organization must develop target component-based architecture of computing / information system (Figure4). The Component-Based architecture is natural in designing computing / information services for an organization. This is due to the fact that Component-Based architecture is aimed at resolving problems typical for legacy (current) computing / information systems such as: 1.) Specialization of component services and interfaces standardization not implemented, 2.) Separation of component specification from component implementation not present, and 3.) Component interactions implemented through non-standard interfaces and in a point-to-point mode (creating a look of components integration "spaghetti").

Figure 4. Component-Based Architecture of an Organization ***(figure appears on next page)***

The solutions can be achieved by: 1.) Strict specialization of component services and its interfaces standardization allowing clustering of components by their nature and services providing for clear separation of components concerns and supports clear definitions of component interactions within the architecture through the standard interfaces; 2.) Separation of component specification from component implementation for all newly integrated components allowing component behavior to be described independently of its implementation and supports the possibility of multiple alternate component implementations for the same specification., and 3.) Building a common "plumbing infrastructure" i.e. Interface Infrastructure for integration of components providing for components integration by the common interface enabling the integration at the higher level than DCOM [DCOM], CORBA [OMG] or RMI [RMI].

The Components (Building Blocks) for a typical organization (Figure 4) fall into the following four categories determined by their service specialization: 1.) Business Domain Service components (business applications); 2.) Client Business & Presentation Service components (human / device and computing systems interactions services); 3.) Data Stores (data structures and data access), and 4.) Infrastructure & Specialized Data Access Services components.

First three categories of components will connect into the enterprise architecture through the common **Infrastructure Interface** for integration called Infrastructure Services & Specialized Data Access Services which we consider critical for the success of CBSE. This infrastructure is providing for components interactions at all layers from the basic network to the application through the high level, business-oriented, user-friendly interface. This is accomplished through the following services:

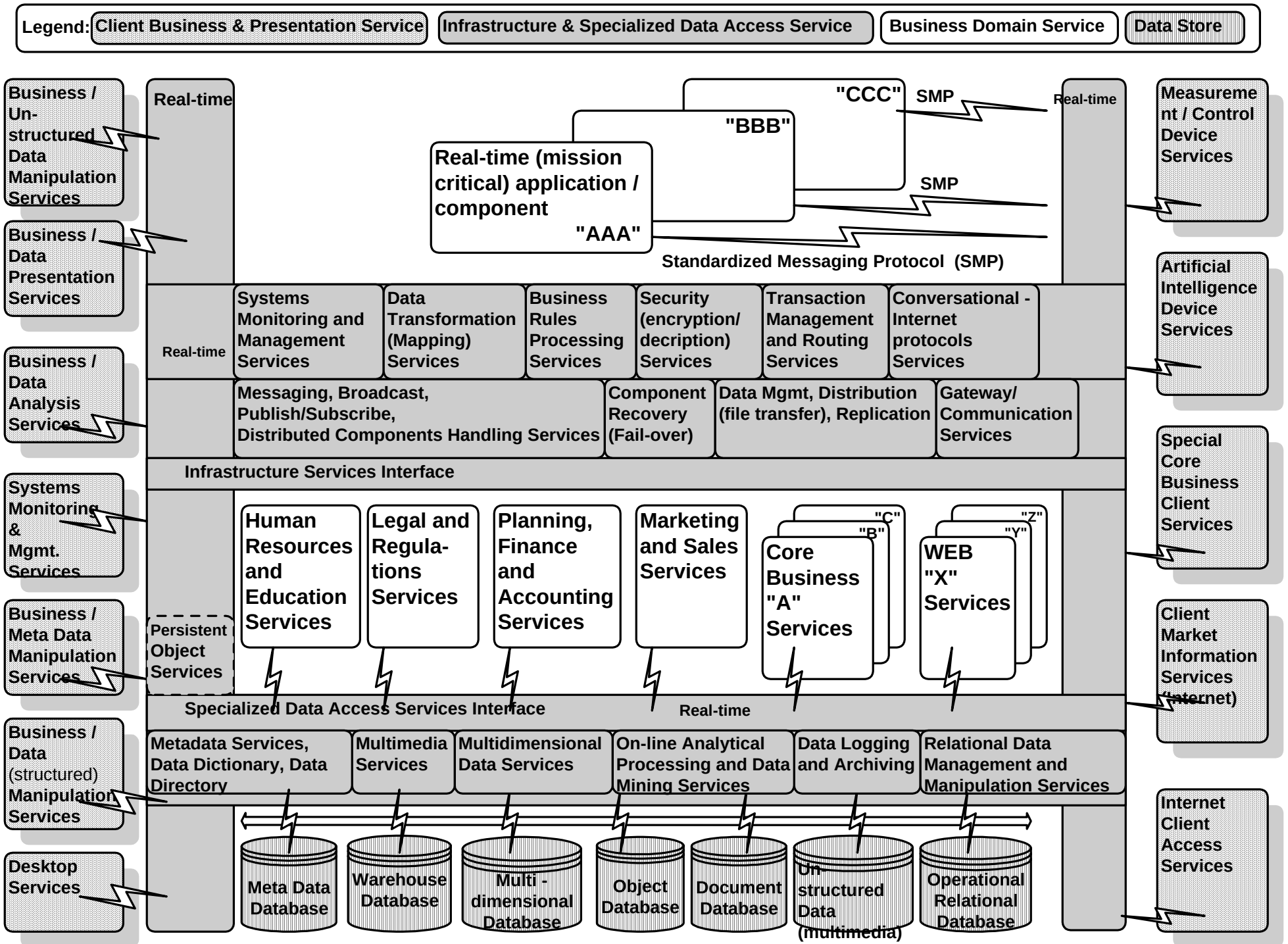
1.Messages, Broadcast, Publish/Subscribe, Distributed Components handling, and Conversational - Internet protocols handling.

2.Business Rules and Events processing. Handle logic of sharing information between multiple components / applications. Support definition of rules and events, and manages associated actions.

3.Data Streams and Messages Transformation and Routing services. Enables data streams and messages mapping, formatting and routing in dynamic and adaptable manner by supporting configuration of transformation and routing parameters based on the format and content of the data stream or message. E.g. a message from a specific component in one format can be routed to another and arrive in the form and context that is understandable and actionable.

4. Components and Transactions services including: Transaction management and monitoring, Security (encryption/decryption), Component fault detection, error handling and recovery, Communications handling.

Specialized Data Access Services is a structure comprising of components providing for the following: Meta-data services, Data Dictionary, Data Directory and Component Repository services, Relational data management, Data Logging & Archiving, Data management, data distribution and data replication, Multimedia management, Multidimensional data handling, On-line analytical processing and data mining, and Persistent objects handling.


Legend:
Client Business & Presentation Service
Infrastructure & Specialized Data Access Service
Business Domain Service
Data Store
Business / Un-structured Data Manipulation Services
Business / Data Presentation Services
Business / Data Analysis Services
Systems Monitoring & Mgmt. Services
Business / Meta Data Manipulation Services
Business / Data (structured) Manipulation Services
Desktop Services
Real-time
Real-time
Real-time
Real-time
Persistent Object Services
"CCC"
"BBB"
"AAA"
SMP
SMP
Real-time (mission critical) application / component
Standardized Messaging Protocol (SMP)
Systems Monitoring and Management Services
Data Transformation (Mapping) Services
Business Rules Processing Services
Security (encryption/ decription) Services
Transaction Management and Routing Services
Conversational - Internet protocols Services
Messaging, Broadcast, Publish/Subscribe, Distributed Components Handling Services
Component Recovery (Fail-over)
Data Mgmt, Distribution (file transfer), Replication
Gateway/ Communication Services
Infrastructure Services Interface
Human Resources and Education Services
Legal and Regula-tions Services
Planning, Finance and Accounting Services
Marketing and Sales Services
"C"
"B"
Core Business "A" Services
"Z"
"Y"
WEB "X" Services
Specialized Data Access Services Interface
Real-time
Metadata Services, Data Dictionary, Data Directory
Multimedia Services
Multidimensional Data Services
On-line Analytical Processing and Data Mining Services
Data Logging and Archiving
Relational Data Management and Manipulation Services
Meta Data Database
Warehouse Database
Multi - dimensional Database
Object Database
Document Database
Un-structured Data (multimedia)
Operational Relational Database
Measurement / Control Device Services
Artificial Intelligence Device Services
Special Core Business Client Services
Client Market Information Services (internet)
Internet Client Access Services

6. Summary and Conclusions

This paper has explored some of the questions related to CBSE in the context of its adoption and implementation in a typical business organization. The ideas expressed have originated from our belief that success of the CBSE approach in an organization is directly proportional to fulfilling the following prerequisites:

1. Position the CBSE within the system's process life cycle which is based on relevant context areas covering business processes, architecture processes, operational computing / information system processes and supporting systems processes.

2. Define framework for the CBSE process which is architecture-centered, compromises between insisting on rigorous procedures and using manageable, user-friendly procedures, is based on concurrent execution of analysis, design and development activities and has standardized framework for process support and enabling (computing / information models base, toolkit, etc.).

3. Adopt the Computing / Information Models life cycle approach using Computing / Information Models Base repositories aimed at supporting Business, Architecture and System Life Cycle processes in delivering their products and/or services.

4. Develop target component-based architecture of computing / information services based on component services specialization and interfaces standardization, applying separation of component's architecture and component's implementation, and implementing common interface (plumbing) infrastructure.

References

[BCK] Bass L., Clements P., Kazman R., Software Architecture in Practice, Addison Wesley, 1998

[DCOM] http://www.microsoft.com/oledev

[OMG] http://www.omg.org

[RMI] http://java.sun.com

Component Based Software Engineering: A Broad Based Model is Needed

Allen Parrish (parrish@cs.ua.edu)
Brandon Dixon (dixon@cs.ua.edu)
David Hale (dhale@alston.cba.ua.edu)

Department of Computer Science
Area of Management Information Systems
The University of Alabama
Tuscaloosa, AL 35487

February 1999

Abstract

Over the past few years, a number of different models of component-based software engineering have been proposed and discussed. We argue that many of these models are too narrow and/or informal, and that a broader and more precise foundation is needed for CBSE.

1. INTRODUCTION

Component-based software engineering (CBSE) has existed in one form or another for a number of years. The idea of constructing modular software has long been recognized as advantageous within the software community, even dating back to the early days of FORTRAN programming with subroutines and libraries serving as "components." Work by Booch [2] and Meyer [5] in the 1980's was generally regarded as seminal in the advancement of ideas regarding the fundamental nature of components, particularly with regard to low-level structural properties of the components. More recent work [1,7,8,10,12] extended these ideas along various dimensions, including the introduction of formal specifications into component frameworks, the development of new paradigms for data movement, and the development of improved design guidelines for what constitutes a good component that is both efficient and independently verifiable.

Over the past few years, advances in enterprise computing and client-server communities have generated renewed interest in the concept of CBSE, bringing the terms "component" and "component engineering" into widespread use within the software community. (Publications such as *Component Strategies* and the recent CBSE special issue of *IEEE Software* [3] represent the current popular view.) However, most current popular references to these terms are in a much different context than that described above. In particular, current popular CBSE references are to technologies such as COM, CORBA and JavaBeans that support the encapsulation of so-called "binary" components. Such CBSE references and the associated technologies are also linked to the support of distributed object computing, where the notion of a component seems to be linked (perhaps equivalent to) an encapsulated object-oriented software unit that is deployed within a distributed architecture.

We feel that there is a tacit impression given in much of the current dialog that the modern notion of components is based upon a brand new set of ideas and concepts. For example, the

following quote appears in [4]: "Components are now an evolution beyond objects, incorporating all the best aspects of objects, adding important new engineering concepts such as separation of interface and implementation, and enabling easier development through provision of rich runtime services." The ideas listed here (e.g., separation of interface and implementation) have been an important part of object-oriented development from the very beginning. A slightly different quote appears in [11]: "Objects and components come from the same family. Like siblings, they squabble about their differences, but have much more in common than they are willing to admit." This quote implies a popular view that places objects and components in totally distinct categories, a view that this author is trying to dispell. However, the fact that they are in completely different categories to begin with is disturbing, as we believe that the relationships are strong and obvious.

The current popular view of components appears to be summed up by Szyperski [9]: "Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system." In particular, the fact that software components are *binary* units appears to be widely assumed in much of the current popular dialog. We do not object to this view *per se*, but we simply claim that it is inappropriate as a *starting point* for proving a foundation for CBSE. Such a starting point seems to suggest that CBSE originated with the advent of binary component technologies (e.g., COM), and that previous work in the areas of source code components and object-oriented technologies is only weakly related to the modern notion of CBSE. In contrast, our position is that the foundation of CBSE should rest on a generic model of components that allows basic CBSE concepts to be expressed as broadly as possible, and results in the field to be applied as broadly as possible. If such a model is designed properly, then constraints can be added at the appropriate time to obtain more specialized notions of components.

2. A FUNDAMENTALS PERSPECTIVE

We propose that the underlying definition of component be approached as a kind of "conceptual theory," by proposing a generic, formal definition of component to use as a starting point. As an example of the beginning of such a theory, we say that a *component* is a software artifact consisting of three parts: a *service interface*, a *client interface* and an *implementation*. Roughly speaking, the service interface consists of the services that the component exports to the rest of the world; the client interface consists of those services used by this component exported from other components, and the implementation is the code necessary for the component to execute its intended functionality. To enforce the idea that a component must interact with other *software*, we might also want to include a property that the service interface *or* the client interface might be empty, but not both.

To borrow from the electronics domain, we say that a service interface consists of one or more *receptacles*, while a client interface consists of one or more *connections.* A connection is *plug compatible* with a particular receptacle if there is formal consistency between the two. The nature of the formal consistency depends on the types of components and types of interfaces.

As with any theory, there are necessarily a number of undefined terms. The idea though is to permit any realization satisfying the formal structure to be treated as a component model. For example, a simple library of C functions may be viewed as a component under this model. In this

case, the service interface is the collection of function prototypes, the client interface is the set of external functions called by functions in the library, and the implementation is the aggregate of all of the function implementations.

Other component realizations of our model include:

- A C++ class.
- A cluster of C++ classes, with a particular class serving as the exported interface, and the other classes functioning as part of the implementation.
- A Windows DLL.
- A COM object.
- A Java Bean.
- A CORBA-based server object.
- A Unix shell program (i.e., functioning within a pipe-and-filter style architecture).

In each case, it is a straightforward process to identify and justify all of the elements in our generic model. Effectively, our model seems to treat almost any software artifact with one or more identifiable interfaces as a component. This may appear to suggest that our model is too generic to be of any practical use. But it should be recognized that we are only proposing a possible "lowest-level" foundation for CBSE. That is, our first principles-based argument is that software modules with connections to other software modules are, at some level of abstraction, components. In the most general sense, all such modules share some very general properties. Thus, when thinking at this level, every time we invent some new kind of software artifact, we are not necessarily inventing something *totally* new. In our opinion, this is a very important concept to clarify.

Once this generic foundation has been established, restrictions may be added. For example, one could simply restrict the universe of components to those components in binary form. Indeed, it might even be appropriate to argue that the "component" label should only be applied to binary artifacts, as is the current popular usage of the term. If this were the case, by *starting* with the less restrictive model and then adding the binary requirement, the strength of the connection between the restricted notion of components and other non-binary artifacts (such as object-oriented classes) would be more explicit. In this case, we might even formally refer to the more general definition as a "module" and the restricted form as "component." Thus, "component" is a type of "module", sharing some generic properties with "modules" that are *not* "components." By identifying what these properties are, the inherited legacies from other technologies are more explicit, and it may be easier to distinguish truly new concepts from reincarnations of old ones.

This model is definitely work in progress. Our objective was to simply outline a possible direction for the solution of an extremely difficult problem. We believe that an incremental approach, where minimal, first-principles properties are first identified and then restrictions are progressively added, is the only way to arrive at a clear, unambiguous foundation for what CBSE should be about. We also note that this view is similar to a recent article by Betrand Meyer [6], who believes in a strong linkage between object-oriented technology and components, and proposes various dimensions for characterizing components. We believe that an extension of his framework into a full-scale, multi-dimensional characterization is badly needed (perhaps

supported by a generic, minimalist foundation such as the one described here).

REFERENCES

[1] Batory, D. and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398.

[2] Booch, G., *Software Components with Ada* , Benjamin-Cummings, 1987.

[3] Brown, A. and K. Walinau, "The Current State of CBSE," *IEEE Software*, vol. 15, no. October 1998, pp.37-46.

[4] Kora, D. "Build vs. Buy – Maximizing the Potential of Components," *Component Strategies*, vol. 1, no. 1, July 1998, pp. 22-35.

[5] Meyer, B., *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall International, 1994.

[6] Meyer, B., "On to Components," *Computer*, vol. 32, no. 1, January 1999, pp. 139-140.

[7] Sitaraman, M. and B. Weide, eds., "Special Feature: Component-Based Software Using RESOLVE," *Software Engineering Notes* , vol. 19, no. 4, October 1994, pp. 21-67.

[8] Weide, B., W. Ogden and S. Zweben, "Reusable Software Components," *Advances in Computers* , vol. 33, M.C. Yovits, ed., Academic Press, 1991, pp. 1-65.

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

[10] Smargdakis, Y. and D. Batory, "Implementing Reusable OO Components," *Proceedings of the International Conference on Software Reuse* , June 1998.

[11] Taylor, D., "Are Objects Obsolete?", *Component Strategies*, vol. 1, no. 1, July 1998, pp. 16- 17.

[12] Zweben, S., S. Edwards, B. Weide, and J. Hollingsworth, "The Effects of Layering and Encapsulation on Software Development Cost and Quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 200-208.

Component-Based Development Environment: An Integrated Model of Object-Oriented Techniques and Other Technologies

Oh-Cheon Kwon, Seok-Jin Yoon and Gyu-Sang Shin

Computer & Software Technology Laboratory

ETRI(Electronics and Telecommunications Research Institute)

Taejon, Korea

{ockwon, sjyoon, gsshin}@etri.re.kr

Abstract

Object-Oriented Programming(OOP) has some weaknesses in that it does not always produce reusable software and is not suitable for a large project and does not support the complete encapsulation of classes due to the inheritance of subclasses. As a evolutionary method of OOP, Component-Based Software Engineering(CBSE) or Component-Based Development(CBD) has recently been hot issues for the Object-Oriented community and reuse community, and the component market is also growing rapidly. Thus, in order to overcome the limitations of OOP and maximize the benefits from reuse, the authors propose an integrated model that links the OOP paradigm with the emerging CBD paradigm. In addition, the authors review most of the technologies related to an integrated CBD environment and describe our current research on re-engineering that will be extended to support a whole CBD environment.

1. Introduction

Object-Oriented Techniques have been considered a powerful means of solving software crisis through their high reusability and maintainability. However, Object-Oriented Programming(OOP) has not brought many benefits since they have not provided interoperability of components at the binary/runtime level. The following article from Byte Magazine shows that the paradigm for a software development method has shifted from OOP to Component-Based Development(CBD).

"Object technology failed to deliver on the promise of reuse. Visual Basic's custom controls succeeded. What role will Object-Oriented Programming play in the component-software revolution that's now finally under way?" [Byte94]

A software component is defined as a unit of software that implements some known functions and hides the implementation of these functions behind the interfaces that it exposes to its environment. The term componentware is also described as software assembled from a set of components [Ovum98]. Ovum's definition of a component is limited to software, but in order to maximize the benefits from reuse we should include all information generated in the course of development, ranging software products to requirement specifications, design specifications, project plans, test plans, quality plans, user manuals, including ideas, methods, experiences, etc.

Component-Based Software Engineering(CBSE) or CBD is considered a new paradigm of a software development method that consists of component production, selection, evaluation and integration. In order for CBSE to be successful and effective, many problems related to this new technology should be solved. This paper reviews several technologies that need to be investigated in order to build a Component-Based Development environment that will work as a total solution. Since CBSE requires the linkages between many related technologies, a tool supporting CBSE/CBD may not work as a feasible tool if these technologies are all not implemented in the tool. There are some hopeful component infrastructure tools or models such as OMG's CORBA 2 specification [Orfali97], Microsoft's ActiveX/OCX and COM/DCOM[Chappell96], and Sun's Java Beans[Sun97]. However, since these component tools are not complete and sophisticated they should include the technologies described in the next section in order to be used for a total solution.

2. Trend of CBD Related Technologies and our Position

As shown in Figure 1, we propose a CBD environment that consists of several technologies related to CBSE. These technologies are categorized into component building technologies and component use technologies. Component building technologies contain 'Design/Development for Reuse' based on Object-Oriented Programming(OOP), Design Patterns and Frameworks, Re-engineering, Component Description, Domain Engineering and Component Certification. Component use technologies include 'Design/Development with Reuse' based on a Reuse Repository, Domain Engineering and Reuse Metrics. Domain Engineering is relevant to both component building technologies and component use technologies. As shown in Figure 1, the CBD environment has been built as an integrated model in order to merge the OOP paradigm with the emerging CBD paradigm, thereby leading to maximization of reuse effects through overcoming the limitations of OOP that is described in the next section. The authors review most of the technologies supporting CBD and describe our current research on re-engineering that will be extended to support a CBD methodology.

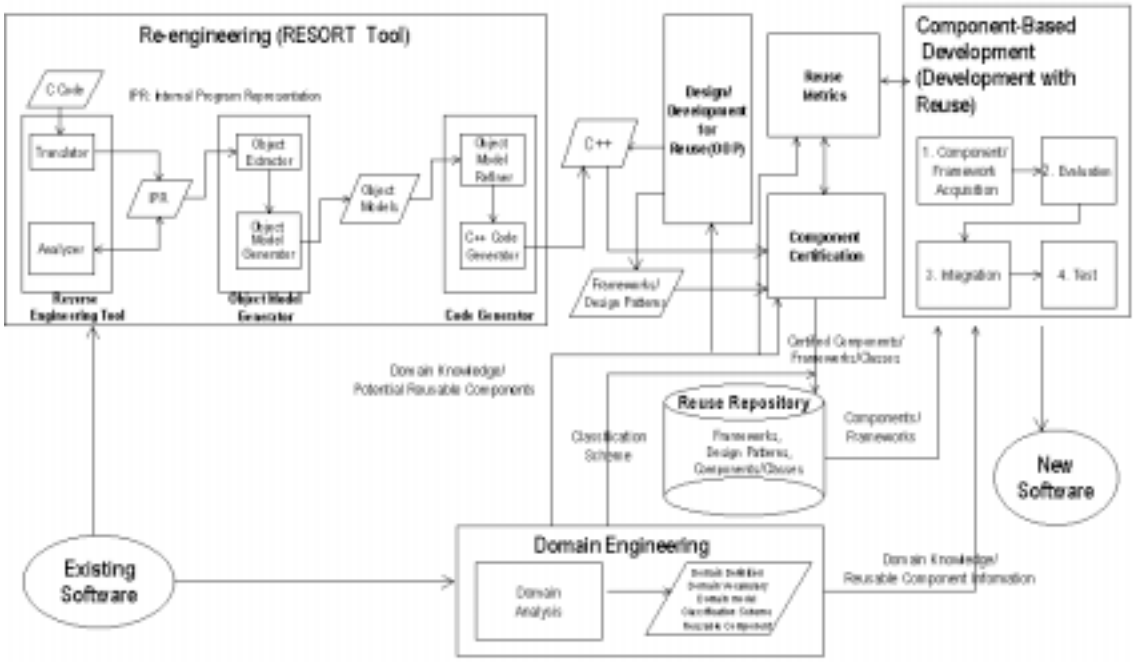


Figure 1. A Component-Based Development Environment Integrating OOP with Related Technologies

2.1 'Design/Development for Reuse' Based on Object-Oriented Programming

The major benefits of CBD can accrue from the inherent reusability of software components. Therefore, 'Design/Development for Reuse' technology based on Object-Oriented Programming(OOP) should be introduced to an IT organization so that components with good quality and reusability can be created, classified, stored and managed for future reuse. However, it is well known that designing reusable components is hard and supported by too few tools, so it is only a job for the best skilled designers and programmers[Short97]. OOP has some weaknesses in that it does not always produce reusable software and is not suitable for a large project and does not support the complete encapsulation of classes due to the inheritance of subclasses. We should agree that the component technology owes some of its characteristics to OOP but the classes built by OOP are not always reusable components.

For the above reasons, some people argue that object-oriented development has failed to draw strong attentions from software developers. Nevertheless, others argue that components are just encapsulated objects/classes, and large scale reuse will be also finally achieved if software developers are motivated to reuse reusable components and provided with class libraries and frameworks described in the following section. From our points of view, OOP should be used to create encapsulated components that consist of some objects and enable users to plug-and-play easily with the components in order to build a new system. Thus, since CBD is not a revolutionary but evolutionary method, the CBD method needs to be cooperated with OOP within a CBD environment.

2.2 Design Patterns and Frameworks

Design patterns can be defined as an abstract solution for common problems in Object-Oriented Programming and consist of classes, objects and interactions between them. Experienced software engineers can find these patterns from their knowledge and experiences and reuse them in developing applications with the same pattern. Gamma[Gamma95] defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". Many design patterns have been found in software architectures[Coad95], [Gamma95].

In OOP, the concept of a framework has proven to be a very useful way to reuse skeletal implementations of architectural designs that can be customized by application developers. Some commercialized frameworks such as IBM's SanFrancisco, Microsoft's MFC, Inprise's OWL and Icon Computing's Catalysis[Icon98] have been released to the market. Since Microsoft's Visual Basic, ActiveX/OCX and COM/DCOM, Sun's Java Beans and OMG's CORBA were announced in the software component market, OOP enthusiasts have argued with component enthusiasts about which way to go to enhance software development productivity and to reduce development costs. In particular, the birth of Visual Basic in 1991 has led to the creation of the large component market.

2.3 Re-engineering

A narrow definition of re-engineering focuses on code restructuring whereas a broad definition of it includes forward engineering as well as reverse engineering. Reverse engineering does not change the system but provides alternate views of the system at various levels of abstraction. Forward engineering is the process of system-building, starting with an existing system structure.

The ultimate objective of our research into re-engineering of the broad context is to implement RESORT system (REsearch on object-oriented SOftware Re-engineering Technology) that supports an integrated software re-engineering environment through transforming an existing system written in procedural languages into a system written in Object-Oriented languages, thereby leading to modernize outdated software and to elevate reusability and maintainability of existing software. As shown in Figure 1, the RESORT system consists of 5 subsystems: a program comprehension tool, a re-documentation tool, a domain object modeling tool, an object extraction tool and a C++ code generator.

The RESORT system provides users with a process that validates and refines the extracted object models after the object generator extracts object models automatically through analyzing Internal Program Representations(IPRs) translated from procedural C programs by a reverse engineering tool.

2.4 Component Description(Specification)

In the case of 'black box reuse', a component consists of an executable component itself that will be used to meet user's intentions or requirements, and a specification that describes the model and service/interface of a component and how to use it. A specification of the component can be specified by an interface description language or a formal description language.

When application developers implement software using a CBD method, they do not need to know about implementation details. They can use components easily by calling the operations described in the interface of a specification separated from implementation. For example, an Interface Definition Language(IDL) is part of the CORBA specification for defining interfaces and operations[CORBA95]. In the 'design for reuse' activity, if the specification of a component is described formally using a formal specification language, it can be transformed into target source code.

In Architecture Specification Language(ASL), the functionality of a component is defined by its interfaces that are classified into provided interfaces and required interfaces[CORBA95]. In addition, description techniques such as UML[UML97] and Catalysis[D'Souza97] have been well known to a CBSE area.

2.5 Domain Engineering

Domain engineering consists of domain analysis and domain modeling activities. A domain is a specialized body of knowledge and an area of particular business. Domain analysis is an activity of identifying objects and operations in a set of related systems, and identifying common objects across a set of existing or future systems through commonality analysis. Domain analysis is a key to successful reuse since it enables us to find reusable components from a legacy system and to decide which reusable components should be created for a new system in collaboration with 'design/development for reuse'. As shown in Figure 1, most information acquired from domain analysis is also used for building a new system using reusable components(i.e., 'design/development with reuse'). A domain modeling activity is associated with understanding component requirements and translating them into interface descriptions of a component specification.

Domain analysis is carried out for a set or family of related systems, whereas system analysis is performed for a single system. Domain models can be used to check the completeness and consistency of system requirements[McClure96]. After performing domain analysis and modeling, we can determine domain vocabularies, domain models, possibly reusable components and classification schemes, as shown in Figure 1.

2.6 Component Certification

After components have been created or possibly reusable components have been extracted, these components must be evaluated and certified through a software metrics system. Component certification enables reusers to eliminate 'Not Invented Here(NIH)' syndrome that is one of barriers to obstacle the success of reuse.

2.7 'Design/Development with Reuse'

If a reuser is not satisfied with a component retrieved from the reuse repository, he/she needs to modify or customize the component. The component to be modified is called a 'white box' component. An example of 'white box reuse' in OOP is a process of developing programs through writing child classes after understanding and subclassing of the parent classes. 'Black

box reuse' or 'gray box reuse'(reuse by instantiation/parameterization) is easier to use than 'white box reuse' since the internals of the related classes do not have to be understood. Class users of 'black box reuse' are only required to grasp the interfaces of classes.

2.8 Reuse Metrics

Reuse metrics are used to identify the components that are highly reusable and the business areas and systems in which reuse has the high potential to provide the greatest benefits to an organization. They can identify these components that recur most frequently across the systems through domain analysis. McClure[McClure95] describes 10 factors for reuse metrics as follows: commonality of a component, reuse threshold of a component, reuse merit of a component, reuse creation cost of a component, reuse usage costs of a component, reuse maintenance cost of a component, degree of commonality of a system, degree of reusability of a system, reuse target level of a system, reuse merit of a system. Reuse metrics are the key technology to elevating benefits from reuse together with component certification.

3. Related Work

In this paper, the authors describe related research work in terms of two aspects: a re-engineering method and a CBD method since we currently focus on a re-engineering area among the technologies for supporting a CBD environment.

A typical re-engineering method called COREM(Capsule Oriented Reverse Engineering Method) [Gall95] that carry out design recovery from a legacy system through a reverse engineering process, create object models by manual intervention and finally build Object-Oriented software. CORET(Capsule Oriented Reverse Engineering Technique) project[Gall97] that started in order to enhance the COREM prototype, has recently been implementing a process that extracts objects from procedural C code and produces C++ code using extracted objects. However, CORET project has used commercialized tools in order to reverse engineer C code and model the objects of a target system, whereas our RESORT project has developed these tools for ourselves.

There have been two representative researches into a CBD environment. Firstly, a CBD methodology called Catalysis[D'Souza97] has been built in order to support object modeling, specification activities and design models by composition. Catalysis is a UML[UML97] and OMG[CORBA95] compliant method for component and framework based development. The Catalysis method provides users with frameworks that can be applied to various areas ranging from business models and many common design patterns to very fundamental definitions. Secondly, Andersen Consulting[Ning98] has carried out a research project for building a Component-Based Development environment that enables an architecture-driven and component plug-and-play style of software system development. Andersen Consulting's CBD environment includes many advanced research ideas and results from the academia and industry in the field of software components and architectures.

4. Conclusions

Component-Based Development(CBD) can be regarded as the best way of developing applications that are cheaper and faster. These applications can match a growing distributed object technology. The authors proposed an integrated CBD model of the component technologies and OOP that produces reuse effects by synergy. The technologies related to the CBD model described in this paper are very central to the success of CBD. In particular, our research is currently focusing on re-engineering that extracts object models from a legacy system written in C and then generates C++ code. After completing our re-engineering project this year, we will extend our research to 'design/development for reuse' that creates a new component and framework, including other technologies related to CBD.

5. References

[Byte94] J. Udell, "Cover Story: Componentware", Byte Magazine, May 1994.

[Chappell96] D. Chappell, "Understanding ActiveX and OLE", Microsoft Press, 1996.

[Coad95] P. Coad, D. North, et al., "Object Models: Patterns, Strategies, and Applications", Yourdon Press, Prentice Hall, 1995.

[Coleman94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, et al., "Object-Oriented Development: The Fusion Method", Prentice-Hall, 1994.

[CORBA95] Object Management Group, Inc., CORBA, "The Common Object Request Broker: Architecture and Specification", July 1995.

[D'Souza97] D. D'Souza and A. Wills, "Composing Modeling Frameworks in Catalysis", TechnicalReport, Icon Computing Inc., 1997.

[Gall95] H. Gall, R. Klosch and R. Mittermeir, "Object-Oriented Re-Architecturing", Proc. of European Software Engineering Conference, September 1995.

[Gall97] H. Gall and J. Weidl, Object-Model Driven Abstraction-to-Code Mapping, Technical Report, Technical University of Vienna, December 1997.

[Gamma95] E. Gamma, R. Helm, et al., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[McClure95] C. McCLure, "Model-Driven Software Reuse", Extended Intelligence, Inc., 1995.

[McClure96] C. McCLure, "Software Reuse Techniques: A Guide to Adding Reuse to the Software Process", Extended Intelligence, Inc., 1996.

[Ning98] J. Q. Ning, "CBSE Research at Andersen Consulting", Proceedings of the 1st International Workshop on Component-Based Software Engineering, April 1998.

[Orfali97] R. Orfali and D. Harkey, "Client/Server Programming with Java and CORBA", John Wiley and Sons, 1997.

[Ovum98] K. Ring and N. Ward-Dutton, "Componentware-Building it, Buying it, Selling it", Ovum Ltd., 1998.

[Short97] K. Short, "Component-Based Development and Object Modeling", Sterling Software, Technical Handbook Version 1.0, February 1998.

[SUN97] Sun Microsystems, Java Beans 1.01, Sun Microsystems, 1997.

[UML97] UML Group, Unified Modeling Language 1.1, Rational, 1997.

Building an Effective CBSE Handbook

Position Paper for ICSE99 Workshop

"Developing a Handbook for Component-Based Software Engineering"

ICSE99, Los Angeles, May 17th-18th, 1999

Martin L. Griss

Hewlett-Packard Laboratories, 1U16

1501 Page Mill Road, Palo Alto, CA 94304-1126

Tel: (650)857-8715, fax: (650)813-3668

Email: griss@hpl.hp.com

URL: http://www.hpl.hp.com/reuse

Abstract

The community has made tremendous progress in understanding the critical technology, methods, processes, and organizational factors that lead to effective architected, component reuse. There is of course much more we can do, but still, far too few software developing organizations see reuse or CBSE as the solution to the problems they have, nor do they practice component reuse systematically. Far too few schools teach much about reuse and components. A CBSE handbook, summarizing industry best practice and state-of-the-art, could be an extremely useful vehicle for moving us forward.

However, experience with such handbooks in the past suggests that we should think very carefully about who the audience(s) for the handbook are, how this handbook should be used, and how it should consequently be structured. I propose several additions and modifications to the strawman handbook outline, to clarify these issues, as well as incremental adoption, organizational and process maturity, and CBSE methods.

Keywords: software engineering, components, education, technology transfer, best-practice, architecture.

1 Background

For the last 16 years, I have worked on reuse methods, modeling, component software, software education and software engineering process improvement at HP[griss95]. I have also gained a lot of experience with software engineering, VB and ActiveX components and reuse courses as an adjunct professor at the University of Utah[kessler97]. I have recently worked on the joint ACM/IEEE software engineering education taskforce and helped develop accreditation guidelines[griss98], and have reviewed the outline and content of a "software engineering body of knowledge".

Over these years, I have had the opportunity to lead and/or work on three handbook projects, a book on architected reuse and component-based software reuse, and several related projects. These include:

1. A "systematic reuse handbook" workgroup at WISR-4 (Workshop on Institutionalizing Software Reuse) 1991, in which we created an outline and discussed a process for building the handbook, and transferring the knowledge. See wisr91-hand.txt for a summary of this workshop. Several alternate handbook structures were discussed, such as one based on reuse process framework, one based on independent modules, one based on incremental adoption, and one with different sections for different audiences.
2. A reuse handbook for use at HP, started in 1990 and evolved during 1991-1992, incorporating the WISR handbook outline, 1. above. In this project we envisioned a "loose leaf" customizable handbook, in the spirit of some time managers or process handbooks: the idea was that a reuse manager could customize a process handbook for his group, selecting only material germane to the current process, reuse, and architecture maturity of his organization. This handbook was used extensively inside HP in the period 1992 to 1995 to establish reuse pilots as part of a corporate reuse program. A training class to introduce the handbook was prototyped. The experience gained in this work, influence the content and structure of the book I subsequently wrote.
3. An architecture handbook workshop at OOPSLA 1994(?), led by Bruce Anderson. This included a structure for architectural styles, mechanisms, standard architectures and the beginning of design patterns.
4. A book on "Software Reuse: Architecture, Process and Organization for Business Success" with Ivar Jacobson, and Patrik Jonsson[jacobson97].. This book provides a holistic view of model-based component software development (using UML), covering a variety of organizational, management, technical, architectural, process and business issues, in a consistent framework.
5. A web-based reuse process guide, produced as part of a 1996 consulting engagement by the HP PSO (Professional Services Organization) OO and subsequently refined for HP use [griss1997].

My related work on the UML standardization effort, on reuse and process maturity models, on technology transfer for an architected, component based product line at HP, on model-driven management agents, and on software engineering standards, provided the many opportunities for thinking and writing about architecture, components, models, patterns, and knowledge transfer. I have produced many training slides, run workshops, produced many small briefing documents and templates.

2 Position

I learned a number of things from the various handbook experiences above, which could be valuable in the proposed CBSE handbook effort, and the workshop itself.

Content is of course really important. We have lots of promising technology, methods, and guidelines that can be adapted from the reuse, architecture, OO and patterns community. But too little is taught systematically [bucci98], or practiced widely. As an example, we believe that we have a good understanding of how issues and choices in several areas could influence critical success or failure of architected, component reuse -- CBSE:

- Technology - OO; architecture; patterns; components; interfaces; generators; library systems and classification schemes; UML, ….
- Process - domain analysis, CFRP, DFR-DWR, incremental pilots, process, product and reuse metrics; process maturity models; economics, Catalysis, RSEB, …
- People: distinct create, reuse, manage, support organizations; architecture teams, explicit high-level management leadership; domain- and component- engineering skills; roles; …

As an expert in many of these areas, I can contribute to content in many sections of the strawman handbook, such as 1b -- "how does CBSE relate to similar fields ", 2a - "processes and methodologies", 2b - "organizational issues", and to many of the sub-sections of 3, such as "UML", "libraries", tool, etc.

A key learning from the HP handbook project, was the tension between size and usability of the handbook (partially addressed by a loose leaf approach). Another was the need to provide customized material to different groups. Another was the difficulty of keeping such a handbook up-to-date. And finally, the difficulty in getting groups to use such handbooks.

Thus an important concern for the handbook effort proposed is to discuss the goals and audiences(s)of the handbook and how we expect it to be used. This will certainly have an impact on the structure of the outline and mechanisms used for customization and adoption, if any.

Some other observations and suggested changes to the strawman outline:

- The preamble specifically mentions *engineering handbook.* Is the intent to exclude, or segregate management material, perhaps to later produce a management handbook for CBSE? Certainly material on adoption, organization, economics have a large management as well as process content. One idea would be to have a separate management section along with the engineering sections, or for each section to have the managerial implications as well as a technical implications.
- I would separate recommended processes and methods from the adoption section, 2. That is, I would collect managerial and adoption issues in one section, adding material on CBSE and process maturity models, the role of pilots etc.. I would then have a separate section, covering recommended CBSE processes and methods. For example, we should mention domain engineering (e.g., FODA, ODM, Synthesis), Catalysis, Rational Unified process, RSEB, etc., or least integrate these into a recommended CBSE processes for architecture, component design and component use. The CFRP (common framework for reuse process) could provide a useful source of process material. Alternatively, the methods could be considered part of technology or although that is not the current scope of technology in section 3 -- certainly methods and process are more than just notation and modeling languages.
- Section 4, could be more usefully structured to distinguish "best practice", "state-of-the-art", "advanced", "domain-specific" and "research" areas, much in the spirit of the Software Engineering Body of Knowledge [IEEE98].
- It is important to add a reference material, bibliography and some templates and tables.

Key outline elements of the HP reuse handbook (developed before the growing use of modeling, UML) are included in the appendix. Book also provides an interesting overall outline, separating

architecture, process and management issues[jacobson97]. The knowledge captured in the CBSE handbook, as a component community best-practice consensus could be injected into the evolving software body of knowledge (SWEBOK)[IEEE98].

3 Comparison

There are now many books on reuse, components and architecture, and a growing number of reuse courses, both at schools and from vendors. However, apart from the DOD Reuse technology roadmap [DOD96], and the SPC adoption guidebooks, there does not seem to be a community consensus on CBSE best practice, or how best to drive adoption. No one seems to be addressing this in the context of the new licensing and accreditation context.

References

[bucci98] Bucci, Paolo, and Timothy J. Long and Bruce W. Weide, "Teaching Software Architecture Principles in CS1/CS2", Position Paper, Third International Software Architecture Workshop, Orlando, Nov 1-2, 1998, pp. 9-12. (See http://www.cis.ohio-state.edu/).

[DOD96] Reifer, Don, *"Reuse Technology Roadmap"*, Department of Defense, 1996.

[griss95] Griss, Martin and Wosser, Marty, "Making reuse work at Hewlett-Packard." *IEEE Software*, January 1995, V12, #1, pp. 105-107. .

[griss97] Griss, Martin and Balasubramanian, Ramesh, "Reuse Process Guide", *HP PSO/HPL working document*, Nov 1997.

[griss98] Griss, Martin, "Letter from the Executive Committee," *Software Engineering Notes*, Vol. 23, No. 5, Sept. 1998, pp. 1-2.

[IEEE98] Bourque, Pierre at. al., "Guide to the Software Engineering Body of Knowledge (SWEBOK)", Strawman Version, IEEE Computer Society, September 1998. (See http://www.ieee.org/).

[jacobson97] Jacobson, Ivar, and Martin Griss and Patrik Jonsson , *"Software Reuse: Architecture, Process and Organization for Business Reuse,"* Addison-Wesley-Longman, 1997.

[kessler97] Kessler, Robert R.,"CS451-CS453 - Software Engineering Laboratory", Computer Science Department, University of Utah, Salt Lake City, UT, 1997. See http://www.cs.utah.edu/~cs451.

[tracz98] Tracz, Will, and Mary Shaw, and Martin Griss, and Don Gotterbarn, "Panel: Views on the State of Texas Licensure of Software Engineers", *Proceedings of FSE-6: ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering*, Nov 3-5, Orlando, ACM SIGSOFT, 1998, pp. 203-208.

Biography

Martin L. Griss (griss@hpl.hp.com) HP Laboratories, *http://www.hpl.hp.com/reuse.*

Martin is a Principal Laboratory Scientist in the Software Technology Laboratory at Hewlett-Packard Laboratories, Palo Alto, California. He has over 30 years of experience in software development, education and research. For the last 16 years at HP, he has researched software engineering processes and systems, systematic software reuse, object-oriented development and software process improvement at HP. He created and led the first HP corporate reuse program and participated in the development and execution of the HP corporate software initiative. He led HP efforts to standardize UML for the OMG, and is a member of the OMG UML revision taskforce. He was previously director of the Software Technology Laboratory at HP Laboratories, and an Associate Professor of Computer Science at the University of Utah.

Martin is co-author of the influential book "Software Reuse: Architecture, Process and Organization for Business Success" (with Ivar Jacobson and Patrik Jonsson), which holistically addresses technology, people and process issues in a UML framework. He writes numerous articles on software engineering and a reuse column for the "Object Magazine/Component Strategies". He lectures widely on systematic reuse and software process improvement. He is a member of the ACM SIGSOFT Executive Committee, and of the joint IEEE/ACM committee on software engineering education and accreditation. He is an adjunct professor at the University of Utah, co-developing component-oriented software engineering courses, and advising on software engineering curriculum design.

Appendix: Extract from HP reuse handbook, circa 1994.

- **Introduction**
  - *Audience and purpose*
  - *Philosophy and assumptions: definitions, guidelines vs processes, …*
  - *How to use this handbook*
- **Preparing for a reuse program**
  - *Starting a reuse project*
  - *Work product inventory*
  - *Factors influencing effective reuse*
  - *Maturity models*
  - *Effective adoption*
- **The reuse process**
  - *Managing the reuse process; planning, organization , economics, metrics, …*
  - *Producing reusable work products: architecture, domain engineering, coding guidelines,…*
  - *Using reusable work products: finding, adapting requirements to match assets, adapting assets to meet requirements*
  - *Supporting reusable work products:: testing, certification, documentation, library issues*
- **Reference documents**
  - *Working references*
  - *Glossary*
  - *Available resources*
  - *Online templates*
    - *Assessment*
    - *Metrics*
    - *Inventory*
    - *Costing*
    - *Architecture discovery*
    - *Best practice capture*
    - *Domain dictionary*
- **Indices**

- *Topics*
- *Concepts*

Human, Social and Organisational Influences on Component-Based Software Engineering

Douglas Kunda and Laurence Brooks
Department of Computer Science, University of York, UK
Douglas@cs.york.ac.uk

Abstract

This paper discusses some human, social and organisational issues affecting Component based software engineering (CBSE) processes and the introduction of CBSE in organisations. We present some of the non-technical problems we have identified from literature and case studies. In each case we suggest some actions that developers and project managers should consider in order to alleviate these problems.

1. Introduction

Building systems from pre-fabricated software components, also known as Component-Based Systems Engineering (CBSE), changes the focus of software engineering from one of system specification and construction to one of component: identification, qualification, adaptation, integration and upgrade (for system evolution). The CBSE approach relies on the existence of an inventory of existing software components, the emergence of component integration technologies such as Common Object Request Broker Architecture (CORBA) and Component Object Model (COM), and the development of organisational capabilities for CBSE trade-off analysis and design. Growing capabilities in each of these areas is encouraging the migration towards COTS-based systems in a broad range of domains. CBSE can potentially be used to reduce software development and maintenance costs, as well as reducing software development time by bringing the system to markets as early as possible [2][6] . CBSE also improves reuse across programs and promotes a competitive component marketplace.

Software systems do not exist in isolation they are used in social and organisational contexts. Experience and many studies show that the major cause of most software failures is the people rather than technical issues [3][9]. Even with the availability of a wide range of advanced software development methodologies, techniques and tools, serious problems with software are still being faced. It is the people and culture of the organisation that determines how any system is used. For example poor training may result in people not co-operating with the information system leading to failure and project abandonment [1].

Curtis et al, [3] has highlighted that human, social and organisational considerations affect software processes and introduction of software technology. Le Quesne, [3] agreed that certain aspects of the design of information systems would make its likely success dependent on characteristics of the particular organisation environment. Friedman and Kahn Jr., [4] give two examples of computer systems that passed technical muster, but posed ethical concerns or made little sense for the social context of their use.

This paper presents some thoughts on the organisational issues to be considered when developing software application using CSBE approaches.

2. Social and organisational factors

Studies by Le Quesne, Grudin and others [5][7] have identified a number of social and organisation issues that affect the software systems and these can be best summarised in the three level behavioural model by Bill Curtis, et al [3]. Mullins adds the environmental level to the individual level, group level and organisational level identified by Curtis [10]. There are some factors that overlap at different levels and the figure 1 describes the model of the factors that affect software systems.

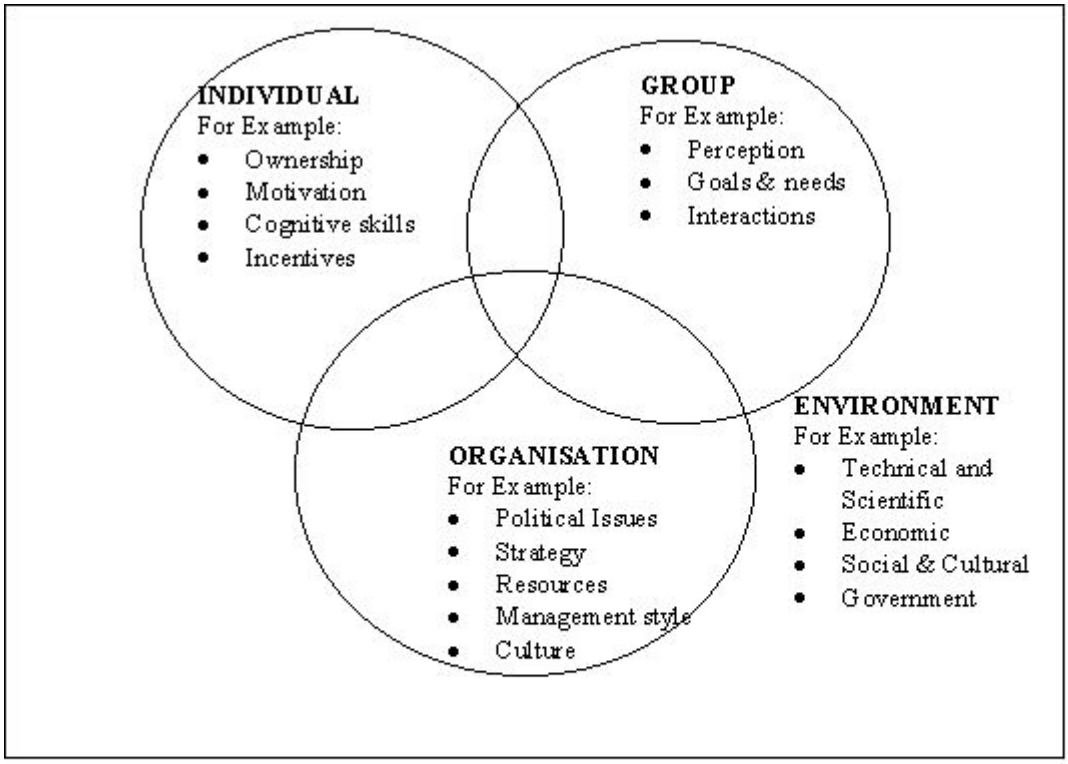


Figure 1: Organisational factors affecting software systems.

Individual behavioural factors, organisations are made up of individual members who may be stakeholders to the system being developed. Where the needs of the individual and the demands of the organisation are incompatible, this can result in frustration and conflict. The following are some of factors that should be addressed by developers and project managers during software development at individual level.
End user ownership,

Motivation and gradient of enthusiasm ,
Cognitive and education,
Incentives.

Group behavioural factors, individuals within organisation belong to one or more groups. Informal groups arise from the social needs of people within organisation. People in groups influence each other in many ways. The following are some of factors that should be addressed by developers and project managers during software development at the group level:
different perception,
different goals, and
Interactions and communication,

Organisational behavioural factors, individuals and groups interact within the structure of the formal organisation. However people sometimes relate in an informal manner and this can impact the success of the software system. The following are some of factors that should be addressed by developers and project managers during software development at organisational level:
Political issues,
Organisational and business strategy,
Organisational resources and support,
Organisational setting and management style, and
Organisational culture.

External environmental factors, the organisation functions as part of the broader external environment of which it is part. The environment affects the organisation through for example, technological and scientific development, economic activity, social and cultural influences and governmental actions [10]. This in turn will affect the software development.

3. Thoughts about actions to tackle organisational issues

A lot of effort has been made by researchers and software developers to address these organisational issues, for instance a phased development strategy. In this strategy the product idea is identified and product objectives are determined, and then at some point during the process, a portion of the product is selected and developed first before developing other portions. Incremental and evolutionary are two examples of phased development strategy [11]. The advantages are that the selected portion is delivered early, increased user acceptance, adaptation. However this approach does not solve other organisations issues such as organisational culture. Table 1 below gives a summary of the organisational problems and the proposed solution.

Table 1: Organisational factors affecting CBSE success

Organisational issue	***Problem description***	***Possible solution***
Customer ownership	Customer resistance because the user does not identify with the software system, that it belonging to them.	User participation in the software process can give a sense of user ownership.

Motivation and gradient of enthusiasm	These are factors or circumstances that induces a person to act in a particular way, they are explained in terms of the person's drives and needs. For example it is difficult to convince developers used to structured programming to migrate to CBSE, there is a threat to professionalism.	Educating stakeholders on the benefits of CBSE and training of developers in CBSE methods can help.
Incentives	The way incentives are given to developers discourages reuse and CBSE because some developers are paid the number of lines of code they write and managers are paid based on annual outputs while CBSE may take some time before the benefits are realised.	Changing the policy on incentives and bonus.
Cognitive skills	The customer/ developer's cognitive and education skills are important and can affect the software system. For instance the skills set in most organisation is currently based on structured methodology rather than CBSE.	Identification and provision of appropriate CBSE training is an important activity.
Different perception	Individuals working together in a group may have different perceptions and bias, an example is the perception of what makes good software interface.	Bringing individual to talk to each other in order to harmonise differences.
Different goals	Individuals working together in a group may have different needs and objectives and these need to be harmonised, an example is that one individual may be thinking about retirement while the other about career advancement.	Understanding individual goals and ensuring they do not sharply differ from organisational goals
Interactions and communication	Individuals working together in a group will normally interact both formally and informally. If individuals within a group are not in harmony, the success of software system will be affected.	Informal communication should be encouraged among during CBSE development process.

Political issues	These are organisational processes or principles affecting power, authority, status, etc. Some people within organisations are more powerful than others by virtual of their positions while others by their connections with powerful people within an organisation. A good example is a junior staff who can not be disciplined because of their relationship with the chief executive.	A recognition of power differences and their causes can aid in the design and development of information systems that support the organisation, its functions and individuals within it.
Organisational and business strategy	Business environment is becoming complex and dynamic - even turbulent - leading to the need for systems with shorter lives and greater adaptability. Short-term approach can discourage CBSE since benefits of CBSE may only be seen after a longer period of time.	Encourage medium to long term strategies
Organisational resources and support	Organisations work overload, skill shortage and budgetary pressure can affect the success of software system. That is not to say that all organisations with adequate resources will always have successful software systems. However this means that it can be difficult to get management (sponsor) support.	Educate management using incremental approach and successful case studies
Organisational setting and management style	The arrangement of organisational subsystems and the accompanying division of labour and hierarchy of authority relations can inhibit CBSE success.	Re-structuring the organisation according new business strategies.
Organisational culture	Organisational culture affects systems requirements and system acceptance. Customer, developer and management resistance to change in most cases is due inherent organisational culture.	Long term education and training can contribute to the change in people's attitude and organisational culture.

Therefore in this position paper we wish to recommend a social-technical development approach

as the best method to deal with organisational issues. Social-technical development is oriented to developing both social and technical subsystems in an integrated way, so that the integrated system functions in an optimal way. According to Jirokta [8] there are several approaches that have been introduced to incorporate organisational (or social) issues, called social-technical approaches. One such approach is where the social issues could be integrated with existing requirements engineering methods. In this case, an extra level of analysis could be added that incorporates the social. This would preserve the separateness and apparent strengths of each in addressing different issues, which are to be combined subsequently in some way. Multiview is a good example of this approach [1].

Other social-technical approaches include participative design and ethnography during requirement engineering phase. Participative design involves the participants directly in the requirements engineering process. Here analysts use materials drawn from meetings between participants and designers, or from user trials of prototypes. ETHICS is good example of this approach [1]. In ethnographic designs, the social and technical are seen as thoroughly intertwined and this approach attempts to develop analytic categories from the participants themselves. Here the technical is thoroughly embedded within the social.

References

1. Avison D. E and Fitzgerald G., Information Systems Development: Methods, techniques and tools, McGraw-Hill Book Company, London, 1995
2. Clements Paul C., From Subroutines to Subsystems: Component-Based Software Development, American Programmer, V8#11, Cutter Information Corp., November 1995.
3. Curtis Bill and Krasner Herb, A field study of the software design process for large systems, Communication of the ACM, 31(11):1268-1286, November 1988
4. Friedman Batya and Kahn, Jr. Peter H., Educating Computer Scientists: Linking the social and technical, Communication of the ACM, 37(1):65-70, January 1994
5. Grudin Jonathan, Eight challenges for developers, Communications of the ACM, 37(1):93-105, January 1994
6. Haines Capt Gary, Carney David and Foreman John, Component-Based Software Development/ COTS Integration, Software Technology Review, Available WWW (online) <URL:http://www.sei.cmu.edu/str/ descriptions/CBSE_body.html>, 1997.
7. Hirschheim R. and Newman M., Information Systems and User Resistance: Theory and Practice, Computer Journal, 31(5):398-408, 1988
8. Jirotka Marina and Goguen Joseph A., editors, Requirements Engineering: social and technical issues, Academic Press Limited, London, 1994
9. Le Quesne P. N., Individual and Organisational factors and the Design of IPSEs, Computer Journal, 31(5):391-397, 1988
10. Mullins Laurie, Management and Organisational Behaviour, Pitman Publishing, London, 1996
11. Wieringa R. J., Requirements Engineering: Frameworks for understanding, John Wiley and Sons, Chichester, 1996

Component-Based ERP Design

in a Distributed Object Environment

Bonn-Oh Kim

Department of Management

Univ. of Nebraska-Lincoln

Lincoln, NE 68588

March 2, 1999

Bonn-Oh Kim

Management Department

College of Business Administration

University of Nebraska

Lincoln, NE 68588-0491

Tel: 402-472-2317

Fax: 402-472-5855

Email: bkim1@unl.edu

Abstract

ERP (Enterprise Resource Planning) vendors have seen a dramatic increase in their sales this decade. Even though several vendors are producing great products and making huge profits, there are some problems to be resolved to make ERP applications a continuous success in the next decades. Current ERP applications have the low reusability and interchangeability of various modules among different vendors' packages. One of the main reasons for these shortfalls is a tight coupling of ERP domain knowledge with the particular implementation tools. Also, efforts in establishing and using the standards in specifications of ERP applications have been inconsequential. In this article, strategic steps to wield a dominant power in the future ERP market are discussed. These steps are as follows: 1. Knowledge Modeling: Abstraction of Domain Knowledge from Tools; 2. Componentization of Domain Knowledge; 3. Implementation of Componentized Domain Knowledge; 4. Marketing Strategies for Domain Knowledge Components.

Introduction

Since the early 1990s, a notion of business reengineering has been very popular in many companies, especially in the USA. One of the contributions of business reengineering is that corporate information systems should be viewed as an enabler to transform the business processes and consequently organizational structures. To fulfill the mission of an enabler of business transformation, corporate executives found that corporate information systems should be planned, designed and implemented from an enterprise-wide perspective. A collection of islands of software located in various divisions of an organization could not satisfy the new needs of large corporations.

ERP (Enterprise Resource Planning) vendors promise to deliver an integrated set of software systems for various functions of a company, including accounting, manufacturing, logistics and others. Recently, ERP vendors such as SAP, Baan, PeopleSoft, Oracle and J. D. Edwards have seen their sales growing exponentially. Behind the successful stories of ERP, however, there are several issues to be dealt with in order to adapt to the ever-changing computing environment and maintain the competitive advantages.

Borrowing the idea from the industrial manufacturing, software components built based on standard specifications can be a building block for resolving the current problems in designing ERP applications. To build software components, however, we need to have a set of specifications at the knowledge level. In this article, knowledge modeling abstracted from the implementation tools is discussed as a precursor for building the components for ERP applications after the problems of current ERP applications are discussed and core competencies of ERP vendors are reviewed from a perspective of overall computing architectures.

Problems of Current ERP Design

Currently, each ERP vendor has been developing its own proprietary systems in various domain areas. Since ERP customers prefer the seamless systems across their business functions, ERP vendors are continuously expanding into new domain areas. However, one vendor does not necessarily produce superior ERP packages across all business functions. Each vendor maintains superiority in some functional domains, e.g., PeopleSoft for human resource management.

From a perspective of ERP customers, they have to opt for using all ERP applications primarily from one vendor or selecting many packages from different vendors. If customers can choose the best from different ERP vendors without worrying about the compatibility among different vendors' ERP packages, they can maximize the productivity gains by installing the best ERP applications in their organization. From an ERP vendor's perspective, it is very difficult to specialize in any particular domain functions (e.g., manufacturing, financials, etc.) because many customers want a smorgasbord of ERP packages from one vendor. If ERP packages from different vendors are interchangeable or compatible, some problems aforementioned can be somewhat resolved.

There have been overlaps in efforts developing virtually the same type of applications (e.g., accounting packages) by many different vendors. Reinventing a wheel is a last thing we need to do. Current ERP designs in industry lacks reusability and interchangeability of domain application components. To develop a successful and dominant company in the ERP market, a strategic move to a component-based ERP design and marketing will be required. ERP vendors should be in a business of specifying the ERP components as well as building them. Once the design of specifications for ERP components is produced, manufacturing of each component can be outsourced to third-party developers.

Core Competencies of ERP Vendors

Currently, ERP vendors' core competency appears to reside in its conceptualizations of application domain knowledge in financials, manufacturing, distribution and others rather than the application development tools (e.g., OneWorld from J. D. Edwards). Even though they are making profits by selling the ERP software on different machines, there will be even more profitable and huge markets for specifying and producing various components of each application. Readers are reminded that automobile companies make huge profits specifying and selling the automobile parts (or components). For example, GM controls an automobile business by specifying how the third-party manufacturers produce the parts for GM cars and trucks. Controlling the standards for specifications of parts endows an intrinsic dominant power to GM. GM does not produce all the parts. GM basically controls the specifications of parts.

By breaking down a huge complex application package into many independently packaged components, we can sell each component to all types of customers. Customers of ERP products do not have to be a mid-sized company wishing to have the financial applications installed. We can expand the market to software developers and end-users as well as our traditional mid- to large-sized companies. For example, if we package the accounts receivable application as a separate independent product using DCOM (Distributed Component Object Model) standard in

Microsoft Windows 98/NT environment, potential profits could be immense.

Component-Based ERP Design

What is important is that we need to rethink how we develop the applications. Software design should be more or less like designing and manufacturing automobiles. GM and Ford make money by specifying components and assembling cars as well as by manufacturing parts. ERP vendors should be prepared to design and sell components of applications as well as the final whole ERP solution. Packaging of each component needs to be done using the industry standards. Once we conceptualize and build each component, we can package it using Microsoft DCOM, OMG's (Object Management Group) CORBA (Common Object Request Broker Architecture), SUN's JavaBeans or whatever. ERP vendors are not in a business of setting the standards for packaging. Their strength should be in conceptualization and specification of components and packaging them in various forms.

Currently, most of conceptualizations of knowledge in application domains are already available in the forms of computer code and some high level designs. Unfortunately, however, they are frequently hidden and dormant. They are tightly coupled with the tools (e.g., OneWorld). What we need to do is to abstract the knowledge from the tools and specify each knowledge component independently of any tools. Then, each knowledge component can be manufactured using whatever tools in a massively distributed environment. Thereby, we can give a new profitable life to this latent asset of ERP vendors. We need to recreate and repackage the knowledge. For effective packaging and distribution, it is very important to adopt a distributed object-oriented approach in software development and to normalize the database systems.

More specifically, the following needs to be done:

1. Knowledge Modeling: Abstraction of Domain Knowledge from Tools:

When designing software systems, we need to think about what is constantly changing and what is not. Invariant parts of the system should be separated from the variant parts in order to make a whole system adaptable to a new environment. In the ERP market, the domain knowledge in accounting or manufacturing does not change much over time while implementation tools are almost constantly changing. When there are new implementation tools available, the domain knowledge should be easily ported to a new tool environment.

Knowledge models in various domains should be independent of any lower-level abstractions, including implementation tools, middle-ware and others. Knowledge modeling has been a research topic in artificial intelligence for a long time and there are many models available now. For the ERP knowledge, however, two modeling tools can be most effective: i.e., object-oriented modeling for business activities and entity-relationship modeling for persistent data. These object models and entity-relationship models should be independent of any particular tools or technical environment. Depending on a market situation, we should be able to implement the knowledge models in almost any programming language and hardware environment.

2. Componentization of Domain Knowledge

One of the most important characteristics of components is the separation of "what" from "how". Each component should have a clearly defined interface specifying "what" it does while hiding "how" it does. Using this interface, each component can communicate with other components. A collection of objects will constitute a various grain size of knowledge in each domain application as patterns or frameworks of objects. IBM's San Francisco project can provide a good reference model. For more details, visit the following Web site: http://www.ibm.com/Java/Sanfrancisco/.

3. Implementation of Componentized Domain Knowledge

As we have seen over many years, technical environments are changing at a very fast speed. ERP vendors should not be in the component packaging business. Currently, there are several packaging standards available, including Microsoft's DCOM (Distributed Component Object Model), OMG's (Object Management Group) CORBA (Common Object Request Broker Architecture) or SUN's JavaBeans. Knowledge components specified should be packaged using whatever standards popular. If we design and specify the domain knowledge independently of any particular packaging standard, we should be able to repackage the domain knowledge components as dictated by the market.

4. Marketing Strategies for Domain Knowledge Components

To become and stay a dominant power in the ERP market, an ERP vendor needs to control and own the standards for ERP components and allow others to manufacture the approved components. Open Applications Group's work can be a good place to see what is going on in the area of open standards. For more details, readers are referred to the following Web site: http://www.openapplications.org/. Microsoft practically controls the microcomputer market by owning a standard in the operating systems while many other companies build software based on Microsoft's standard. Each ERP vendor does not have to manufacture all the components.

Once an ERP vendor possesses the standards, it should be an owner of components catalogue. The catalogue of ERP components should be a market place where software builders can shop to build or customize their own applications. it should be more than just a component builder. We should create and control the market for ERP components. Readers are reminded that NYSE (New York Stock Exchange) has become a very profitable venture by creating a market for stock exchanges and by controlling how stocks should be exchanged. We should be able to set the rules for building and exchanging components in the ERP market. Thereby, we can dominate the ERP market for a long time.

Concluding Remarks

As in the industrial sectors of the USA economy, there will be more profits in specifying and designing software packages rather than just manufacturing them even in the software business in the near future. These days, manufacturing of software can be achieved less costly by exporting it to countries like India. In the ERP market, we need to think more like Nike.

Designing Nike shoes is a lot more profitable than just manufacturing them. Activities involved in designing the specifications for the ERP components are quite distinct from manufacturing them. Once an ERP vendor controls the knowledge component specifications for the ERP domains, it can be in a strategic position to dominate the ERP market with an absolute competitive advantage in the 21st century.

Composite Nature of Component

Wojtek Kozaczynski

Rational Software, USA

wojtek@rationla.com

Introduction

The summary of the First International Workshop on CBSE [BrWa98] contains the following definitions of a *software component*:

1. *A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.*

2. *A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time.*

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

4. *A Business Component represents the software implementation of an "autonomous" business concept or business process. It consists of all the software artifacts necessary to express, implement and deploy the concept as a reusable element of a larger business system.*

These definitions are indicative of the fact, that components may come in different forms and granularity. They also demonstrate that different participants of the development, deployment and maintenance process still see components differently. So what are the properties of components that distinguish them from other software artifacts such as objects, files, libraries, design documents, ect.?

In the following I propose a definition of a component. This definition captures what I believe to be the essence of software components. I compare this definition with the above proposed definitions and briefly discuss the composite structure and representation of components.

The definition

The initial idea for the definition comes from a number of sources. The key influence was my work on the BPCS ERP system at SSA. After many discussions and arguments we agreed that one of the key roles of a well-formed component is to assist configuration and project management. We also agreed that components were not atomic but they were composites.

At the January 1999 OMG meeting David Curtis presented the OMG Component Model [orbos/98-10-18]. During that presentation he made a comment that the main difference between a component and an object, which may look very similar when one describes them, is: *"that a component is at the same time a unit of construction, packaging and execution."*

I also came across similar ideas while discussing with Kurt Bittner, a member of the Rational Unified Process Group, the issues of representing frameworks and patterns in UML.

This lead me to the following definition of a component:

> *A component is a part of a system that is (at the same time) a unit of design, construction, configuration management, and substitution. A component conforms to and provides the realization of a set of interfaces in the context of well-formed system architecture.*

The operative phrase is "*at the same time*". Many software artifacts can be units of design, but not construction or substitution, for example a class. Some other, like a file for example, may be a unit of configuration management, but not a unit of design or substitution. Component seems to be the only unit that is all of the above (at the same time.)

The definition captures or implies component properties defined by the other definitions or renders them no-essential. Let us look at these properties one at a time:

Property	Essential definition
Non-trivial, nearly independent part	A properly designed unit of construction and configuration management should be non-trivial and as independent (or autonomous) as possible. A good design will localize parts that evolve at the same time and rate in order to minimize coupling between components.
Replaceable part (of a system)	Substituability implies replacability
Fulfills a clear (business) function	To be a unity of design a component should encapsulate a well-scoped (business) function. However, this is neither guaranteed nor it is a necessary condition. Functional cleanness of a component is its quality attribute, not a defining characteristic.
Dynamically bindable, introspective (interfaces discovered at run-time)	Dynamic binding of a component is a specific form of substitution. Introspection is useful, yet not necessary property of dynamically bindable components. DLLs supports dynamic binding [Rog97] but do not require introspection in its strict sense like Jini does.
Deployable independently	Components can be deployed independently in the sense that they can be given run-time resources and be activated. However, they will do something useful only if they coexist and communicate with other components that provide them with required services. This is the architecture context of the definition.
Contains all software artifacts implementing a business concept	The SSA definition expresses a particular choice of packaging components. Very consistently with our definition, these units (called Business Components) were also units of design, configuration management and substitution. The proposed definition does not imply that a component is a singe artifact.

Component structure and views

The last point in the table above is very important. A component is not a single artifact, but it is a collection of artifacts. Not all of these artifacts have to be seen at once. Similarly to architecture, a component has different views that expose its properties and contents relevant to specific set of concerns. These views and their contents (or

elements) are presented in the table below:

View	Elements
Design View	• *Interface (with a protocol)* • *Interactions between components* • *Relationships between components* • *Documentation, …*
Implementation (Construction) View	• *File* • *Source* • *Link library* • *Directory* • *Compile and link relationships* • *Class* • *Realization relationship between class and interface* • *Implementation relationship between file and class* • *DB table or file record* • *Shared memory structure* • *Documentation, …*
Deployment View	• *Object code* • *Executable* • *DDL (Shared Library)* • *HTML page* • *Bycode file (for JVM)* • *Run time configuration/parameters file* • *Documentation, …*

The main concerns addressed by the *Design View* are the interfaces of a component and how it interacts with and depends on other components. This is the classical OO view of a system. Additional concern is how atomic units (like a file) are packaged together into composite components and how these are layered.

The *Implementation View* is concerned with the configuration management aspects of component development. It worth noting, that in the proposed paradigm (language) classes are considered a mechanism for realizing interfaces and are associated with files. They are not, however, considered the key modeling elements at the system level since they have no containment semantics. The key modeling element is the component.

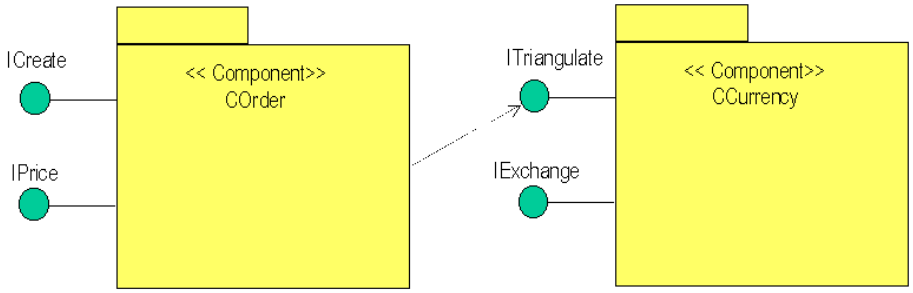
Finally, the *Deployment View* is concerned with what gets shipped and installed as the run-time version of the component.

The views are not independent of each other and could be combined, at the end of the component development cycle, into a single component model.

Component representation

UML has a concept of a component [BooRuJa98], but unfortunately it implies a single artifact. A more convenient way of representing components would be to stereotype the UML package [HoNoSo99]. This is because the package by definition is a collection of packages and/or other artifacts. The figure below shows (a part) of the *Design View* of

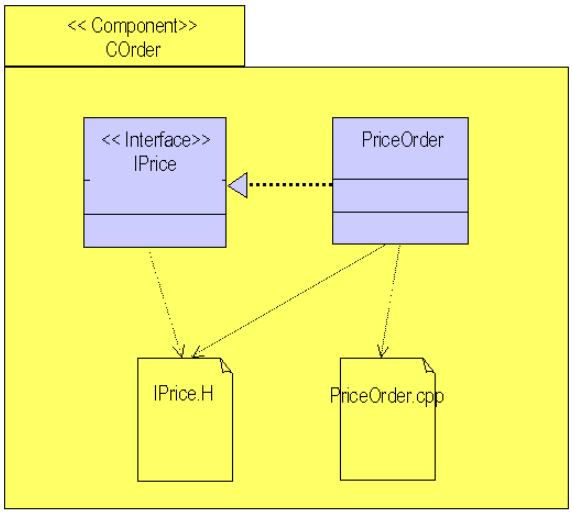
a component. This is a very simple example where a *COrder* component depends on interfaces provided by the *CCurrency* component.



Other parts of the view may contain state diagrams for interfaces, interaction diagrams of component cooperations, or detailed specifications of interfaces.

The figure below shows a portion of the *Implementation View* of a component. The interface class describes the same interface shown as a popsicle on the *Design View* and is a link between the two views. There are many other details that can be show on the *Implementation View* including directories, compile and link relationships (if they are not directly implied from other relationships), persistent date structures, etc.

We have not illustrated the *Deployment View*, but the reader should easily imagine how it would look like. Most importantly, it may contain more than one executable constructed (derived) from the elements of the *Implementation View*.



Summary

The proposed definition of a component attempts to capture its very important property of being a conceptual unity of design, construction and deployment. One of the most difficult and important roles of an architect is to decompose a system into such units – into concepts that will transcend all phases of system development.

A direct consequence of the proposed definition is that a component becomes a collection of multiple artifacts including multiple executables. This is not a common interpretation of a component. This is especially important for

the deployment and maintenance aspects of the development process.

Just as a footnote, some of the component representation aspects (in particular containment) could be more conveniently show in tabular, not graphical form.

References

[BrWa98] http://www.sei.cmu.edu/cbs/icse98/summary.html

[BoRuJa99] Grady Booch, James Rambaugh, Ivar Jacobson; *The Unified Modeling Language User Guide*, Addison Wesley, 1999

[HoNoSo99] C. Hofmeister, R.L.Nord, D. Soni, Describing Software Architecture in UML, *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1),* Kluware Academic Publishers, 1999

[DsoWil99] Desmond D'Souza, Alan C. Willis; *Objects, Components, And Frameworks With UML*, Addison Wesley, 1999

[orbos98] *CORBA Components, Joint Revised Submission,* OMG TC Document orbos/98-10-18, *November 7, 1998*

[Rog97] D. Rogerson, *Inside COM,* Microsoft Press, 1997

Transition from Conventional to Component-Based Development

Goran Grahn

Senior Consultant

Volvo Information Technology

Dept 9712, HC1N

SE - 405 08 Gothenborg, Sweden

E-mail: it.gorang@memo.volvo.se

A position paper submitted to the 1999 International Workshop on Component-Based Software Engineering.

1. Introduction

During fall 1998, I was leading a pre-study on "Introduction of an Enterprise-level Reuse Program" within Volvo Information Technology. The objective of the study was to establish a common understanding of what is needed to make the introduction of such a program successful, and to propose the way ahead. This paper is an extract of the pre-study report.

Volvo IT is organised in a number of Business Process Units (BPU's), each one supporting one of the main processes of the companies in the Volvo Group. These main processes are:

- Product Development
- Sales to Order
- Order to Delivery
- Delivery to Repurchase
- Business Administration

Each BPU contains a number of Application Areas, which is equal to one or more Domains.

The proposed program to transition from conventional to component-based development

describes activities on the Enterprise, BPU and Application Area / Domain levels, split into the following three phases:

- Initiate Reuse Program, where information on reuse concepts is presented as a basis for identifying reuse potential and establishing reuse ambition and plan within each BPU
- Preliminary Reuse process, where training starts, new roles are staffed and an initial architecture is developed as a basis for performing pilot projects. In this phase also Component Catalog tools are evaluated.
- Establish Reuse process, where the reuse process is formalized, the large-scale implementation of the reuse process is planned, and component provisioning and application assembly as the approach for software delivery is established.

2. Initiate Reuse Program

2.1 Enterprise level

2.1.1 Inform on Reuse Concepts

The purpose of this step is to present the basic concepts of Software Reuse to the organization, as a basis for initial activities at the BPU level.

2.2 BPU level

2.2.1 Identify Resue potential within each BPU

In this step a group of people representing the Application Areas and / or the major projects within the BPU is established, to start working reuse issues. Input for this activity could be:

- Existing Business Process models
- Data and Process Models from existing applications
- External sources of reusable components that may be used by the BPU

This step would identify the potential for reuse from two aspects

- What is the potential for reuse from an architectural, long term perspective
- What is the potential for reusing pieces of existing applications, considering aspects like functionality, interfaces, reliability and the technical environment where it is implemented

2.2.2 Establish Reuse ambition and plan within each BPU

The purpose of this step is to establish management support of Reuse. This includes the selection of which Application Areas are best suited for performing pilot projects, where concepts and process guidelines can be refined.

For areas selected the next phase is planned, where a preliminary process will be defined, new roles are to be staffed and an initial architecture established as a basis for the pilot.

3. Preliminary Reuse Process

3.1 Enterprise level

3.1.1 Upgrade Application Development Process to supporting reuse

The AD process currently used is architecture-driven, and based on an object oriented approach, but lacks specific activities pointing out at which point in each development phase the opportunity to satisfy requirements using existing components, or to harvest newly developed components are analyzed. Such activities supporting reuse must be added to the AD process.

3.1.2 Build Reuse skills

This step includes developing a training plan, and presenting classes to people at the enterprise and BPU levels responsible for the reuse program.

3.1.3 Staff new roles

At the enterprise level the following new roles are staffed:

- Component Provisioner, responsible for providing technical and business infrastructure components
- Reuse Support, responsible for supporting developers in the use of the catalog, and in the utilization of infrastructure components

3.1.4 Define Reuse Metrics

Reuse Metrics are defined, and a process for measuring them is put in place.

3.1.5 Evaluate Component Catalog Tools

In this step, available tools in the Component Catalog market are evaluated, and a recommendation on which tool is best suited to the needs of the enterprise is given.

3.2 BPU level

3.2.1 Start Reuse Awareness and Training process

The purpose of this step is to present the basic concepts of Software Reuse to the organization, as a basis for pilot activities at the BPU level.

3.2.2 Staff new role

The role of BPU Reuse Manager is staffed. This role is responsible for coordinating the application architecture within the BPU.

3.2.3 Establish funding of pilot projects

Methods for funding reusable components of the pilot projects are put in place.

3.3 Application Area level

3.3.1 Establish Application Area Architects

This step includes selecting individuals for the role of Application Area Architect. The architects should have some background in the Application Area and be familiar with model-based methods. The introduction of the architects involves establishing working relationships with the Business Process Owner, with Enterprise Architects and support groups, and with project leaders of current projects.

3.3.2 Develop High Level Application Area Models

Based on Business Process Models, Business Data Models or Class Diagrams and the existing System Architectures, the Application Area Architect develop high level Application Area Models and Plans regarding support for new or changed business processes, Plans for new systems or major enhancements, and Plans regarding move to new technologies.

3.3.3 Establish Application Area Architecture

In this step, the Application Area Models and Plans are reviewed with the stakeholders, which include:

- Business Process Owner
- BPU Strategy group
- Enterprise Architects

3.3.4 Staff new roles

The roles of Component Provisioner and Application Developer for the pilot projects are staffed.

3.3.5 Perform Pilot Projects

Pilot projects are prepared, executed and concluded.

4. Establish Reuse Process

4.1 Enterprise level

4.1.1 Formalize Reuse Process

Based on experience from pilot projects, the Reuse Management Process is adjusted, documented and established.

4.1.2 Introduce Component Catalog

The catalog tool recommended in step 3.1.5 is introduced. This step includes negotiating a contract with the vendor, installation of the tool, setting up standards and procedures for using the catalog and training the catalog administrator and users.

4.1.3 Wrap / Acquire / Build Common Reusable Components

Based on the Enterprise Architecture and Application Area Architectures, common Technical and Business Infrastructure components are identified. For each component is decided whether to make it available by wrapping part of an existing system, buy it from outside or develop it inhouse. Once the components are available they are documented for use in the Component Catalog.

4.2 BPU level

4.2.1 Plan Implementation of Reuse Process

This step involves planning the large-scale implementation of the reuse process, and includes extending the high-level models and expanding the training and support resources.

4.2.2 Establish funding of reusable components

Methods for funding reusable components of all software delivery projects are put in place.

4.3 Application Area level

4.3.1 Extend High Level Application Area Models

The High Level Application Area Models and Plans regarding support for new or changed business

processes, Plans for new systems or major enhancements, and Plans regarding move to new technologies initiated in step 3.3.2 are extended in scope and level of detail.

4.3.2 Wrap / Acquire / Build Reusable Components

Based on the Application Area Architecture, Business Process Components are identified. For each component is decided whether to make it available by wrapping part of an existing system, buy it from outside or develop it inhouse. Once the components are available they are documented for use in the Component Catalog.

Conclusion

This paper is an attempt to describe the transition from conventional to component-based development. At Volvo this is still theory, since our efforts currently is stopped due to changes in our business following the deal to sell Volvo Cars to Ford Motor Company. However, we firmly believe that Component-Based Development is the way ahead!

Acknowledgements

This paper draws on a paper titled "Integration on Software Process Maturity and Domain Engineering to facilitate Reuse within the Enterprise" presented at the Reuse ’97 conference in Morgantown, West Virginia, USA by Kevin T. Shea at The Boeing Company.

Software Components in Contexts and Service Negotiations

Guijun Wang, H. Alan MacLean

Applied Research and Technology, The Boeing Company, Seattle, WA 98124

guijun.wang@boeing.com, alan.maclean@boeing.com

1. Software Components

Before the term "component" became popular, terms such as subroutine, procedure, function, module, and object have been used from the earliest days of software development to denote software artifacts. Software developers have a subjective understanding of what they mean when speaking of subroutines, procedures, functions, modules, and objects, including a sense of how the meanings are distinctive one from another. In general, the notions embodied in these terms have been introduced and used with the objective of identifying and addressing issues pertaining to reusable software artifacts, the management of complexity, and the development of more flexible systems. The point of departure of this article is the premise that components should be used to address the same basic objectives, but that they constitute a distinctly new form of software artifact, one that offers better capability in meeting the objectives. As such, the same kind of direct programming language level support available for subroutines, procedures, functions, modules, and objects is required for components.

Software Engineering is an iterative process that comprises multiple stages, including modeling, design, implementation, and deployment. The "component" concept has been utilized in all these stages, albeit without consensus on a common definition. Indeed, a number of definitions for "components" appear in the literature [1]. For example, Kozaczynski defines a business component as the software implementation of an "autonomous" business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. Szyperski defines a software component as a unit of composition with contractually specified interfaces and explicit context dependencies only. In this context, a component can be deployed independently and is subject to third-party composition. The Gartner Group defines a run-time software component as a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time. Clearly, there are substantial differences between these definitions. In large part, these differences appear to arise as a consequence of different perspectives regarding the software engineering process. More precisely, the definitions reflect views of components at different software engineering stages. For example, Kozaczynski's definition reflects the component concept at the business process modeling stage of a software engineering process, while Szyperski's definition embodies the component concept at the architectural design stage. The Gartner Group's definition reflects the component concept at the deployment stage of a software engineering process.

At the architectural design stage of a software engineering process, Szyperski provides what we believe to be the appropriate general definition of a component. The definition stops short of describing the details needed to characterize what attributes a component should possess, and what functional and nonfunctional elements a component should have, as well as how they should be specified and exposed. In this regard, current research combined with developments in the commercial sector on component models help provide important details. For example, [5] [6] [7] present an architectural component model and describe a "Port and Link" framework to support component assembly for distributed systems. In this architecture level component model, a component's functional boundary is exposed by means of four types of functional elements, namely services a component provides, services a component requires, events a component observes, and events a component generates. Ports are used to represent these functional elements, and act as agents for communication between a component's internal parts and other components outside the component's boundary. Links are used to achieve communications, local or distributed, between components in general via specific local or distributed infrastructures. In a similar fashion, commercial models including

JavaBeans, Enterprise JavaBeans, ActiveX, and CORBA Components explicitly or implicitly allow components to expose their capabilities via the four functional elements. For example, the CORBA Components specification currently under consideration by OMG [4] explicitly allows a component to declare services it provides, services it uses, events it emits, and events it consumes. Rather than using Ports and Links along the lines described in [5] [6] [7], the commercial component models impose various architectural constraints on components, such as the interfaces every component must implement. A comparison between the architectural component model discussed in [5] [6] [7] and the commercial component models is summarized in [5]. Others such as [3] used a sliding interpretation approach to characterize components where a component's interface is characterized by signature, constraints, configurations, and non-functional properties.

We believe that the Szyperski component definition combined with the details supplied by the component models discussed above provides a good working definition of what a component should be in terms of the attributes it should possess, and how its functional elements should be specified and exposed. This characterization distinguishes the concept of a component from that of a subroutine, procedure, function, module, or object, and potentially establishes an appropriate level of abstraction to more effectively address the objectives of these earlier software artifacts. More importantly, this approach provides a foundation for addressing the nonfunctional properties of components and systems of components, which is discussed in the next section.

2. Functional Elements and Nonfunctional Properties of Architectural Components

As noted in Section 1, components exist in different contexts within the software development lifecycle, including business process models, software architecture, and real software systems. Here we focus on components at the software architecture level, which we refer to, naturally enough, as architectural components [3] [5] [6] [7]. As a working definition, we take the phrase "software architecture" to mean the collection of components constituting a system, together with a description of their interactions, as well as the constraints imposed on the components and interactions. Components in an architecture have contractual obligations: they provide services to others and/or generate events that others may be interested in, and they may require services from others and/or observe events generated by others. These contractual obligations are the functional elements of a component's boundary in an architecture context.

Nonfunctional properties of architectural components include performance, reliability, security, and the other so called 'ilities' that contribute to the overall Quality of Service (QoS) provided by the system. The characterization of components adopted in the previous section is central here because it enables nonfunctional properties of a component to be addressed via the functional elements of the component. In fact, as a consequence of the separation of the services provided by a component from the services required by a component, we can specify nonfunctional properties that a component offers (for provided services) and nonfunctional properties that a component expects (for required services). Of course, for components in a specific architecture, these nonfunctional properties must be specified in concrete terms to be useful. We illustrate these notions in the examples that follow.

An Architecture Style is a family of software architectures that share common architecture properties [2]. For example, the Supplier-Mediator-Consumer is an architecture style known for its flexibility and maintainability due to the separation of suppliers and consumers. A simple instance of this architecture style is the Supplier-Buffer-Consumer architecture where the Buffer is a simple Mediator. For simplicity, let's assume that the data which the Supplier supplies, the Buffer stores, and the Consumer consumes is a data stream of a simple type, say Integer. In addition, let's assume that the Supplier pushes data into the Buffer, the Consumer pulls data from the Buffer, and the Buffer has a fixed size. The functional elements of the Supplier component, the Buffer component, and the Consumer component can be specified as follows: **Supplier**: provides *InputDataStream* service, **Buffer**: requires *InputDataStream* service, provides *OutputDataStream* service, and **Consumer**: requires *OutputDataStream* service.

The *InputDataStream* service can be defined by an interface InputDataStream with operations like "void

nextInputItem(int)", and the *OutputDataStream* service can be defined by an interface OutputDataStream with operations like "int nextOutputItem()".

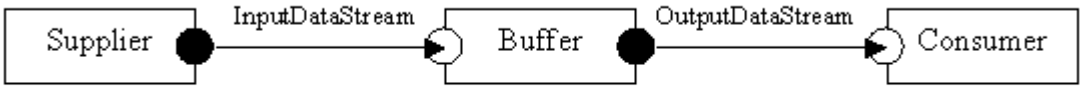


Figure 1. Components in the Supplier-Buffer-Consumer architecture

To describe the nonfunctional performance property of the Supplier, Buffer, and Consumer, we may specify that the Supplier provides data at a rate *alpha*, the Buffer expects input data at a rate *x* and offers output data at a rate *y*, and the Consumer expects data at a consuming rate *beta*. Suppose we have a QoS requirement stating that each data item in the Buffer must be consumed once and only once. Since the Buffer size is fixed, the following constraints must hold to avoid buffer overflow: alpha <= x, beta >= y. Assuming the Buffer does not do any further data processing, we know that x = y. As a consequence, we derive the constraint beta >= alpha.

Up to this point, our specification of the nonfunctional performance property assumes that the inter-component communication time (the time to transfer data from the Supplier to the Buffer, and to transfer data from the Buffer to the Consumer) is negligible. This assumption is more than likely not true, especially for distributed systems, in which it may take a substantial amount of time to transfer data from one component to another. Thus it may also be necessary to specify nonfunctional properties on communication links (for Links see [5] [6] [7]). In a distributed Supplier-Buffer-Consumer architecture, we can accomplish this by specifying that the time it takes to transfer one unit of data from the Supplier to the Buffer is t1, and the time it takes to transfer one unit of data from the Buffer to the Consumer is t2. Then the time it takes for one unit of data to reach the Buffer is t1+1/alpha, and the time it takes for one unit of data to be taken out of the Buffer is t2+1/beta. To avoid buffer overflow, the constraint t2+1/beta <= t1+1/alpha must hold. Hence, we obtain the same constraint beta >= alpha as in the case when the inter-component communication time is negligible (i.e., t1=t2=0).

A set of constraints is "under constrained" if there are more unknown variables than the number of constraints. Generally speaking, constraints for component nonfunctional properties in an architecture are under constrained. Moreover, some constraints may have an infinite number of potential solutions. As a consequence, negotiations among components to establish desired nonfunctional properties may be necessary, and system QoS may vary as a result of such negotiations. At the architecture level, components are primarily conceptual, and while constraints can be imposed, many negotiations cannot occur until the components are assembled into a real system or until run-time. We explore this point further in Section 3.

For simple architectures like the Supplier-Buffer-Consumer described above, a service request from one component may be satisfied in a straightforward manner by a service provided by another component. For complex architectures, however, a component that requires a service may have to go through a number of steps to obtain the right service from a provider. For example, the CORBA Event Model architecture, which also employs the Supplier-Mediator-Consumer architecture style, has three types of components: Supplier, Event Channel, and Consumer, each of which can be typed. The Event Channel plays the role of the Mediator. For any two components, communication between them can use either a Pull interaction style or a Push interaction style. Figure 2 illustrates the CORBA Event Service architecture and associated component roles. The functional specifications for the components in the CORBA Event Service can be found in [4]. A Typed Push Supplier that intends to connect to and communicate with a Typed Push Consumer in an Event Channel must first use the "TypedSupplierAdmin for_suppliers()" operation to obtain a TypedSupplierAdmin component and then use the "TypedProxyPushConsumer obtain_typed_ push_consumer(in Key supported_interface)" operation to obtain a TypedProxyPushConsumer component. Next, the "Object get_typed_consumer()" operation is used to get the typed push consumer from the TypedProxyPushConsumer component and narrow it to the expected type, the *supported_interface* parameter

in the obtain_typed_push_consumer(in Key supported_interface) operation. Finally, use the "void connect_push_supplier(push_supplier)" operation to actually connect to the typed push consumer. Note that only the typed push consumer offers services required by the typed push supplier, and that a series of negotiations are required to obtain this service. The example shows also that functional service negotiations may be necessary to match a service request with a service provision. This kind of negotiation results primarily from using the aggregation method to form a larger component by composing other components, a point that will be discussed further in the Section 3.

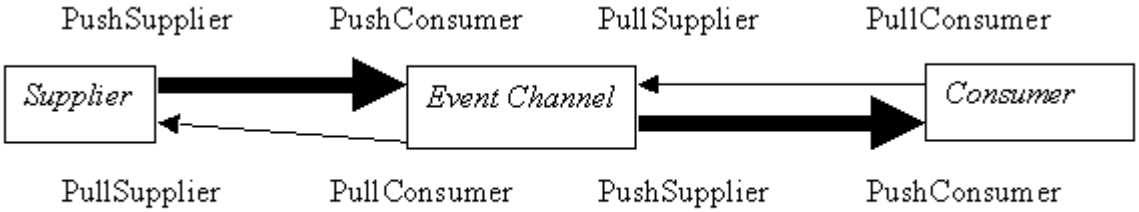


Figure 2 CORBA Event Service Architecture and Component Roles

3. Service Negotiations: functional and nonfunctional negotiations for components in real systems

The functional elements (services and events) and nonfunctional properties (performance, reliability, etc.) that are specified for components during the architecture phase of the software development process reflect only those characteristics that are critical to the system from the perspective of the overall architecture. In order to construct real systems based on a selected architecture, however, components need not only possess the functional and nonfunctional characteristics specified in the architecture, but also those functional elements and nonfunctional properties that arise as specialized requirements in a specific application area. We saw in the previous section that the acquisition of desired functional and nonfunctional characteristics often requires negotiation between components in the system. In the case of nonfunctional properties, these negotiations cannot happen until the implementation phase, when components are to be assembled into real systems.

3.1 Negotiations of Nonfunctional Properties

While desired nonfunctional properties can be specified at the architecture level via components, and constraints on nonfunctional properties may be imposed by an architecture, it is only at the implementation level that a component can concretely state how nonfunctional properties are supported. The manner in which a component implements its support for a nonfunctional property in terms of values offered or expected varies from a single fixed value, to a discrete set of choices, to a continuous range of values. Further, the real system under construction very likely has a variety of concrete QoS requirements that first appear during the implementation stage. To illustrate these points, we revisit the Supplier-Buffer-Consumer example described in the last section, but this time in the context of a real application.

Suppose we are to build a signal processing application based on the Supplier-Buffer-Consumer architecture. Here the Supplier is an analog/digital Sampler component and the Consumer is a Compressor component. The Sampler component offers data at three different sampling rates: 8kb/s, 14kb/s, or 16kb/s. The Compressor component is capable of compressing data at 2.5 ms per frame (32b/frame) if it runs on a 12 MHz microprocessor, 2.125 ms per frame if it runs on a 16 MHz microprocessor, and 1.75 ms per frame if it runs on a 33 MHz microprocessor. Suppose the communication times between the Sampler and the Buffer and between the Buffer and the Compressor are negligible. It is not difficult to calculate that the Compressor data consumption rate is approximately 12800b/s, 15058b/s, and 18285b/s, respectively for the three different microprocessor speeds. The architecture constraint beta >= alpha, where alpha is the data supply rate and beta is the data consumption rate, limits the number of choices for a sampling rate as a consequence of the available compression rates, but may not uniquely restrict the sampling rate to one choice with respect

to the compression rate. For example, the required resources may be offered to the Compressor component enabling it to compress data at a rate of at least 1.75 ms per frame, and thus the Sampler can attain the maximum sampling rate of 16kb/s. The ability to negotiate a higher sampling rate to provide higher signal fidelity may increase the overall QoS offered by the system. If the Compressor component is to negotiate the sampling rate dynamically, the Compressor may find it necessary to ask the Buffer to issue a request informing the Sampler to adjust its sampling rate subject to the architecture constraint beta >= alpha. In this case, the Buffer acquires increased responsibility, and we must implement additional interfaces to support the required negotiations between the Sampler and the Buffer, and between the Buffer and the Compressor.

It is worth noting that it is also possible to have mismatches, or even conflicts, for nonfunctional properties. For example, suppose that a video Supplier produces video at 30 frames per second, while the associated video Player can only consume 29 frames per second. In this case, the constraint beta >= alpha in the Supplier-Buffer-Consumer architecture cannot be satisfied. Such a system may still work, but with considerably degraded QoS (e.g., the Buffer discards a frame per second and, in effect y=x-1). Another example occurs when the range of values offered by one component in support of a nonfunctional property overlaps, but does not coincide, with values expected by another component. In such situations, negotiation to obtain a mutually acceptable value may be necessary.

3.2 Negotiations of Functional Services

The discussion in this section centers around the negotiation of functional services, and how the need for such negotiation arises.

First, the negotiation of functional services may result from the fact that a number of steps are required to match a service expected by one party to a service offered by another party. This may depend simply on how the components involved in the negotiation were developed, e.g., as a consequence of whether they were constructed using either the Containment or Aggregation method of component composition. A component composed from "parts" using the Containment method offers services as a whole and does not expose its internal parts, even though certain services are performed by its parts. Interactions with the component will always go through the component as a whole. On the other hand, a component composed from parts using the Aggregation method offers initial services that will lead to the exposure of services offered by its parts. Interactions with the component will actually be with the part directly once it is exposed. Since multiple parts of an aggregated component may offer the same service, a series of negotiations may be necessary in order to find a right part to interact with. One example of this is the CORBA Event Service architecture discussed earlier. There might be multiple push consumers of the same type in the Event Channel. As shown in the last section, only the typed push consumers offer the services required by the typed push supplier, and a typed push supplier must go though a number of steps to find a typed push consumer. Once it finds a typed push consumer, it interacts with the consumer directly.

A second reason that the need for negotiation of functional services may arise is that a component may offer multiple services that eventually achieve the same purpose, but with different QoS characteristics. For example, a Store might offer two shopping services: one allows negotiation for discounts, but takes longer to deliver, while the other does not offer discount negotiation, but delivers more quickly. Whether a Shopper uses the shopping service with discount negotiation or the one without discount negotiation depends on the Shopper’s desired QoS in terms of cost, quantity of purchase, urgency, and so on. A component that offers multiple levels of similar services, but with different security requirements provides another example of this situation.

A third reason that negotiation of functional services may occur is simply that there may be mismatches between services offered and services expected. Simple cases are syntactic mismatches involving different names, different operation signatures, different orderings, and so on. Syntactic differences are typically easy to bridge. On the other hand, the mismatches may be semantic. A simple example of this situation is when the expected data type is Real, but the service offered returns an Integer type. Another example occurs when the communicating components involved in the negotiation are implemented in different languages on different

platforms. Semantic mismatches may be resolved by employing adapters to bridge the difficulty, or they may not be resolvable at all.

Finally, negotiation of functional services may be necessary because the availability of certain services is unknown until a later time, e.g., until run-time. In this situation, a component must be prepared to negotiate in order to find a matching service provider at run-time.

4. Conclusions

This paper examines definitions of software components and argues that different component concepts arise in relation to the different stages of the software engineering process. We suggest combining Szyperski's general component definition with the details supplied by component models to obtain the right mix of ingredients for defining what a component should be, including what attributes a component should possess, and how component functional elements and nonfunctional properties should be specified and exposed. The paper then focuses on issues related to components at the software architecture level and the implementation level.

For Architectural Components, this paper suggests a concrete approach to describe component functional elements and nonfunctional properties. Component functional services can be separated into services a component provides and services a component requires. Nonfunctional properties correspond to services provided and required, and can be characterized in terms of parameters a component offers or expects. The paper also demonstrates how architecture constraints on component nonfunctional properties can be specified and analyzed. While it is possible that a service provision matches exactly with a service request, service negotiations both in terms of functional elements and nonfunctional properties are necessary in many cases. Service negotiations may be quite complex and may impact the Quality of Service provided by the system. This paper characterizes and demonstrates service negotiations using several examples. It also argues that many service negotiations cannot be carried out until component assembly time or run-time. It shows that the issue of service negotiation is a fundamental consideration in the development of component-based systems.

5. References

[1] Alan Brown, Kurt Wallnau, "The Current State of CBSE", IEEE Software, October 1998, page 37-46.

[2] D. Garlan, R. Allen, J. Ockerbloom, "Exploiting Style in Architectural Design Environments", Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994.

[3] Jun Han, "Characterization of Components", 1998 international workshop on component-based software engineering, Kyoto, Japan, April 25-26, 1998.

[4] OMG, OMG Event Service specification, http://www.omg.org

[5] Guijun Wang, Liz Ungar, Dan Klawitter, "A Framework Supporting Component Assembly for Distributed Systems", Proceedings of the Second Enterprise Distributed Object Computing, page 136-146, San Diego, CA, November 1998.

[6] Guijun Wang, "SoftBean Composer: a Visual Environment for Component Assembly", in Proceedings of the IEEE 14th Symposium on Visual Languages, Halifax, Canada, page 92-93, Sept. 1998.

[7] Guijun Wang, H. Alan MacLean, "Architectural Components and Object-Oriented Implementations", 1998 international workshop on component-based software engineering, Kyoto, Japan, April 1998.

Software Engineering Component Repositories

Robert C. Seacord

Software Engineering Institute

Carnegie Mellon University

Pittsburgh, PA 15213 USA

+1 412 268 7608

rcs@sei.cmu.edu

ABSTRACT

Traditional, large-scale software repositories have historically failed, principally as a result of their conception as centralized systems. New and emerging technologies such as traders, brokers, location services and search engines have yet to be proven effective in the location and adoption of reusable software components. The Component-Based Systems (CBS) Initiative at the Software Engineering Institute (SEI) developed the Agora software prototype to investigate the integration of search technology with component introspection to create a distributed, worldwide component repository. This paper provides a description of Agora, its strengths and shortcomings, and discusses the evolution of component-based software engineering necessary to support an effective component repository.

Keywords

Components, Search Engine, COTS, CBSE, JavaBeans, Software Repository.

Introduction

Agora is a prototype component repository being developed by the Software Engineering Institute at Carnegie Mellon University [1] [2]. The object of this work is to create an automatically generated, indexed, database of software components classified by component model (e.g., JavaBean, ActiveX, CORBA, Enterprise JavaBean). Agora combines introspection with Web search engines to reduce the costs of bringing software components to, and finding components in, the software marketplace.

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. The President's Information Technology Advisory Committee (PITAC) interim report [3] to the President states that:

The construction and availability of libraries of certifiably robust, specified, modeled and tested software components would greatly aid the development of new software.

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).

Search engines are a rapidly evolving Web technology that has the potential to solve the conundrum of a useful component library. Existing search engines provide convenient support for different kinds of Web content. Different search capabilities are provided for different types of content. For example, text content can be searched by simple but effective pattern matching, while images can be searched only by image name.

The AltaVista search service, for example, supports special functions for Web searches. In particular, searches of the format: "`applet:class`" can locate HTML pages containing applet tags where the code parameter is equal to specified Java applet class. For example, a search for "`applet:sine`" can be used to find applets where the code parameter is specified as "`sine`" or "`sine.class`". While this approach has obvious advantages of simply searching for the term, it still only allows the name of the components to be indexed and searched.

The Agora search engine enhances existing but rudimentary search capabilities for Java applets. By using Java introspection, the Agora search engine can maintain a more structured and descriptive index that is targeted to the type of content (the component model) and the intended audience (application developers) than is supported by existing search engines. For example, information about component properties, events, and methods can be retrieved from Agora.

Issues, Tradeoffs & Future Directions

Agora was designed and implemented to demonstrate the feasibility of a component repository using existing infrastructures and available information. Agora demonstrates the extent to which an automatically indexed, database of software components can be implemented given the current state of the practice. Agora does not address all the issues that need to be resolved before such an approach can be effectively used on a broad scale. Some of the tradeoffs between the approach taken by Agora and more traditional repositories are discussed in this section. Areas in which component-based software engineering may evolve to more effectively support component repositories are identified.

Modeled

The PITAC report stipulates that components available in a repository must be fully *modeled*. This may mean that a behavioral model of the component has been developed. Alternately, it could mean that the component adheres to a predefined component model.

A component model describes the coordination model used by subscribing components so that they may be seamlessly integrated into a system that applies the model. The most prominent component technologies including Enterprise JavaBeans, and ActiveX all impose constraints on components [4]. Existing CORBA servers do not really meet the requirements of being a component model, but are still of interest as the OMG is planning on adopting a component model as part of the CORBA 3.0 specification.

Agora was designed to support search and retrieval of multiple component models, but only JavaBeans were fully developed. Some experimentation with CORBA was attempted, but results were not promising due to the difficulty in locating and introspecting CORBA servers. A component repository should be able to locate, index and retrieve a broad variety of components.

Traditional repositories often collected software products, language specific subroutines, or link-able libraries. In removing sources of architectural mismatch [5], evolving component models should improve the effectiveness of the software repository concept.

Interface Descriptions

An obvious problem or limitation of Agora is the lack of descriptive information about the component's interface. In the case of JavaBeans, for example, information that can gathered through introspection is principally restricted to method signatures including function names, return and parameter types. In other component models, such as CORBA servers, interface information is maintained externally to the component and may not be available at all. In all cases, descriptive information about the overall purpose of each component and the various APIs is lacking. In addition, information about functional semantics is completely absent.

Traditional repositories are better positioned to provide interface descriptions than Agora, by documenting the interfaces according to some standard format used by the repository or making the original component documentation available within the repository.

The PITAC report stipulates that components available in a repository must be fully specified. Existing component models do not provide a specification that is sufficiently detailed to allow programmers to take advantage of the component without reference to further documentation or excessive experimentation.

JavaBeans, for example, supports introspection to determine the signatures of the class methods but does not provide a description of the semantics of the calls. The semantics of the API calls is normally described, albeit in an informal manner, in documentation for the API. Although this documentation is not available in a JavaBeans' executable form, the information is often available in the source code in the form of structured comments. The `javadoc` tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing (by default) the public and protected classes, inner classes, interfaces, constructors, methods, and fields. This information could also be converted by a doclet into a runtime accessible format.

As demonstrated in WaterBeans [6], component models could also provide a means of including contact information so that the original author(s) may be contacted.

Quality Assurance

The quality of components in traditional software repositories is often assured by the organization that maintains the repository. There are number of problems with this approach:

1. Timeliness – having a single organization responsible for providing quality assurance for every version of very component created is an impossible goal. Taking this approach instantly creates a bottleneck, restricting the number of components that can be incorporated into the amount of time it takes to get a component reviewed and incorporated into the repository.

2. Lack of context – components must be qualified as being able to perform a specific task. A component that is qualified to perform a certain function may not be suitable, for example, for a real time application.

3. Bias – it is highly unlikely that the organization maintaining the component repository is completely unbiased. Most organizations that might be capable of providing this service have their own products, customers, and strategic relationships and partnerships. It would be naïve to assume these relationships will not impact, at the very least, the components that are included in the repository.

The Agora model is based on the premise that component databases need to be free and inclusive. Value-added industries such as consumer reports and underwriter labs can add value by providing independent quality assurance of popular components. The existing Agora prototype would be extended to allow underwriter labs to link product evaluations to specific components maintained within the repository. Again, this process will be handled in a completely decentralized fashion. Potential consumers can review these reports and form their own opinions as to the reputation of the organization providing the information and the value of the report. It may be also possible to automatically generate mailing lists on a per component basis to let consumers of that component directly exchange experiences.

Components in the repository (or elsewhere) can be digitally signed to indicate that the provide specific quality of

service attributes, for example that they are guaranteed to execute in a specified period of time (for use in real time systems) or that the component has completed some battery of tests. The objects can be signed directly by the certifying agency. Providing tamper-proof packaging will increase the level of trust of consumers that the component being evaluated does in fact have the certified qualities.

Open System

Traditional component repositories were conceived as centrally managed systems. This allowed the group or organization maintaining the repository to certify the degree to which components in the repository were robust, specified, modeled and tested. Without central management, it is very difficult to ensure the quality of the components in the repository.

In the implementation of the Agora prototype, it was felt that component repositories need to be, at first, free and inclusive. Agora automatically compiles indexes by going out over the Internet and discovering and collecting information about software components. Component collection is performed in a nonjudgmental manner, so the problem of having a sole arbiter decide what does and does not belong in the repository is eliminated.

The traditional, centralized approach is analogous to a centrally planned economy in that both are slow to respond to market changes and often succeed only when the investment outweighs the benefits. The free and inclusive approach can be compared to a market driven economy that is more responsive to market realities but does not provide the safeguards of the more rigidly planned system.

Component Uniqueness

A goal of a component repository is to incorporate each component once and only once, but this is easier said than done. There are a number of different ways to determine if a component is unique. Here are a sample of methods that can be used to determine uniqueness and the associated drawbacks:

1. **The component has a unique URL.** Unfortunately, multiple identical copies of a component often appear at multiple locations so this approach does not guarantee a component only appears once in the database.

2. **The component has the same application programming interface (API).** Since the API can be introspected, it can be compared with other APIs as a test for uniqueness. However, multiple different versions of a component could easily have identical interfaces. Another potential issue is if different versions of the same component should be considered to be different components or not.

3. **The component matches byte for byte with an existing component.** This is a relatively restrictive test for uniqueness although there are still potential problems. For example, an Enterprise JavaBean may be modified at deployment time to support specific characteristics defined in the deployment descriptor. It is therefore possible that the same Enterprise JavaBean deployed in one environment will not match the same Enterprise JavaBean deployed elsewhere.

Component models should have a means of specifying a major and minor version number in the component. This could provide a useful mechanism for differentiating two components that otherwise have the same API.

Data Rights and Privacy

Agora automatically collects components discovered over the Internet using a spider. This raises some interesting questions regarding data rights and privacy.

Spiders follow HTML links in existing documents. Most of these documents have been placed in locations where they are accessible to the public. There are probably some cases where confidential or proprietary documents have inadvertently been made accessible to the public but this is an exceptional occurrence.

Java classes are often included on web pages to increase the dynamic content of the page. In most cases the

components are not provided for the express means of collection and integration by other system integrators.

Although the source code used to generate these components can be protected under copyright laws, there is no defined mechanism for protecting the binary copies of a program other than preventing copies from being distributed or integrating some manner of licensing software.

ActiveX controls, for example, can be built to work only at design time, runtime or in either situation using LPK files.

Electronic Commerce in Components

Closely related to the problem of data rights and privacy of components is electronic commerce in components. Most publicly accessible components are freely available, non-proprietary, and non-commercial. However, because of the degree of investment required, many types of components may only be available commercially. However, vendors of commercial components are very unlikely to make these components publicly available unless they have some means of protecting their investments.

There are a number of potential solutions for this problem. One solution would require the development of an electronic commerce model where components could be "rented". An initialization call could, in fact, provide a credit card number that can be electronically authorized (in a similar fashion to any retail store). Subsequent calls to the component could be charged against the credit card. Depending on the expected calling frequency, this might require the use of *nano-bucks* – extremely small measures of currency.

Another solution would be that commercial companies make skeleton versions of components publicly available as a means of advertising the features of the component. The system integrator would then need to contact the vendor to license the component prior to employing it. ActiveX controls, for example, could be built to work only at design time. This would allow the controls to be indexed by an automated component repository while the development organization retained control of component distribution.

It is easy to imagine other schemes that could also be employed to achieve similar results.

Software Engineering

Introspection, as defined by the JavaBeans specification, provides a means for development tools, such as the BeanBox, Borland's Jbuilder™, IBM's Visual Age® for Java and Symantec's Visual Cafe to discover component interfaces at runtime. This allows developers to integrate components within a development environment without having to "teach" the development tool about the component.

This same introspection capability made possible the indexing of interface information by Agora. A Component repository can be thought of as a software engineering tool used by system integrators to develop component-based systems. This is relevant because any enhancement to existing component models to support component repositories will generally benefit the broader class of development tools. For example, extending the JavaBean component model to support the description of the API would allow development tools such as the BeanBox to provide on-line documentation for API calls at run-time to assist the developer in the integration of the component.

Location Services

In general locating components in the Agora model is problematic. In developing Agora, we found a ready supply of JavaBeans that have been used as applets on World Wide Web pages. These applets can then be easily found by normal spidering techniques.

JavaBean components found by this method are typically used to add dynamic behavior to static HTML-based Web pages. While this is a potentially useful area of components to be made available in a component repository, it does not represent the full range of JavaBeans that may be available. While it is expected that this Web-based method could also be used to successively retrieve ActiveX components, it may be less effective in finding CORBA servers,

Enterprise JavaBeans, or other component types.

CORBA defines multiple, competing mechanisms for locating CORBA servers including implementation repositories, location services, naming services and Object Trader Services. Implementation repositories and location services are not generally useful outside of specific subnet. Object Trader Services require that components be registered, a process that often requires fees. Locating CORBA objects in a naming service turned out to be problematic for several reasons. First, the majority of CORBA servers do not store their object references in a naming service. Second, even if they did, there is no good bootstrapping process for finding an initial object reference for the naming service. This problem could be addressed by having naming services respond to queries on a well-known standard port number or providing some sort of meta-naming service. The best opportunity to discover a naming service is to look for them at vendor-supplied default port numbers.

Agora currently provides a means for component developers to register their components at the Agora web site. This allows components that cannot be located using existing location techniques to be included in the repository.

Summary and Conclusions

The issues associated with developing a useful and effective software repository are distributed across a range of technology areas.

Existing component models need to be extended so that additional information about the component can be accessed at runtime, particularly a description of the semantics of the API calls and the purpose of the component.

Existing naming and directory services need to be standardized, so that automated search tools can search well known port numbers to find and access registered components.

Component developers need to take advantage of existing capabilities (such as the naming service in CORBA) and integrate enhancements to these component models as they become available.

It is unlikely that these varied technology areas will all converge on useful solution to the software repository problem without education and direction from a central group advocating the establishment of these software engineering repositories.

AcknowledgEments

Neil Christopher at the National Institute of Standards and Technology Manufacturing Engineering Lab (NIST/MEL) sponsored this work. Special thanks to Kurt Wallnau and Scott Hissam who helped develop the ideas and the Agora prototype and reveiwers Dan Plakosh, Chuck Buhman and John Foreman.

REFERENCES

1. Seacord, R., Hissam, S., and Wallnau, K., "Agora: A Search Engine for Software Components." IEEE Internet Computing, 2: 6, November/December, 1998, pgs. 62-70.

2. Seacord, R., Hissam, S., Wallnau, C., Agora: A Search Engine for Software Components, CMU/SEI-98-TR-011, August 1998.

3. President's Information Technology Advisory Committee Interim Report to the President, National Coordination Center for Computing, Information, and Communications, Arlington, VA, August 1998.

4. Alan Brown, Kurt Wallnau, The Current State of Component-Based, Software Engineering (CBSE) IEEE Software, September 1998, pg. 37-47.

5. Garlan, D., Allen, R., and Ockerbloom, J., Architectural Mismatch: Why reuse is so hard. IEEE Software, November 1995.

6. Plakosh, D., Smith, D., Wallnau, K., Building a Custom Component Model: Concepts and Application to Water Quality Models, to be published as an SEI Technical Report, Spring 1999.

An Approach to Software Component Specification

Jun Han
Peninsula School of Computing and Information Technology
Monash University, Melbourne, Australia

Abstract. Current models for software components have made component-based software engineering practical. However, these models are limited in the sense that their support for the characterization/specification of software components primarily deals with syntactic issues. To avoid mismatch and misuse of components, more comprehensive specification of software components is required, especially in a scenario where components are dynamically discovered and used at run-time over corporate intranets and the Internet. Our approach to software component specification aims at comprehensive interface modelling/packaging for software components. It deals with the semantic, usage, quality as well as syntactic aspects of software component specification.

Introduction

Software systems form an essential part of most enterprises' business infrastructure, and become increasingly complex. In today's global market, these enterprises have to continuously adjust and improve their business practices to maintain a competitive edge. Such changes to business practices often raise requirements for change to their underlying software systems and the need for new systems, which have to be fulfilled in a timely fashion. It is in this business context that being able to assemble or adapt software systems with reusable components proves vital.

We have seen examples of integrating software components or packages into systems to achieve specific business objectives of enterprises. Perhaps, the most prominent is the use of Commercial-Off-The-Shelf (COTS) software packages in enterprise systems. Experience has shown that even with advanced technological support, in general, it is not an easy task to assemble software components into systems. A major issue of concern is the mismatches of the components in the context of an assembled system, especially when the mismatches are not easily identifiable [Garlan et al, 1995]. The hard-to-identify mismatches are largely due to the fact that the capability of the components are not clearly described or understood through their *interfaces*. Most commercially available software components are delivered in binary form. We have to rely on the components' interface description to understand their*exact* capability. Even with the components' development documentation available, people would certainly prefer or can only afford to explore their interface descriptions rather than digesting their development details. Furthermore, interface descriptions in natural languages do not provide the level of precision required for component understanding, and therefore have resulted in the above mentioned mismatches. When discovering components and assembling systems at run-time over corporate intranets and the Internet, it becomes a must that the components have precise and even comprehensive interface descriptions.

Most current approaches for component interface definition deal with primarily syntactic issues, like those of the CORBA Interface Definition Language (IDL). To gain a clear understanding of

a component's exact capability, other essential aspects of the component should also be described, including the semantics of the interface elements, their relationships, the assumed contexts of use, and the quality attributes. Our approach to software component specification deals with these aspects through comprehensive interface specification. It has been developed and applied during a large-scale telecommunications R&D project at a multi-national company. It provides not only the basis of notational and tool support for software component specification, but also the basis of methodological guidance for architecture-directed and component-based system development, composition and integration.

While our framework highlights comprehensive packaging, it is unrealistic to expect that every component is to be defined formally and comprehensively in practice. For example, JavaBeans and COM components are still very useful even though their interface definitions are mainly syntactic. In these cases, the full understanding of the components has to rely on other means, e.g., informal documents. Or, the user of the component is happy with the partial information that has been offered by the interface. It is very important to allow such flexibility in packaging software components. It is the component user who decides whether the provided interface information is enough to warrant the adoption of the component in his/her use context. This is particularly true for the quality attributes of the component. We generally refer to this flexibility as "sliding characterization/specification".

An analysis of existing industrial component models and the need for comprehensive component characterization can be found in [Han, 1998a]. In the following sections, we give a brief account of our approach to software component specification. Further details can be found in [Han, 1998b; Han and Zheng, 1998].

An overview of the approach

As argued earlier, proper characterization of software components is essential to their effective management and use in the context of component-based software engineering. While there have been industrial and experimental projects that build systems from (existing) components, the approaches taken are ad hoc and heavily rely on the specifics of the systems and components concerned. That is, component-based software engineering is still very much in its infancy. Characterization of components through comprehensive interface specification is a step towards systematic approaches to CBSE and their enabling technologies.

Our approach to component specification aims to provide a basis for the development, management and use of components. It has four aspects. First, there is the *signature* of the component, which forms the basis for the component's interaction with the outside world and deals with the syntactic aspect of the necessary mechanisms for such interaction (i.e., properties, operations and events). The next aspect of component specification concerns the semantics of the component interaction, including the semantic specification of individual signature elements and more importantly additional *constraints* on the component signature in terms of their proper use. The component signature and its semantic constraints define the overall capability of the component. The third aspect of component specification concerns the *packaging* of the interface signature according to the component's roles in given scenarios of use, so that the component interface has different *configurations* depending on the use contexts. The fourth aspect of component specification is about the characterization of the component in terms of their

non-functional properties or *quality attributes* (code named *illities* [Thompson, 1998]).

Interface signature

Fundamental to a component's interface is its signature that characterizes its functionality. The component interface signature forms the basis of all other aspects of the component interface. As commonly recognised, the interface signature of a component comprises *properties*, *operations* and *events*. A software component may have a number of properties externally observable. These properties form an essential part of the component interface, i.e., the observable structural elements of the component. The users (including people and other software components) may use (i.e., observe and even change) their values, to understand and influence the component's behaviour. A common use of component properties is for component customisation and configuration at the time of use. It should be noted that certain component properties can only be observed, but not changed.

Another aspect of a component signature is the operations, with which the outside world interacts with the component. The operations capture the dynamic behavioural capability of the component, and represent the service/functionality that the component provides. Besides proactive control (usually in the form of explicit operation invocation or message passing), another form of control used to realise system behaviour is reactive control (usually in the form of event-driven implicit operation invocation or message passing). It is often the case that certain aspects of a system are better captured through proactive control via operations, while other aspects of the system are better captured in the form of reactive control via events. To facilitate reactive control, a component may generate events from time to time, which other components in the system may choose to respond to. In this type of event-based component interactions, there may be none or many responses to an event, and they may change as time goes on. As such, this model of interaction allows communication channels to be established dynamically, and gives the system the capability of dynamic configuration.

In our approach, the specification of interface signature takes a form similar to the current Interface Definition Languages found in CORBA and Java, including assemble-time and run-time properties, operations and events.

Interface constraints

The signature of a component interface only spells out the individual elements of the component for interaction in mostly syntactic terms. In addition to the constraints imposed by their associated types, the properties and operations of a component interface may be subject to a number of further semantic constraints regarding their use. In general, there are two types of such constraints: those on individual elements and those concerning the relationships among the elements. Examples of the first type are the definition of the operation semantics (say, in terms of pre-/post-conditions) and range constraints on properties. There are a variety of constraints of the second type. For example, different properties may be inter-related in terms of their value settings. An operation can only be invoked when a specific property value is in a given range. One operation has to be immediately invoked after another operation's invocation.

The explicit specification of semantic properties are important. First of all, they form part of the

defining characteristics of the component. They make more precise about the capability of the component. Furthermore, it is essential for the user of the component to understand these constraints. Only then, proper use of the component can be guaranteed and therefore the composed system's behaviour is predictable. Without such constraints, the proper understanding and use of the component will be much harder. Informal and possibly incomplete and imprecise documentation has to be relied on. While we all know the dangers and problems associated with such a scenario, it has even greater significance for component based software engineering. This is because the interface definition of a component may well be the only source of information for the component as we may not have access to its source code or any other development documentation, e.g., in a scenario where components are discovered and used dynamically at run-time.

The use of pre-/post-conditions for defining operation semantics has been well studied, such as those used in Eiffel [Meyer, 1997] and Catalysis [D'Souza and Wills, 1998]. Our approach focuses on constraints concerning the relationships among signature elements. A common example in telecommunications systems is that a system module has to be initialised through a sequence of operations before it is enabled to accept normal requests (e.g., invocations of further operations). Specific mechanisms are available to specify this type of constraints in our approach.

Interface packaging and configurations

The signature and the semantic constraints of a component define the overall capability of the component. For the component to be used, certain packaging is required. It involves two aspects: (1) the component plays different roles in a given context, and (2) the component may be used in different types of contexts. In a particular use scenario, a component usually interacts with a number of other components, and plays specific roles relative to them. The interactions between the component concerned and these other components may differ depending on the components and their related perspectives. When interacting with a particular type of component from a specific perspective, for example, only certain properties are visible, only some operations are applicable and some further constraints on properties and operations may apply. More specifically, for example, the value range of a property may be further restricted in a particular role. In general, this suggests the need for defining perspective/role-oriented interaction protocols for a given component, i.e., *an interface configuration*, as the effect of interface packaging. Since the role-based configuration definition is oriented towards component interaction, a role-based interface of a component should include not only what the component provides but also what it requires from the other end (another component) of the interaction.

Scenarios provide the contexts of use for a component. A component may be used in different scenarios and has different role partitions in these scenarios. For a component, therefore, there may be the need for different sets of interaction protocols, with each set for a scenario in which the component is to be used. This suggests that a component may have different interface configurations. In principle, an interface configuration should be defined in terms of both the component and the use scenario, and it relates the component to the use context.

Usually, when a component is designed, the designer has one or more use scenarios in mind. Therefore, a few packaging configurations may be defined for the component interface. When a

new use scenario is discovered, a new packaging configuration may be added. It should be noted that the packaging of an interface configuration is subject to the component's underlying capability as defined by the component's signature and semantic constraints and will not alter this defined capability.

The importance of interface packaging can not be over emphasised. It serves to relate the component to a context of use. In fact, much of the requirements for the component is derived from the use scenarios. The roles that a component plays in a use context are vital to the architectural design of the enclosing system. It provides the basis for defining the interactions between the components of the system and realising the system functionality. It enables the relative independent development of the system components with clearly defined interfaces as well as requirements.

In our approach, mechanisms are provided for specifying interface configurations and roles within configurations. Additional constraints about component interaction can be specified in association with roles and configurations.

Quality attributes (illities)

Another aspect of a component is its non-functional properties or quality attributes, such as security, performance and reliability. In the context of building systems from existing components, the characterization of the components' illities and their impact on their enclosing systems are particularly important because the components are usually provided as blackboxes. However, there is not much work done in this area. Therefore, there is an urgent need to develop the various illity models in the context of software components and composition. For a particular quality attribute, we need to address two issues: (1) how to characterize the quality attribute for a component, and (2) how to analyse the component property's impact on the enclosing system in a given context of use (i.e., in the context of a system architecture). A related issue is whether the characterization of the quality attribute will change in different contexts of use. Currently, we are investigating the security properties of software components and their impact on system composition in the context of developing distributed electronic commerce systems [Han and Zheng, 1998]. In general, the interface definition of component illity characterization will be dependent on the specific characterization models developed. While we do not have definite models available yet, the component specification framework proposed in our approach can be extended to accommodate new models concerning quality attributes.

Summary

In our approach to software component specification, the properties, operation and events form the signature of the component interface. The constraints further restrict and make precise the definition (and hence the usage) of the component interface. The signature and the constraints characterize the component capability. The configurations are based on the component use scenarios, and define specialised usages of the component. A configuration identifies the roles and defines the role-based interfaces of the component in a given use context. The component's non-functional properties are useful in assessing the component's usability in given situations and in analysing properties of the enclosing systems. In general, the proposed framework provides the basis of notational and tool support for component-based system development, composition

and integration. It also contributes to the standardisation of software component specification and its infrastructural support.

Our approach to component specification has been developed and applied in the context of a real-world industrial project that concerns the development of a telecommunications access network system involving software and hardware codesign. Combined with object oriented analysis techniques such as scenario analysis, the approach had been used in the system's architecture design. Immediate benefits of using this framework have been the clear definition of the subsystems/modules' capabilities through their interfaces, the clear identification of the interactions between the modules, and the analysis of system behaviour at architectural level. This has significantly reduced the interaction between the teams responsible for the various modules, and avoided many of the architectural changes later in the development cycle that had been experienced in earlier projects.

References

1. D. D'Souza and A.C. Wills, 1998. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley.
2. D. Garlan, R. Allen and J. Ockerbloom, 1995. Architectural Mismatch: Why reuse is so hard. *IEEE Software*, 12(6): 17-26.
3. J. Han, 1998a. Characterization of components. In *Proceedings of 1998 International Workshop on Component-Based Software Engineering*, Kyoto, Japan, pages 39-42.
4. J. Han, 1998b. A Comprehensive interface definition framework for software components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, Pages 110-117. IEEE Computer Society Press.
5. J. Han and Y. Zheng, 1998. Security Characterisation and integrity assurance for software components and component-based systems. In *Proceedings of 1998 Australiasian Workshop on Software Architectures*, Melbourne, Australia, pages 83-89.
6. B. Meyer, 1997. *Object-Oriented Software Construction*, 2nd edition. Prentice Hall.
7. C. Thompson. *Workshop on Compositional Software Architectures: Workshop Report.* http://www.objs.com/workshops/ws9801/report.html, Monterey, USA.

A Hierarchical Technique for Composing COM based Components

Author: Chris Peltz

Microsoft Corporation

Microsoft Research – Advanced Applications (ComApps)

chrispe@microsoft.com

Last Updated: March 16, 1999

Abstract: This is a position paper in response to a call for participation from the Second International Workshop on Component-Based Software Engineering. This paper discusses a technology currently under development to combine arbitrarily complex component structures. We call these structures Assemblies.

It will be useful to have knowledge of Microsoft COM technology. This can be obtained from documents available at http://www.microsoft.com/com/dcom.asp.

The Problems of Component Composition

The concept of software *components* comprises a spectrum of different technologies. Some people think of components as the pieces of large multi-tier distributed systems, such as business objects. Each of the components in these systems is rarely composed of many other smaller components. On the other end of the spectrum, components are used to build all kinds of software not just distributed systems.

At the very extreme end of that spectrum is the idea that components are small (from 100-4000 lines of C) and are composed in a hierarchical fashion to achieve larger and more functional sets of components. This is where the ComApps research team is investigating.

While there are several facets to our research, this position paper only discusses our efforts in solving some of the problems inherent in the hierarchical composition of larger components from smaller components.

In our research, we call a composition of a set of components an *assembly.* Components not composed of other components are called *elements*. In this document, the term *component* refers to both assemblies and elements. A pointer reference from one component to another component is called a *connection.*

In researching the composition of COM components, several problems have been recognized and need to be solved.

1. **Tight Coupling** -- We want to avoid having the connections force a tight coupling between components. Component class A shouldn't be tied to only allowing a connection to objects of class B. This limits reuse. COM's QueryInterface mechanism allows loose coupling between any two elements. The problem becomes more difficult between an assembly and an element, or between two assemblies. In these cases, the assembly object is composed of many different components. QueryInterface only works for a single object identity, not multiple identities.
2. **Versioning Connection Relationships** -- We want to avoid having the assembly mechanism enforce specific semantics on connections. The composition mechanism must allow the components to negotiate the proper relationship between themselves. For instance, consider a component A that can deal with different backward compatible versions of a component B, say B-old and B-new. The components B-old and B-new both implement an interface IBar. However, B-new has extended functionality that is exposed via IBaz. The composition mechanism must allow A to have a fully functional connection to B-old as well as B-new.
3. **Component lifetime management**. – We need to be able to determine when an assembly of components can finally be shut down. Within an assembly and across assemblies there are connections being made between components. Some of these connections may actually cause circular references (that is, object A is holding onto B and B is holding onto A and the objects can't be destroyed until all references are zero). The composition mechanism must provide a solution for all circular references.
4. **Specifying Assemblies** – We need a way to allow the programmer to specify assemblies of components. These assemblies may contain complex connections between components within the assembly as well as connections to components outside of the assembly.

The following sections of this document comprise an overview of the component assembly mechanism and how it addresses these problems.

Components

As mentioned in the previous section, there are two types of components: Assemblies and Elements. Connections are managed via a mechanism called *connectors*. The following sections describe each of these concepts.

Assemblies

How does a programmer construct a large reusable component from smaller components? In COM,

aggregation is one such mechanism, but it suffers from some limitations:

- Aggregation can only construct single objects; there is no direct support for part-whole hierarchies. The programmer is responsible for instantiation and management of subordinate objects, with no help.
- The aggregating component must manage the lifetimes of its parts.
- Internal references between parts within the aggregate have special QueryInterface and reference counting rules that need to be followed. This adds complexity and confusion.

In aggregation, the outer component may include code for the semantic behavior of the composite, as well as code for instantiation of parts and lifecycle management. In particular, the outer component knows the classes of parts it will instantiate. This means that if you want to reuse the composite, but modify it to use a different part, then the source code for the outer component must be modified. This limits reuse.

An alternative approach is to implement a part-whole hierarchy, where an object is responsible for managing the objects immediately below it in the hierarchy. As in aggregation, this model puts the code for instantiating and managing the structure into the same components as the code that provides the semantic behavior of (that level of) the composite. Configurations other than pure hierarchies have added complexity in management (e.g. cyclic references or confusion over ownership). Management of the structure becomes one of the hardest parts of programming.

An *assembly* is a connected group of components (including sub-assemblies) that enables connection to its elements by other assemblies, through *connectors (See* Connectors*)*. Assemblies are then candidates for reuse, as well as the constituent components. Connectors provide encapsulation and polymorphism for connecting to components. In addition, assemblies can be parameterized, to give the effect of a framework where you specify one or more partial behaviors. Assemblies thus support hierarchical construction of arbitrarily large composites.

The assembly mechanism separates code for instantiation and connection of components (the *structure-management*) from the code for the semantic behavior of the composite (*the run-time behavior* of the components). Separating the structure-management from run-time behavior has the advantage of making sure that ordering of connections is independent from the component behavior. The component can easily be re-used within many different assemblies by merely connecting different instances without the worry of ordering dependencies of connections within the bounds of each assembly. Ordering dependencies between connections can limit reuse.

We further factor the lifecycle management problem into two parts:

- a lifetime boundary wrapper that detects when there are no references to any element in the assembly (See Assembly Lifetime Management)
- a generic assembly run-time wiring engine which knows how to cleanly disconnect and shut down all the components. This solves the cyclic reference problem, and provides one clear owner for the components independent of the kind and number of composite behaviors that use the components. (See COMCAD Diagrams for Specifying Assemblies)

Elements

Elements are simply COM objects. They may or may not expose connectors. They typically do not create other components, although some may instantiate helper components such as collections.

Connectors

Connections within an assembly are instigated by the assembly. The assembly mechanism will tell one of the components involved in the connection to perform the connection. For instance, the connection between components A and B is actually made by A and not the assembly. Component A can then use QueryInterface to enhance the connection. For instance, consider a connection that is characterized by component A wanting to refer to component B’s IBar interface. Component A can extend the meaning of

that connection to optionally include IBaz by simply doing a QueryInterface from IBar to see if IBaz is also supported.

Connectors (sometimes referred to as *Unary Connectors*) are interfaces that facilitate the connection between two components. A connector is implemented by at least one and possibly both components that participate in the connection. Any given connector on a component can specify that the component implements one or more specific interfaces (called connector *out pins*). Additionally, a connector can specify one or more interfaces the component wants to use (called connector *in pins*). The in pins and out pins characterize a given connector implementation. The actual connection that takes place however is performed by the component that implements the connector in pin.

Different out pins on a single connector are not expected to belong to the same object identity, so it is not expected that a call QueryInterface from the interface on one pin, will result in the interface on another pin. This contract is necessary since assemblies are composed of multiple sub-components, connectors are used on assemblies to expose interfaces from the components within.

Its also interesting to note that since connectors are interfaces, entire connectors can be used as the pin of a different connector. In this way, complex connections can be made across assembly borders.

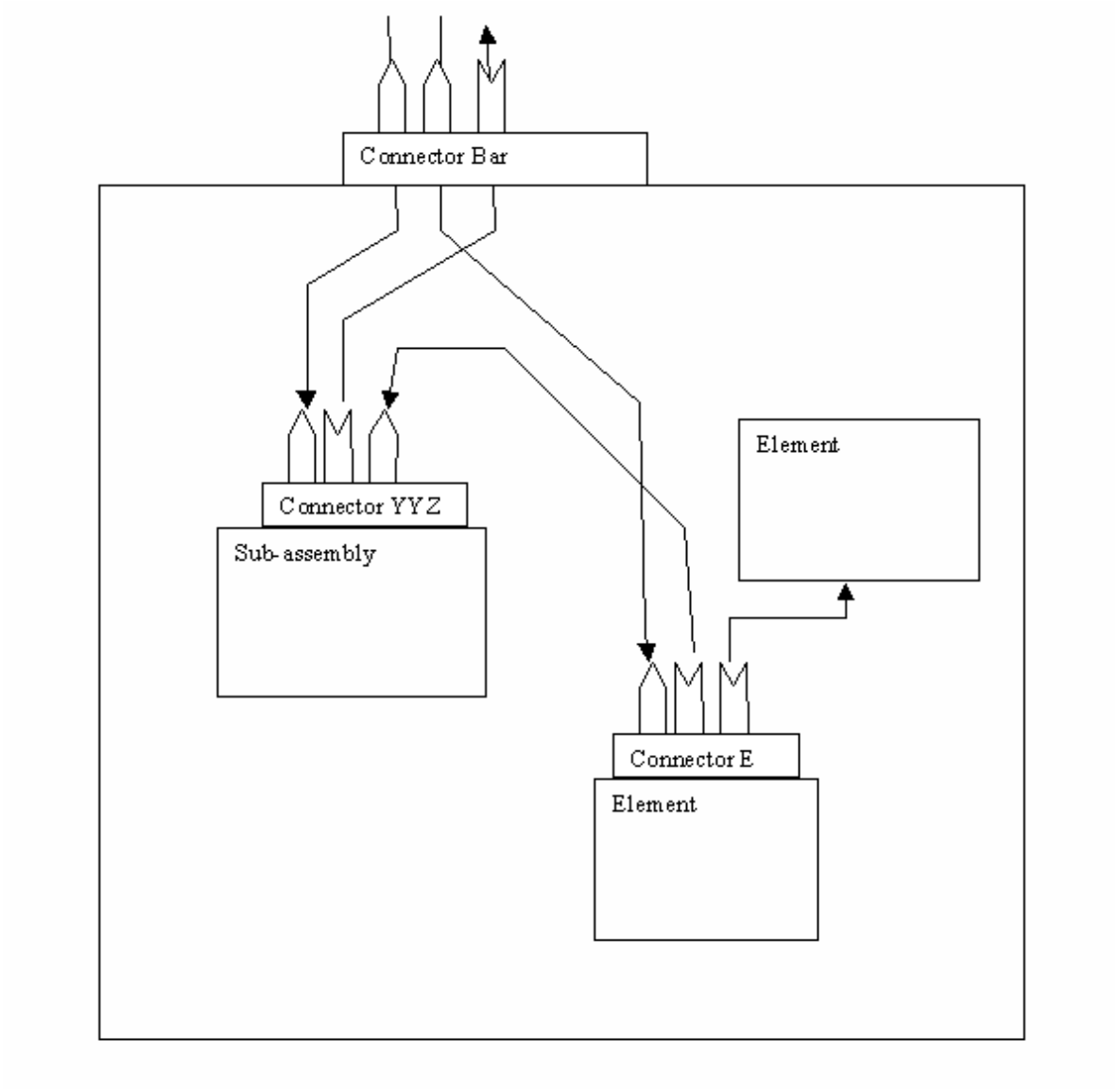


Figure 1 , An assembly. The out pins on the connectors represent interfaces implemented by the component. The in pins on the connectors represent the interfaces used by the component, the arrows represent actual connections that point toward the implementer of a particular interface.

Any component may expose multiple connectors. In this way, COM components have static, polymorphic type definitions to facilitate the wiring of components at multiple levels of assembly. Each connector can be considered a specific type contract that extends the individual interface type contracts of its pins.

Essentially, connectors are used to expose parts of an component's sub-structure. It is an encapsulation mechanism.

Two COM interfaces are used to implement connectors on components:

- **IConnectorProvider** – this interface is implemented on the component that supplies a connector interface, typically an assembly. It has one method "GetConnector" on it to retrieve a GUID named IConnector interface from the component. Note that the IConnector interface is not expected to be on the same object instance as IGetConnector.

```
interface IConnectorProvider : IUnknown

{

HRESULT GetConnector (REFCONNECTORNAME conname, REFIID riid,
[out, iid_is(riid)] void **ppvConn);

}
```

- **IConnector – this interface is implemented by a component or by a component within an assembly. Actually this is a templated interface where specific connector types are trivial derivations of IConnector, The derivations are used in order to specify a correct type-contract regarding which pins are present on the connector.**

```
interface IConnector : IUnknown

{

HRESULT GetElement (LONG dwOutPinId, REFIID riid, [out,
iid_is(riid)] void **ppvElement);

HRESULT Connect (LONG dwInPinId, IUnknown *punkOther);

.

.

.

}
```

The interesting methods on this interface are:

- "Connect" indicates a new reference should be made for the given in pin. A generic assembly-wiring engine (See COMCAD Diagrams for Specifying Assemblies) will call this to both connect and disconnect wires between components.
- "GetElement" is used for retrieving an interface reference to the component on the other side of the connection, i.e., the element with the corresponding out pin on the other side of the wire. If the

other side of the wire is not an out pin, then QueryInterface is used to retrieve the necessary interface, rather than GetElement.

Binary Connectors

Binary connectors are a specific flavor of connector that differs from the Unary Connector. They are still driven by an assembly mechanism to perform connections.

Using Unary Connectors, two objects are wired together within an assembly via a connector interface implemented on the object that has the connector in pin.

Binary Connectors enable a third party connection mechanism where neither object contains the actual code to perform the connection. Binary connectors are used:

- when two objects need to be connected in a complex way
- when a component is introduced to an assembly that doesn't use the connector interfaces to facilitate connections… (such as legacy COM components)

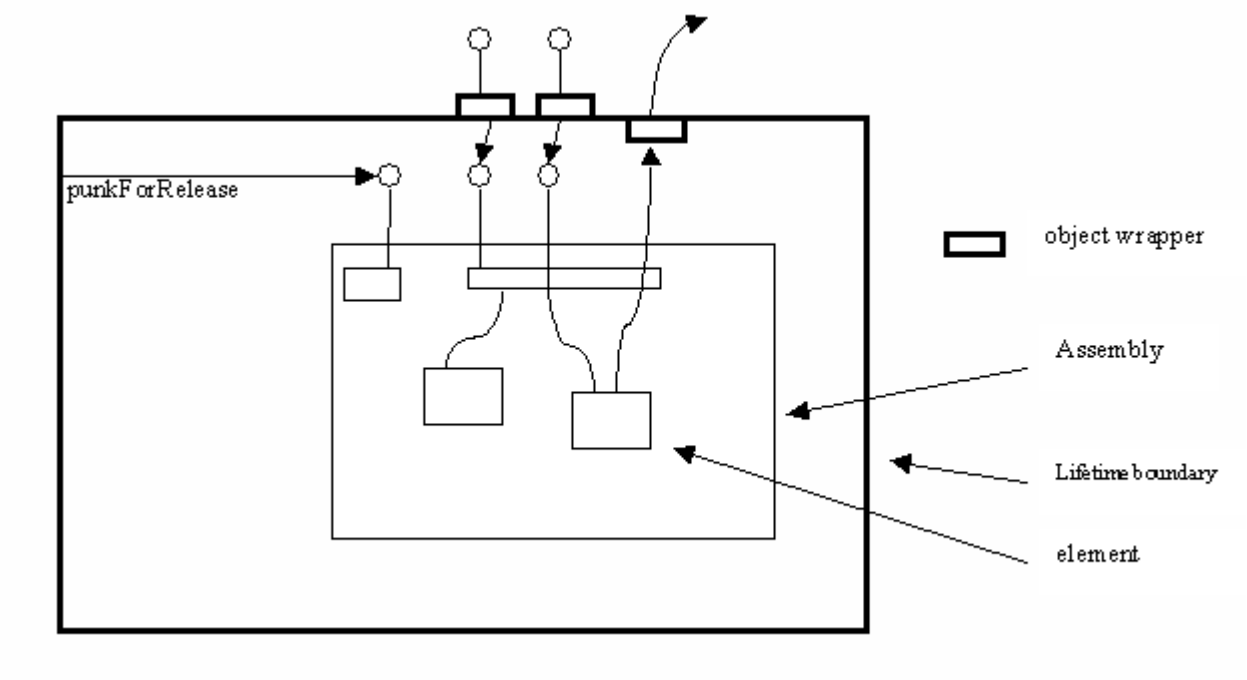
Ordering of Connections within an Assembly

Within any given assembly, it is required that the order in which connections are made to elements is immaterial. Components should never be designed or built with any assumptions about connection order.

Assembly Lifetime Management

The lifetime of an assembly is dependent on the assembly object from which it was created. The lifetime wrapper boundary is used to enable all elements of the assembly to contribute their reference counts to the assembly's lifetime. All references going across the boundary are tracked in order to determine when the assembly should be shut down. The assembly can do this since the assembly knows what connections are both internally and externally.

The assembly manages all component creations as well as connections between sub-components via a lifetime boundary:



COMCAD Diagrams for Specifying Assemblies

Having factored out instantiation and connection (structure-management) behavior from run-time behavior, we can now consider a generic data driven component that can instantiate and manage assemblies. We call this component *the assembly-wiring engine*. We liken assembly diagrams to hardware wiring diagrams, where the wires represent connections between components.

The wiring engine knows how to interpret data that describes an assembly to actually perform the necessary instantiation and connection of sub-components. We call the data a *wiring template*.

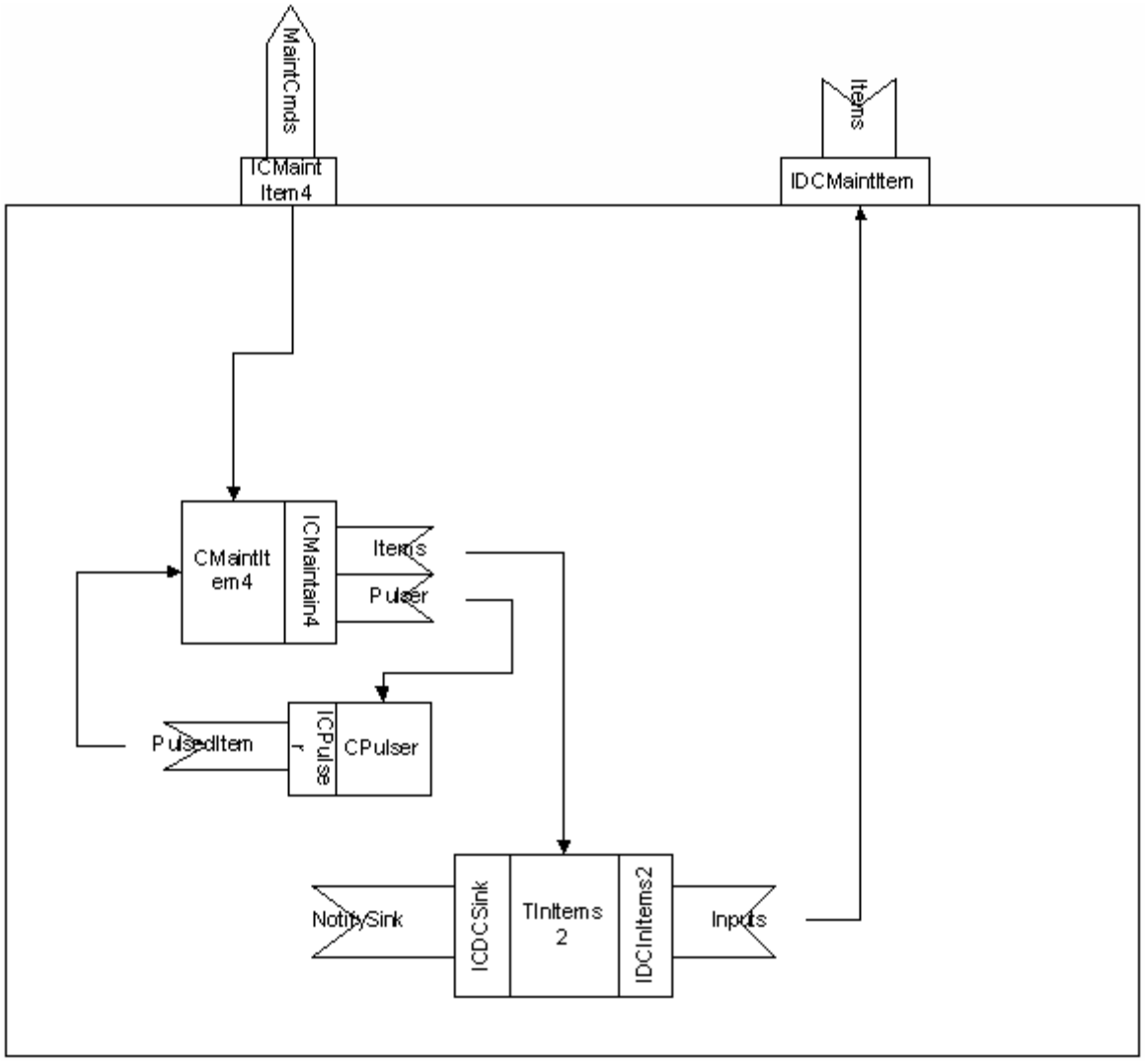
Our team has built a prototype graphical tool for interactively drawing the assemblies. This tool, called COMCAD, then generates a wiring template from an assembly diagram. The wiring template is then executed by the assembly-wiring engine to perform the actual instantiation and connection of the COM components.

Essentially this has become a form of graphical programming or executable diagrams. We believe that that many interesting composites can be constructed out of relatively few parameterized components and sub-assemblies. Wiring templates are an interesting representation for use across networks because they are small, cheap to download, processor neutral, and can be determined to be safe (being data rather than code).

Note that data driven wiring is not an exclusive requirement for implementing assemblies. The architecture defines how any other software can do wiring, and participate in the system in exactly the same way.

Visual Elements and the COMCAD Diagram Tool

The following diagram is an image of an actual diagram created with our prototype Assembly drawing tool:



This is an image of a single assembly. It contains three sub-components, in this case all of the sub-components are elements that support connectors (although these could have been sub-assemblies as well). There are two export connectors each with a single pin. The arrows (a.k.a., wires) indicate the connections that will be made during instantiation of the assembly.

When the wires point to a box rather than a connector, it means that a IUnknown::QueryInterface is being used rather than IConnector::GetElement to retrieve the out pin side of the wire. The notation is simple and straightforward.

Once an assembly has been saved and built, it can then be used in another assembly diagram as a sub-assembly. In this way, a hierarchy can be built up with progressively more functionality at each level of the hierarchy.

Back to top

Automating Interoperabilty For Heterogeneous Software Components

Alan Kaplan
Clemson University, Department of Computer Science
Box 341906, Clemson, SC 29634-1906 USA
kaplan@cs.clemson.edu,

Bradley Schmerl
Clemson University, Department of Computer Science
Box 341906, Clemson, SC 29634-1906 USA
schmerl@cs.clemson.edu,

Jack C. Wileden
University of Massachusetts, Department of Computer Science
Box 34610, Amherst, MA 01003-4610 USA
wileden@cs.umass.edu,

This position paper addresses Section 4, Research Issues and Directions, in the *Proposed Outline for Handbook of CBSE*. Specifically, this paper discusses research issues and directions related to the problem of assembling and integrating software components that have been constructed in different programming languages. This problem is frequently termed the *interoperability problem*. We claim that the interoperability problem is a fundamental concern in the area of component-based software engineering.

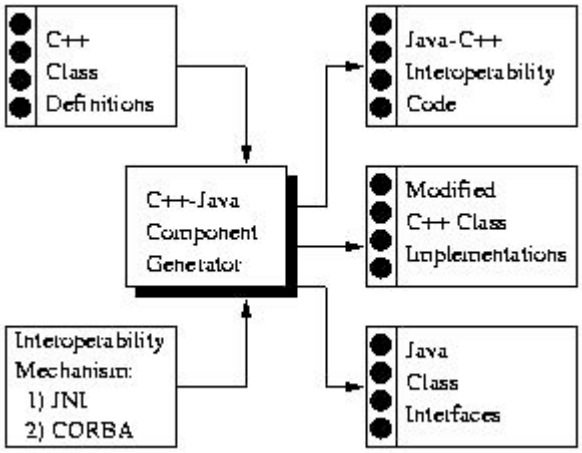
Interest in interoperation among software components developed using multiple programming languages is growing rapidly in the software engineering and programming languages community. Reusable software libraries and integration with legacy systems are two common interoperability problems faced by software developers. The expected growth and acceptance of the Internet along with the advent of new programming languages strongly suggest that interoperability will become an even greater issue in the coming years. Although various software engineering tools and programming language constructs supporting interoperation have been proposed and used in the past, these approaches do not meet the demands imposed by today's rapidly evolving heterogeneous computing environment. They are generally difficult to use and prone to error, often forcing developers to waste valuable time dealing with the complexities of a particular interoperability mechanism.

Various software engineering tools and programming language constructs supporting interoperation have been proposed and used in the past. Examples of language-based approaches include the C++ *extern* construct [1] and the Java Native Interface (JNI) [2], each of which support interoperation with the C programming language. In recent years, a myriad of alternative, sometimes competing, interoperability mechanisms (frequently referred to as *middleware* or *componentware*), has been developed. Instead of using specific language constructs, these mechanisms generally rely on a combination of specification languages or intermediaries, code generators and/or highly specialized design styles to achieve interoperability. Examples here

include OMG's CORBA [3], Xerox Parc's ILU [4], Microsoft's OLE/DCOM [5], ODMG's ODL [6] and Sun's RMI[7].

Despite, or perhaps due to, their number and variety, contemporary interoperability mechanisms are difficult to use and applications that require interoperability mechanisms are difficult to engineer and maintain. First, the choice of a specific interoperability mechanism often becomes inextricably intertwined with the application. This not only hinders development, but makes it extremely difficult, if not infeasible, to change the interoperability mechanism at a later point during the application's lifetime. Second, applications that require access to pre-existing components often force programmers to re-engineer these components and/or the application itself, thus further reducing the flexibility and robustness of an application. Third, contemporary approaches to interoperability demand far too much programmer involvement in low level details to be appealing to most software developers. Although some of these approaches provide a modicum of automated support, in general they are not well-integrated and require manual intervention, thus making them tedious to use and prone to error. As a result, they force software developers to waste valuable time dealing with the complexities of a particular interoperability mechanism, instead of focusing on the problem domain.

We are currently working on several projects that attempt to address these issues [8,9,10]. The details of these projects are beyond the scope of this position paper. However, they each share the overall goal of trying to hide the underlying interoperability mechanism for heterogeneous software components. For example, we are currently developing and experimenting with a tool that automates interoperability between Java and C++ components. Specifically, the tool allows engineers to create Java class interfaces to existing C++ class libraries using either the Java Native Interface or CORBA. The figure below provides a conceptual architecture of this tool:



The purpose of this tool is to provide Java interfaces to existing C++ classes. The tool takes as input the C++ class interface and implementation definitions. The user of the tool also indicates whether JNI or CORBA should be used as the underlying interoperability mechanism. The tool analyzes the C++ class interface and produces a corresponding Java class. Specifically, all public C++ methods are provided in its Java counterpart. Note that the tool does not simply translate from C++ to Java. For example, instance variables defined in the C++ class are not created in the

generated Java class. Similarly, private method members are not created in the generated Java class. The tool essentially produces a Java *proxy* for the C++ class. The tool also automatically generates the required Java-C++ interoperability code (as dictated by JNI or CORBA). Finally, the tool may modify the implementation of a C++ class; however, the C++ class interface remains unchanged. Currently, our tool is only in a prototype stage. We are in the process of refining its capabilities and experimenting with techniques to make it more robust. The tool presently only allows for invoking C++ from Java. We intend to support the invoking Java from C++ in the near future. We are also discovering interesting phenomena that arise when trying to provide transparent interoperation between two seemingly similar object-oriented languages. For example, C++ allows:

- pointers to pointers to pointers and so on
- multiple inheritance
- virtual and non-virtual methods

At this stage, it is not clear how to provide corresponding constructs in Java. Our approach is unique compared to existing approaches in that:

- It does not require the use of separate languages and/or type systems (such as CORBA's IDL) to achieve component interoperability. Our tool, working much like a compiler or translator, analyzes the component interfaces and generates the necessary code allowing heterogeneous components to interoperate.
- It allows the possibility of employing different interoperability mechanisms.This allows programmers to develop components without having to commit to a particular interoperability mechanism. For example, an application may choose to integrate components into a single address space (e.g., using JNI) or combine components in a distributed environment (e.g., using CORBA).
- Components provide the same interface independent of an underlying interoperability mechanism. In other words, the decision to interoperate has minimal impact on the component's interface. While contemporary approaches pollute the component code space with interoperability-specific code, our tools ensures that a component's interface remains interoperability code-free.

In summary, it is our position that interoperability is a fundamental concern in component-based software engineering. Although advances in programming language technology have yielded numerous improvements in the construction of components, it is clear that no single programming language will ever dominate software engineering practices. Therefore, new tools and techniques are needed to help software engineers interoperate among heterogeneous components. In this position paper, we have briefly outlined several related research issues and directions. We have also outlined some of our own work that addresses this area.

REFERENCES

1. Stanley B. Lippman. *C++ Primer*, chapter 4, pages 213-214. Addison-Wesley, second edition, 1993.

2. Sun Microsystems Inc. Java native interface, May 1997.

http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

3. Object Management Group. *The Common Object Request Broker: Architecture and Specification,* August 1997. Revision 2.1.

4. Bill Janssen and Mike Spreitzer. ILU: Inter-language unification via object modules. In *Workshop on Multi-Language Object Models*, Portland, OR, August 1994 (in conjunction with OOPSLA'94).

5. Kraig Brockschmidt. *Inside OLE, 2nd Edition*. Microsoft Press, 1995.

6. R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade, editors. *The Object Database Standard: ODMG 2.0.* Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1997.

7. Sun Microsystems Inc. Java remote method invocation, May 1997. http://java.sun.com:80/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

8. A. Kaplan, J. V.E. Ridgway, and J.C. Wileden. Why IDLs are not ideal. In *Proceedings of the Ninth IEEE International Workshop on Software Specification and Design,* Ise-Shima, Japan, April 1998.

9. Alan Kaplan and Jack C. Wileden. Toward painless polylingual persistence. In *Seventh International Workshop on Persistence Object Systems,* Cape May, NJ, May 1996.

10. Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *The Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.

Clemson University, *Department of Computer Science*
Box 341906, Clemson, SC 29634-1906 USA
kaplan@cs.clemson.edu, *http://www.cs.clemson.edu/~kaplan*

Alan Kaplan
Last modified: Thu Mar 18 11:29:34 EST 1999

Second International Workshop on Component-Based Software Engineering

Position Paper: David Budgen

Department of Computer Science
Keele University
Staffordshire
ST5 5BG
U.K.

1. Comments on the CBSE Strawman Document

My (brief) comments on this are as follows:

- The idea of exploring the issues via a Strawman document such as this seems a very good one, and I think that producing such a document forms an excellent step in the development of the area.
- I would argue that the document cannot afford to ignore the problems of *creating* such systems and that there is a need for a section that addresses this, even if at the present, we have only limited experiences to offer. I have suggested the outline for a new section below (probably it should become a new section 3, with the present 3 becoming 4). As a part-justification for this, I would observe that this is identified as a significant theme within the PITAC Interim Report (http://www.ccic.gov/ac/interim/), which comments that "Special emphasis should be placed on component based software design and production techniques" in identifying research priorities.
- The other area that has emerged as needing more attention from some work of our own, that has involved making a small investigation into *how* people design solutions from components, is that of component *documentation.* This is an area where there appears to be a potentially key need for standard forms of description to be developed.

2. Suggested structure for a new section

System development for CBSE-based projects
- Life-cycle issues
  - Integration with 'traditional' waterfall activities (req. elicitation; design; testing)
  - Role in Rapid Application Development
- Design Practices
  - Reuse strategies (horizontal/vertical)
  - Changes to practices used in system design
  - Underpinning technology needs /* reason to precede existing section 3 */
  - Component development practices
- Testing and Evaluation
  - Recommended practices
  - Documentation

3. Discussion of design practices for CBSE

At present no widely recognised software design practices incorporate the concept of reusing pre-existing components. The 'historical' approach to software development has (on paper at least) encouraged the adoption of design methods such as SSA/SD, JSD, OOSD etc., which assume:

- A 'clean slate' solution, in which each design solution is custom-built to fit the needs of the particular problem.
- The reuse of design 'process' experience is mainly conveyed through the practices of the design method.

Indeed, some of the current interest in the concept of 'design patterns' [Gamma et al., 1995] may well reflect the difficulty that these existing practices have in incorporating guidance on reuse.

Support for the idea of software component reuse in design needs to be related to the concept of software architecture. In the past, each system has been custom built, and its architectural form has been determined on the basis of such considerations as reuse and locally preferred practices, or as a separate and independent choice. This is also an area where the experience of hardware development has been very positive, in that the adoption of standard architectural forms has formed an important underpinning to component reuse. For software the concept is still in relatively formative stages [Shaw and Garlan, 1996; Monroe et al., 1997], and the ease with which software can be modified also makes it harder to enforce any standards for software architecture.

Historically the main corpus of software design literature has been focused upon the task of designing 'bespoke' solutions, although it can be argued that use of a design method produces an 'architectural style' that increases the potential for reusing elements of designs produced using the same practices. However, th
is has traditionally received little attention and we are not aware of the existence of any widely used systematic practices that are based upon reuse of part-solutions.

Observations of software designers and their practices in a 'bespoke development' context have indicated that such reuse of part-solutions does occur [Adelson and Soloway, 1985; Guindon, 1990], although during the design process this may be more a case of reusing paradigms or idioms from previous experience (termed 'labels for plans'), rather than involving actual system elements.

These notes review the question of how a transition to a component-based paradigm is likely to alter the designer's goals and the overall system expectations. We then examine some of the empirical work on software design activities as a preliminary to considering how design practices will need to change to meet these goals. Finally, we make some brief comments based upon some recent work of our
own.

Changing the goal posts

The distinction between our expectations when producing bespoke software solutions and reusing components should perhaps be similar to our expectations of ‘made to measure’ and mass-produced clothing. It we have a suit ‘made to measure’, then we expect it to be not only a good fit (and many people are physically asymmetrical as regards such issues as shoulder height and arm length), but also to be able to provide for any personal preferences such as secure inner pockets. However, if purchasing ‘off the peg’ clothing then we are prepared to accept some degree of compromise providing that the overall fit and serviceability is acceptable (and of course, that the cost is also significantly less than that of the bespoke product). Should it become necessary to alter the off the peg product to achieve an acceptable fit, the question of cost becomes critical, since this may approach or even exceed the cost of purchasing a bespoke garment.

Within the software development process there are clear analogies to this, and in order to achieve an equivalent degree of compromise when proposing to develop a solution using components, we need to be able to determine both:

- what degree of compromise we can accept
- how we are to measure this

and to agree these ’up front’ before beginning the processes of design and development. Indeed, it may even be that a thorough analysis will reveal that a component-based solution is not necessarily more cost-effective than a bespoke development. (There is an interesting caution to this effect in the conclusions of [Lewis et al., 1992], where they observe that what they term ‘strong reuse encouragement’ may lead to "the subject’s reuse of inappropriate components", resulting in lower productivity!)

Unfortunately, the invisible and multi-faceted nature of software makes it much harder to find criteria for compromise than is the case for clothing! Some possible parameters that a designer might need to consider (for both the overall solution and also in some cases, for the components) include:

eventual system performance
resource use (especially memory)
interface forms
functionality

(Note that these are not necessarily independent.) These factors will also have implications for the ways that components may need to be documented.

Empirical studies of designers

The corpus of material covering empirical studies of software design activities is a fairly small one, generated chiefly by workers with a background in cognitive psychology as well as software. A significant feature of this is the relatively small number of subjects used by most researchers, reflecting the difficulty of gaining access to suitable experienced software designers, and even of suitably inexperienced student subjects, since even inexperienced subjects need knowledge about design.

We review this material here mainly in terms of the extent to which any component-related concepts can be identified. (Since none of it was originally concerned with such systems, we should not regard this as necessarily providing an exhaustive analysis!)

One of the earliest published studies on observations of larger-scale software design activities is that of [Adelson and Soloway, 1985], which examined the strategies used by a small number of experienced designers (three) plus two novice designers, when faced with problems that could be categorised as follows.

- familiar problems in a familiar domain
- unfamiliar problems in a familiar domain
- unfamiliar problems in an unfamiliar domain

Two of the important ideas that they identified in terms of our own focus of interest was that designers frequently adopted an opportunistic strategy (which we can characterise as being problem-driven rather than method-driven) in solving their problems, and the concept of labels for plans. The latter can be considered as a means of identifying where previous experience can be reused, by noting the existence of a previously-used plan for a part-solution, which can be retrieved from memory without the need to work this through in detail while solving the current problem.

Later work by Guindon was based upon a study of three experienced designers, and the work reported in [Guindon, 1990] focused on investigating the means by which the subjects exploited their prior knowledge and experience in solving a problem. In particular, this study identified the use of schemas by designers to aid reuse of experience, where "a schema is a complex rule composed of a pattern which specifies the similarities in requirements between different instances of a class of systems". Such a description of a schema can probably be loosely equated to the concept of 'labels for plans' described by Adelson and Soloway. It was also noted that "designers tend to reuse the same successful solutions over and over in their career".

In terms of a component-based approach, both of these studies do indicate that experience is reused at an abstract level, although not necessarily mapped on to concrete components. However, since components were not included in the studies, this is perhaps not surprising.

A very thorough analysis of both the nature of the design process as applied to software, and also of the methodological issues that arise when conducting any study of how software is designed is provided in [Davies and Castell, 1992] . A similar analysis in [Visser and Hoc, 1990] contains one of the few references to problem of reuse that we could identify, and concludes that:

> "*one may think that most psychological studies paying so little attention to this reuse - contrary to software engineering, which considers it a major problem to be solved - is due to their limited context making reuse difficult to implement*"

Modifying the design process

Having argued that the 'traditional' approach to software development based upon design methods that produced bespoke solutions does not provide adequate support for a component-based development strategy, and identified that the bulk of empirical studies do not address the component concept, one question that arises is that of how such systems might be most effectively developed.

Two identifiable strategies, based on different criteria, are described below. We should note that these probably describe fairly extreme forms, and that any practical development is likely to involve some degree of compromise between the two.

1. Assuming that an architectural style is already established, for whatever reasons, then the components need to be identified on the basis of their ability to conform to the needs of the architecture as well as on their functionality. We can regard this strategy as one of **Framework First**, using the pre-determined framework to help narrow down the search space for components. Certainly such an architecture will itself have some specific characteristics, and is likely to imply some form of open, loosely coupled interaction, including the possibility of 'plug and play' capability to support evolution.

2. A quite different approach is one that begins by identifying a set of suitable components and then to decide upon the architectural form for the solution on the basis of how to get the best from the selected components. Again, such an **Element First** approach is more likely to imply a more closely coupled strategy, and may provide less scope for long-term evolution.

Framework First The adoption of such a strategy is consistent with the needs of larger organisations, concerned with reuse of internally generated components as well as those obtained externally, and with longer-term maintenance needs also in view. As with adoption of design methods, such a strategy does require careful consideration before choosing the framework, since this decision occurs both early in development (or in transition to a component-based philosophy) and may also have significant longer-term effects. (While the Object-Oriented paradigm is probably the predominant one currently in use for new developments, it may not always be the most appropriate framework, especially where real-time needs predominate.)

Implicit in this choice of strategy must be an acceptance that it may be necessary to develop either new components, or modifications to existing ones, in order to achieve a solution for a given problem. This represents a major risk factor, and is also strongly linked to the choice of framework.

Element First For this strategy to work, the designer may need to be able to search a very large solution space in order to find appropriate components [Frakes and Pole, 1994]. Its adoption is therefore much less one of an organisational one, and as we identified above, it is probably a strategy that is only really suited to solving 'one-off' problems, with relatively little need to consider longer-term maintenance needs.

The main problem (or risk factor) is that of making an adequate search for components at the

outset. This requires both a wide search space and also efficient methods of searching this. Given that component classification is likely to be uneven, ensuring that any search is made effectively is likely to present the major risk, since an eventual set of components that have mis-matched interfaces are unlikely to provide an overall solution that is satisfactory in terms of the parameters identified above.

A second issue that arises is the degree of modification that is acceptable [Basili et al., 1994]. Since modification is likely to be expensive; require expert skills; introduce the need for additional version management; and increase testing needs, it is clearly an option that requires care. The idea of the wrapper [Brown and Short, 1997] is probably a more attractive approach.

Observations from some simple empirical studies

We have undertaken some simple studies of design activity, based upon the use of Unix processes as reusable components, with some intermediate findings being described in [Budgen and Pohthong, 1999]. Since the choice of shell programs effectively constitutes an element of 'Framework First', we have been mainly concerned to investigate the search and selection strategies used by our subjects in a series of studies. (We might also observe that the whole philosophy of Unix also means that Unix utilities are effectively designed for the purpose of reuse.)

One significant issue that has emerged is the designer's need for*information* about a candidate component, regardless of the strategy used to find it. On some occasions, this has also included executing the component separately, in order to investigate its behaviour when used in a particular mode. While these observations do need to be further investigated in a wider context, they have significant implications for both the design of tools intended to support component-based development and also for any documentation standards that might emerge.

References

[Adelson and Soloway, 1985] Beth Adelson and Elliott Soloway, The role of domain experience in software design, IEEE Transactions on Software Engineering, 11(11), November 1985, 1351-1360.

[Basili et al., 1994] V R Basili, L C Briand and W I Melo, How Reuse influences productivity in Object-Oriented Systems, Communications of the ACM, 37(5), 1994, 104-115.

[Brown and Short, 1997] Alan W Brown and Keith Short, On Components and Objects: The Foundations of Component-Based Development, Proceedings of the 5th Int. Symposium on Assessment of Software Tools and Technologies, 1997, 112-121.

[Budgen and Pohthong, 1999] David Budgen and Amnart Pohthong, Component Reuse in Software Design: An Observational Study, submitted for publication.

[[Davies and Castell, 1992] S P Davies and A M Castell, Contextualizing design: narratives and

rationalization in empirical studies of software design, Design Studies, 13(4), 1992, 379-392.

[Frakes and Pole, 1994] W B Frakes and T P Pole, An Empirical Study of Representation Methods for Reusable Software Components, IEEE Transactions on Software Engineering, 20(8), 1994, 617-630.

[[Gamma et al., 1995] E Gamma, R Helm, R Johnson and J Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995.

[Guindon, 1990] Raymonde Guindon, Knowledge exploited by experts during software system design, Int. Journal of Man-Machine Studies, 33, 1990, 279-304.

[Lewis et al., 1992] J A Lewis, S M Henry, D G Kafura and R S Schulman, On the relationship between the object-oriented paradigm and software reuse: an empirical investigation, Journal of Object-Oriented Programming, 5(4), 1992, 35-41.

[Monroe et al., 1997] R T Monroe, A Kompanek, R Melton and D Garlan, Architectural Styles, Design Patterns, and Objects, IEEE Software, January 1997, 43-52.

[Shaw and Garlan, 1996], Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice/Hall, 1996.

[Visser, 1987] Willemien Visser, Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer, in Empirical Studies of Programmers: Second Workshop, Editors G Olson, S Sheppard & E Soloway, Ablex Publishing, 1987, 217-230.

[Visser and Hoc, 1990] W Visser and H-M Hoc, Expert Software Design Strategies, in Psychology of Programming, Editors J-M Hoc, T Green, R Samurçay and D Gilmore, Academic Press, 1990, 235-249.

Evolution of Component Based Systems

Pearl Brereton
Department of Computer Science
Keele University
Keele
Staffordshire
ST5 5BG
UK

Email o.p.brereton@cs.keele.ac.uk

March 1999

1. Introduction

Many organisations are moving towards a component based approach to software development. However, there is a significant risk that component based systems will become the legacy software of the future. The difficulties of maintaining systems for which responsibility is distributed across many authors, owners and organisations is aptly illustrated by the increasing World Wide Web (WWW) maintenance mountain [1].

The strengths and opportunities associated with component based development stem from the potential for reduced costs and increased functionality and quality that multiple suppliers (of components) can bring. Potential benefits also accrue from reuse. These are well documented although considered by many to be slow to materialise. In addition, diversity in the solution space should improve component integrators' options to trade requirements against cost, delivery time and/or other factors such as component quality and supplier reputation.

The commercialisation of software components can be expected to widen the range of drivers for software change. Traditionally, software change has been driven externally by customer and market requirements and internally, by the need for corrective and adaptive development and maintenance. For component based systems (CBS), we might add to these, the push by vendors to stimulate change by offering new improved (or cheaper) components or by withdrawing support for components already in use. Similarly, integrators may choose to move to preferred suppliers or away from risky or blacklisted suppliers. In this way, the supply chain becomes an important dependency to be accommodated by change management systems.

The evolution of CBS, like that of WWW documents, needs to be strictly controlled if such systems are to maintain their initial levels of quality throughout their operational life.

This position paper aims to identify the major CBS maintenance issues and to suggest areas of research needed in order to address these issues. It aims to contribute to sections two and four of the Strawman outline.

2. Maintenance Issues

CBS maintenance issues are listed under the headings: *business*, *management* and *technical*.

Business issues

Responsibility for change - The nature of software makes it notoriously difficult to separate out the source of a particular fault even when an in-house team produces the elements. For CBS it will be important to establish sound methods of assigning and enforcing responsibility for parts and for the 'whole' system. Integrators (and customers) of CBS may chose to work with a limited set of 'preferred suppliers' rather than acquiring components on the open market.

Risks of change - The many risks traditionally associated with change are likely to remain, or even to increase, for CBS. For example, analysing the impact of replacing components by others from different suppliers will be a more complex task than undertaking impact analysis 'in house'. Risk analysis is not a widespread software engineering skill.

Payment for change - A number of issues relating to payment and charging are likely to arise and it is possible that billing will become a major overhead for the component based software industry. If, for example, customers pay for components on a per use basis (where such use may involve remote execution), payment models may have to incorporate payment to component providers as well as to systems integrators. A much broader range of payment models than are used at present may be needed in the future to accommodate both the complex webs of owners and agents as well as different purchasing and licensing models.

Future proofing - The potential for providing long-term support is likely to be a major factor to be considered when purchasing components (since history suggests that even 'throw-away' systems can remain in use for quite long spans of time!). Employing a mechanism such as escrow (keeping source code with an independent, trusted and secure repository) may help with customer reassurance. The use of such practices as employing multiple sources (as with hardware systems) and using preferred suppliers may provide further reassurance.

Management issues

Drivers for change - CBS maintainers are likely to be subject to disparate and potentially overwhelming demands for change. On the one hand, vendors of components will continue to produce and market new improved components and may also either withdraw support for components in use or adjust charging such that change is unavoidable. On the other hand, customers, as today, will always require new features or facilities. In addition, vendors of integrated systems are likely to strive to identify new markets in order to extend their portfolio of products and services.

All three of these 'drivers for change' imply the need for quite different practices to those employed to produce 'bespoke' systems (tailor-made for the individual customer) or packaged systems such as word processors and spreadsheets.

Change policies for distributed systems - The assumption in this paper so far has been that components of a CBS are physically integrated to provide an executable system to run on a customer?s computer. However, current technology supports the development of virtual (integrated) systems where components remain at the provider sites (or at some other remote site) and are accessed as required. Such systems could reduce some of the problems associated with system upgrades and version management. Upgrading a component of such a system could be carried out by 'simply' replacing it by another. Possible upgrade policies (or strategies) might include:

- Replace component with a new version (no notification to users)
- Replace component with a new version and notify users
- Give users a choice to continue using the 'old' component or move to the new one
- Require new users to use only the latest version of a component

Component documentation and description - information about components will need to be accumulated and combined. Such information will come from a number of sources and will have a range of forms (e.g. factual, opinion, statistical). Sources of information might include:

- component suppliers - e.g. advertising literature, component introspection
- standards bodies and certification organisations - e.g. certification of compliance to interface standards
- special interest groups - e.g. component reliability measures from other users
- integrator organisations - e.g. assessments through benchmarking, case studies, experiments

Technical issues

For CBS, the fundamental unit of change is at the component level. Change includes:

- upgrading a component to a new version, provided by the same supplier
- replacing a component with one from a different supplier
- adding a new component
- removing a component

Maintenance activities will include:

- *locating*, *understanding* and *evaluating* potential replacement or additional components
- *determining the impact* (on the overall system) of potential change
- *estimating the cost of re-testing*
- *evaluating risks* associated with using new suppliers

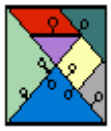
3. Key research areas

>From the issues discussed ,the following CBS maintenance research topics emerge:

- **Evaluation** - including evaluation of components, products suppliers, development and maintenance strategies and alternatives, architectures, risk, productivity, skills.
- **Interaction and integration of business and technical factors** - relating to, for example, selection of components and suppliers, cost/functionality/quality/availability/confidence trade-offs, future proofing, managing change, payment models, integration of business and technical processes.
- **Component descriptions and documentation -** including the range of description forms, sources of descriptive information, maintenance of descriptions, users and usage of descriptions and description quality.

References

1. OP Brereton, D Budgen and G Hamilton, *Hypertext: the next maintenance mountain*, IEEE Computer Vol. 31, No. 12, December 1998, pp 49-55



A Reusable Syntax Directed Processing System

Author: Chuck Strempler

Microsoft Corporation

Microsoft Research – ComApps

chuckstr@microsoft.com

Last Updated: March 18, 1999

Abstract: This paper discusses a system under development that takes a representation of any LL(k) language [1] and some source text written in that language and outputs a component hierarchy representing that source. This hierarchy may an be intermediate representation of the language or a stand-alone component composite.

To understand this paper it will be useful to have knowledge of Microsoft COM technology [2]. It will also be useful to have knowledge of our hierarchical component composition technology called assemblies [3].

1. Syntax Directed Processing and Component Hierarchies

In our research, we've come across many instances requiring interpreting a language or translating a language into another language. The following is a list of some of the problems we've encountered that may be addressed using syntax directed processing:

- C+Com Compiler- We use this tool to automate component generation and production of our component type libraries for use in our assembly tools. It translates annotated C files into C code and component type source files.
- C+Com source wizard - This tool allows declarative specification of component attributes and generates C+Com source files. It builds up an intermediate representation of a C+Com source file and dynamically alters that representation as component attributes are added or changed.
- IDL+Com Compiler- This tool allows type repositories to be built containing extended information unavailable through normal interface definition processing. It translates a type specification language into an IDL file and component type source files.
- A component wiring specification language - This could be an interpreted language that dynamically wires components together.

- Precise interface specification - This could be used to generate interface test and precondition checking code from a rigorous specification.
- Form layout specification - This could be used to generate user interface assemblies to implement complex interfaces.

Analysis of these problems yielded the following list of requirements for a system that facilitates syntax directed processing:

- It must be reusable. That is, is must be usable for language-processing across varied domains. This includes flexible processing of ambiguous language constructs.
- It must be extensible. That is, allow for easy algorithm substitution in and augmentation of the various phases of processing.
- It must be able to generate an intermediate representation for compiling and interpreting and a stand-alone composition for assembly creation.
- It must support nested invocation for mixed languages.
- It must support seamless error reporting.

Although existing syntax directed systems such as lex and yacc addressed some of these requirements, we found the most flexible, complete solution was to develop a set of components each addressing a particular phase of syntax directed processing, and use our assembly technology to combine them into an extensible, reusable composite currently known as SDPAssy. SDPAssy generates either a component hierarchy as the intermediate language representation or the language directed component hierarchy assembly.

2. Using SDPAssy For Syntax Directed Processing

The following sections describe how the SDPAssy assembly addresses the above requirements and how to use it to build assemblies, compilers and interpreters.

2.1 The SDPAssy Assembly

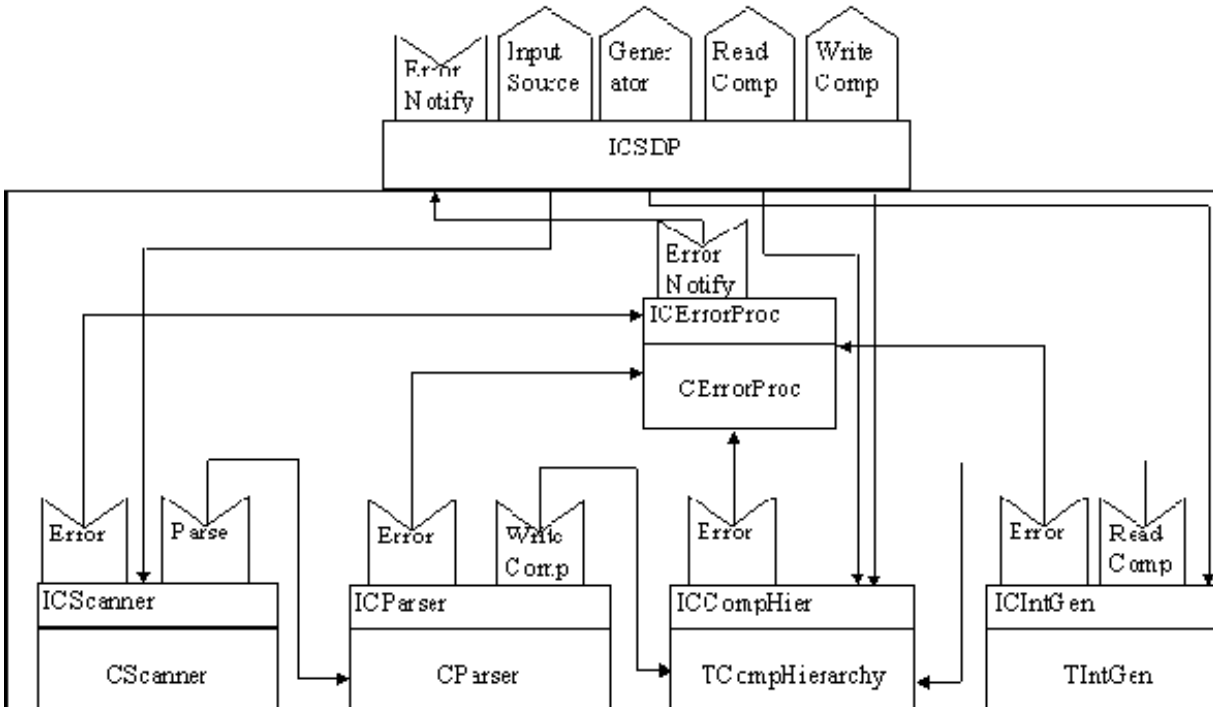


Figure 1 , The SDPAssy assembly. Please see [3] for a description of assemblies.

SDPAssy itself imports an error notification callback interface and exports an interface on the scanner to specify the input source text, an interface on the generator or interpreter and interfaces on the component hierarchy element to navigate and alter the hierarchy. Altering the hierarchy may be required for applications such as the C+Com source generator described in Section 2.1. During language processing, another instance of SDPAssy can be invoked to parse a contained language. This arises, for example, in the C+Com compiler described in Section 2.1. C+Com files can import IDL+Com files to extract type information from them.

The error handling element, CErrorProc, receives a pointer to an interface outside the assembly through the ErrorNotify role, that is called back with a formatted error message and an error code when an error occurs. Other top level elements (those listed in the diagram above) receive a pointer to CErrorProc so that they can report errors.

The lexical analysis component, CScanner, is called from outside the assembly via the Input Source pin, with the language source text to be tokenized. It calls CParser to process each token. CScanner is specialized with a binary representation of the regular-expression grammar describing the tokens in the source language and a keyword table. This binary representation is created once per token set outside SDPAssy. This allows CScanner to reuse this binary representation to efficiently generate tokens for any instances of source text with the given token set.

CParser generates a component hierarchy from the token stream. It is a predictive parser that will recognize any LL(k) [1] language annotated with commands that tell the parser to have the component hierarchy element add a new component to the hierarchy and/or call a method on an existing component. CParser is specialized with a binary representation of the predictive parse table for the source language. This binary representation is created once per language outside the SDPAssy assembly. This allows parsing arbitrary source text in the source language without reanalyzing the language each time. The language can also be annotated with commands that allow the parser to call some custom code to handle ambiguous grammar conditions. One place where this is needed is in the C+Com compiler described in Section 2.1. The C+Com grammar makes use of nested braces and matching a given '}' with the correct '{' is required. Since the grammar representing this match is ambiguous, it cannot be handled by the parser automatically. The C+Com language annotates the grammar with a reference to code that tracks the correct brace to match and can resolve the ambiguity.

The TCompHierarchy element is a placeholder instantiated at assembly initialization time with an actual component hierarchy implementation. This allows applications with differing intermediate representation or assembly generation requirements to reuse the parser and scanner implementations while providing a custom or extended component hierarchy element. The root component in the hierarchy, must implement interfaces to traverse and alter the hierarchy as well as an interface (IParseHelper) that allows communication between the hierarchy components and the parser. Other components in the hierarchy (i.e. those that are not used to specialize the TCompHierarchy place holder), may only implement IParseHelper. The example presented in Section 2.2 shows how the parser calls IParseHelper methods to direct the creation of the component hierarchy. It is worth noting that in assembly generation, the component hierarchy is the final product and the generator/interpreter is not needed. Currently, we only use domain specific hierarchy elements. Because the hierarchy models language entities, new components must be written for every language in which these entities are different. It seems plausible to have a reusable hierarchy component that keeps track of children by type and properties by name. This would allow a user of SDPAssy to write only a custom generator or interpreter and not worry about the intermediate representation at all.

The TIntGen element is also a placeholder that is instantiated with the actual interpreter or generator implementation at assembly initialization time. This component receives a pointer to the root of the component hierarchy, and can then navigate the hierarchy through private interfaces.

2.2 A Simple Example Using SDPAssy

This example illustrates the use of SDPAssy to perform syntax directed processing with a simple

language. The language is a simple form description language and is presented in annotated BNF form with terminals in caps, non terminals lower case and annotations in bold. The annotations are not part of the grammar itself but meta language tags that direct the parser to perform some action. They are attached to their following terminals or non terminals in the parse table so that the parser operates only on grammar symbols. # indicates the beginning of a compound annotation, **$** indicates to the parser that the most recently created hierarchy component should be made current and may annotate non terminals, **CREATE <*clsid*>** causes a new hierarchy component of type <clsid> to be created and passes this as an argument to the current hierarchy component's IParseHelper::Create call. All other annotations are tag names which are passed as arguments to the current hierarchy component's IParseHelper::Call along with the token being processed.

(1)forml -> **#CREATE <*formclsid*>** FORM **#$NAME** STRING **#$X** INT **#$Y** INT BEGIN **$**fieldl END forml

(2)forml -> null

(3)fieldl -> **#CREATE <*buttonclsid*>** BUTTON **#$NAME** STRING **#$X** INT **#$Y** INT **#$FORM** STRING fieldl

(4)fieldl -> **#CREATE <*labelclsid*>** LABEL **#$NAME** STRING **#$X** INT **#$Y** INT fieldl

(5)fieldl -> **#CREATE <*editclsid*>** EDIT **#$NAME** STRING **#$X** INT **#$Y** INT fieldl

(6)fieldl -> null

Here is a very small example of the language:

```
form Form1 10 10
begin
button Button1 0 0 Form2
end
Form Form2 10 10
begin
label "Look at this:" 0 0
edit Edit1 100 0
end
```

In this example, the TCompHierarchy is initialized with CFormRoot. The result of processing the sample form description language is the assembly in Figure 2a.

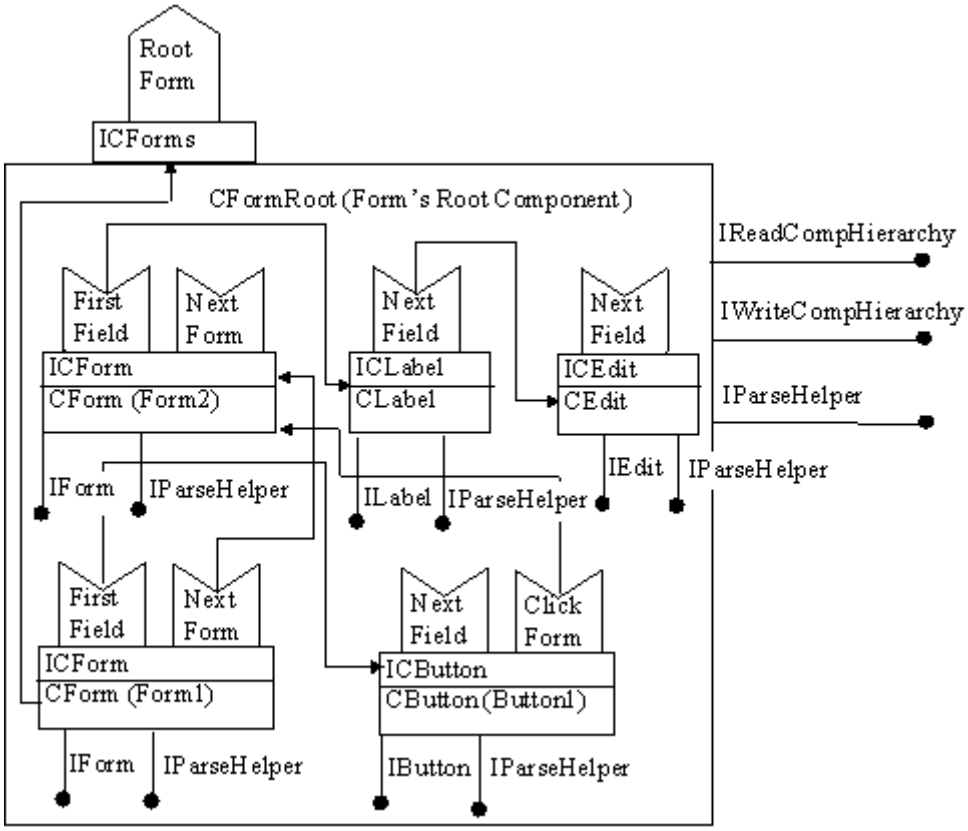


Figure 2, The CFormRoot assembly generated by SDPAssy.

Here is a derivation sequence and the corresponding semantic actions taken by the component hierarchy. The number to the left of the hyphen in each paragraph below corresponds to the rule the parser applies at that point in the parsing. The number is the rule number from the form description grammar listed above:

1 – The FORM token is matched by the parser. The annotation #CREATE <formclsid> resulted in a parse table entry for that token match notifying the parser to create a new component of type <formclsid> and pass it to the current (root) hierarchy component's IParseHelper::Create method. The parser continues and the STRING token is matched by "Form1" in the source. The **#$NAME** annotation resulted in a parse table entry for this token match notifying the parser to set the current component to the newly created form (the **$** triggered this) and call the current component's IParseHelper::Call with the annotation name (**NAME** in this case) and the token. The current component is then reset to the old (root) component. Similarly, the INT, INT and BEGIN tokens are matched with the 10, 10 and begin source text respectively. In each case, the current component is set to the new form component and its IParseHelper::Call is called with the annotation name and the current token. Since the non terminal fieldl is marked with the **$** annotation, the current component becomes the Form1 component for all of the field list processing. This sequence allows the root component to wire in Form1 and export its IForm interface, and to set the form's internal name, X, and Y coordinate attributes.

3 – The button token is matched which causes a new button hierarchy component to be created and passed to the current component (Form1). Form1's IParseHelper::Create is called to wire the field into the form. The NAME, X, Y and FORM attributes are set for the new button due to the matching of the STRING, INT, INT and STRING tokens to "Button1", 0, 0 and "Form2" respectively and the resulting calls to the button's IParseHelper::Call.

6 – There are no more fields left in Form1 and no annotations here. The END token is matched from rule 1 and the current component is reset to the root.

1 – The FORM token is matched resulting in Form2 being created and passed to the root's IParseHelper::Create. The NAME, X and Y properties are set for Form2. The BEGIN token is matched. The current component is set to the newly created form when the non terminal fieldl is processed. The wiring of Form1's button to form2 could also be done in the root's IParseHelper::Create since Form2 is a known entity now. This is the form that will appear when the user clicks the button in form1.

4 – The LABEL token is matched resulting in the creation of a new label component. Form2's IParseHelper::Create is called so that the new field can be wired into the form. The NAME, X and Y attributes are set using the label's IParseHelper::Call method as a result of matching the label name, X and Y coordinate tokens.

5 – The EDIT token is matched resulting in the creation of the edit component and its wiring into the assembly via the call to form2's IParseHelper::Create. The NAME, X and Y attributes are set for the edit field as a result of matching the edit name, x and y coordinate tokens.

6 – There are no more fields left for form2. The END token for form2 is matched in rule1.

2 – The ENDOFINPUT token is matched and processing is complete.

An external component could extract the resulting assembly's root IForm interface (the one for form1), through the root form pin and call IForm::Display to display form1. Clicking on the button in form1 would result in the calling of form2's IForm::Display.

3. References

[1] Aho, Alfred; Sethi, Ravi; Ullman Jeffrey; Section 4.4 Compilers Principals, Techniques and Tools, Addison-Wesley, 1988.

[2] Microsoft, Papers on COM, http://www.microsoft.com/com/dcom.asp.

[3] Peltz, Chris, "A Hierarchical Technique For Composing COM Based Components", 1999.

Back to top

Home

A Model for Classifying Component Interfaces

Sherif Yacoub, Hany Ammar, and Ali Mili
{yacoub,hammar,amili}@csee.wvu.edu
CSEE Department
West Virginia University,
Morgantown, WV 26506

Abstract

This paper identifies some issues related to component interfaces. We present a model for component interactions and interfaces to the surrounding artifacts. We classify interfaces as *Application* and *Platform*. Classification of interfaces helps in identifying issues related to a component's interoperability (interactions with other components) and portability (interactions with the platform). The model is a preliminary step towards establishing a framework for classifying and evaluating which languages and notations are adequate to specify different types of interfaces. We propose this classification for the third section of the CBSE handbook " *Technology for Supporting CBSE: Development Support* " under the " *Models* ".

1. Component Interfaces

Component-based software development is the process of assembling software components in an application such that they interact to satisfy a predefined functionality. Each component will provide and require pre-specified services from other components, hence, the notion of component interfaces becomes an important issue of concern. "*Components are expressed in terms of externally visible interfaces and semantics, not the implementation*" [2] where interfaces are the mechanisms by which information is passed between two communicating components. The use of components exacerbates interface centered software architecture because components offer interfaces to the outside world, by which it may be composed with other components [3].

Several work in component interfaces [for example 8,9, and 7] focused mainly on issues related to interaction between individual components. Component interfaces were classified as "functional" and "extrafunctional" [7], defined for UML models [8], and for object oriented designs [9]. We further abstract component interfaces to incorporate interfaces to platforms and elaborate on the importance of such classification.

In this short paper, we present a model for a component's interactions which mainly classifies interfaces as *Application* and *Platform* interfaces. This classification is useful to:

- Understand the behavior of a component and its interaction with other components and with the system on which it executes.

- Evaluate the adequacy of languages and notations to specify component interfaces.
- Inventory the range of possible inter-component interactions and use this inventory as the basis for a semantic definition of architectural constructs.
- Give some leverage on the opposition between functionality and packaging.

2. Modeling Component Interactions

2.1 The Model

Modeling software components is important to facilitate the understandability of the components themselves and the understandability of activities related to CBSD such as adapting and assembling components. The following figure shows a model that describes the component as related to its surrounding artifacts with emphasis on types of interfaces. The model is used to categorize component interactions.

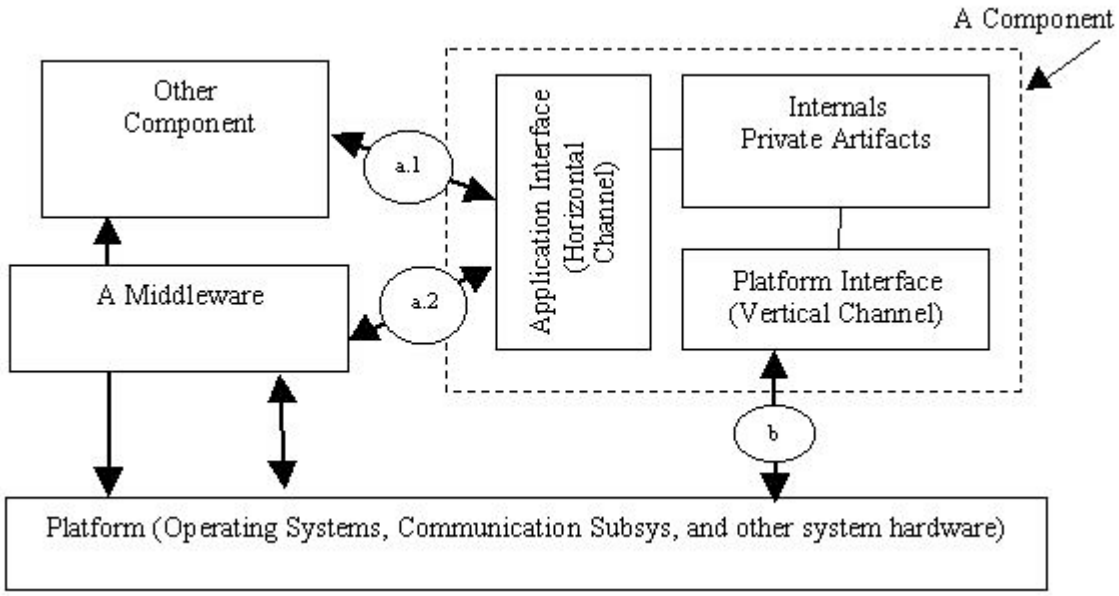


Figure 1 The Model

We distinguish the following model elements:

Internals (Private Aspects)
This section of the model represents the internal information and structure of a component. It provides the actual functionality of the component as exposed by its interface. This element is private to the component and it is not exposed to any other components or the platform on which it runs. The component internals is characterized by encapsulating the decisions and hiding them from other components.

Application Interface
Those interfaces define the interaction with other application artifacts such as other components or applications. This interface represents the import and export relationship with other components (or the middleware) with which the component interacts. A set of exported interfaces represents the functionality that this component can provide. A set of imported interfaces represent the functionalities that this module requires from other external components which might be needed in the work progress of the component functional execution. We term these interfaces as "*Horizontal Channels*" as they specify the interaction with other peer components and application entities irrespective of the platform or hardware on which they run. The horizontal channel allows us to identify:

- The structure of messages sent/received from other component.
- Timing issues as related to requests going to/from the component
- Incompatibilities in data format, types and message protocol

Platform Interfaces
Those interfaces define the component interaction with the platform on which it executes. These interfaces would include operating system calls, the underlying hardware technology, and communication subsystems. For a component to run it should be supported by specific processor, memory, communication equipment and probably other hardware as well. This type of interaction is as important as interaction with other software components. It determines the portability of the component and how it runs and executes on specific hardware. This layered approach helps the designer in specifying and designing components that are independent of programming languages and operating systems. Several implementations may have different platform interfaces and yet have the same design and specifications. This interface layer is also called "*Vertical Channel*" because it identifies interactions with lower layers of hardware not with other peer components. This type of interfaces is essential for special type of applications (embedded systems for example) in which 20-30 % of safety-related errors discovered were related to these interfaces [4, 5]. The following are examples of platform interfaces:

- Operating System
- Hardware platform
- Communication channels (and protocol stacks)
- Compilers (if required to compile the component)

The Vertical Channel allows us to identify impacts of failures and risks as related:

- Failure to detect and respond to operating system and communication event
- Produce undesirable outputs to communication channels
- Misunderstanding how the hardware operates
- Portability to other platforms, (ex. a component running on Unix operating system should be differentiated from those running on Windows based or on micro-controllers)

2.2 Component Interactions

Patterns of component interaction in component-based software engineering is another major

concern. Using the model of the component, we identify the following types of component interactions (numbered as shown in figure 1):

a) Application Interfaces (Horizontal Channels)

a.1) Direct Interaction
Direct interaction are those from one component to the other, in this case a component knows of the existence of other components and directly invokes one or more of its services. This type of interaction creates a direct coupling between components in the application.

a.2) Indirect Interaction
Components can interact with each other through a standardized middleware or kernel. A component publishes its services to the middleware. Other components can inquire about the possible supported services and require them without knowing where the other component is located. Indirect interaction is established through a standardized kernel, usually referred to as a middleware such as COM [1] or CORBA [6].

b) Platform Interfaces (Vertical Channels)
Components interact with other operating system components, communication subsystems, or other hardware components. These interaction protocols are determined by the nature and functionality of the component as well as the underlying platform capabilities.

2.3 Example

The model presented in the previous section is closely related to the real practice of using components in application development. For example, assume that we are developing a CORBA object that sorts an array of integers and we are making the source code availabe as a reusable component. We can identify the model elements as follows:

- *Internal*s: The sort mechanism is designed and coded in C++, this represents the private aspects of the components.
- *Application Interfaces (Horizontal Channel*): The application interface will be the Interface Definition Language IDL interface [6] specifying the functionality available (sorting) and its signature. The component can then be called through the middleware i.e the ORB.
- *Platform Interfaces (Vertical Channel*): To run this component on a Windows environment (for example), a subset of the platform interfaces could be specified by:
    - Compile with : C++ compiler for windows
    - Run on: Windows Platform

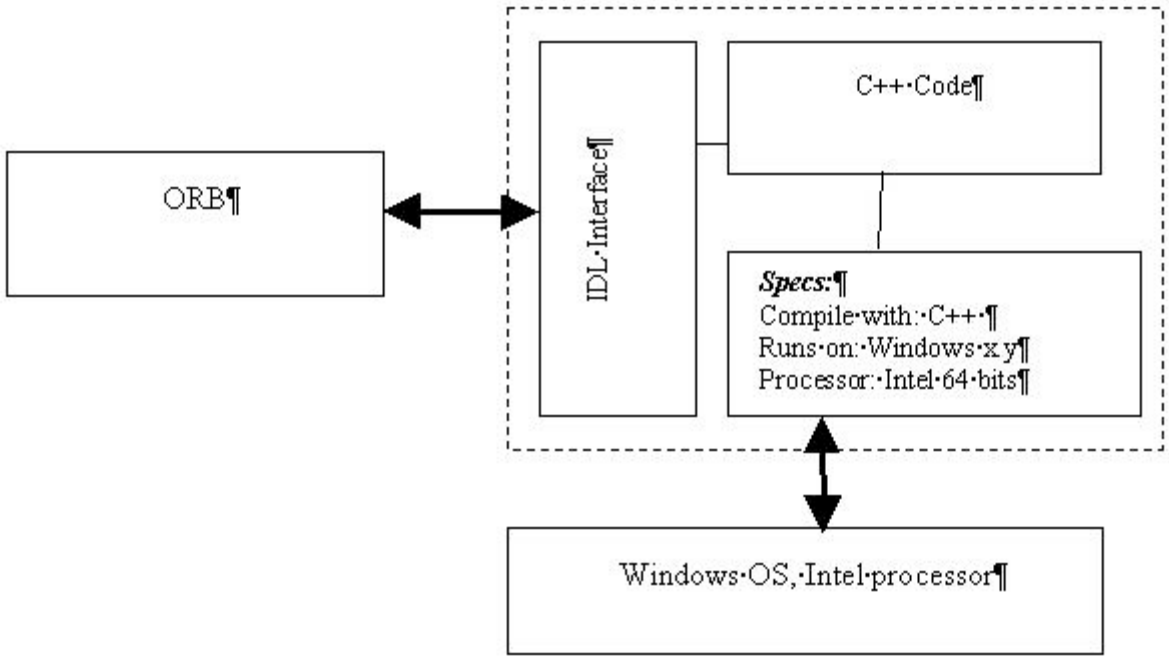


Figure 2 An Example

Now assume that we want to develop the same component in Java.

- *Internals*: The sort mechanism is designed and coded in Java.
- *Application Interfaces (Horizontal Channel*): The application interface will still be an IDL interface.
- *Platform Interfaces (Vertical Channel*): The platform interfaces would include the *Java Virtual Machine* for that specific platform.

3. Impact of Classifying Interfaces

- Establishing a framework for understanding the adequacy of existing notations and languages to specify different types of interfaces. For example one could argue that IDL is adequate for application interfaces, Java Virtual Machines are suitable for platform interfaces, or UML is generic enough for specifying internals and interfaces. *We expect to elaborate on such discussion during the Workshop.*

- Better understanding of interface mismatches. The model separates concerns abou interfaces into two categroies: Issues related to timing and message exchange between components, and issues related to hardware, communications, and other platform related issues. i.e distinguishing portability and inter-operability properties of a component. *During the Workshop, we expect to discuss the Ariane5 problem in the context of this model.*

4. References

[1] Component Object Model home page http://www.microsoft.com/com/dcom.asp

[2] Digre, T.,"Business Object Component Architecture" *IEEE Computer*, Sept/Oct 1998, pp60-69

[3] D'Souza, D. F., and Alan C. Wills "Objects, Components, and Frameworks with UML : The Catalysis Approach" , ISBN 0-201-31012-0 Addison-Wesley, 1998

[4] Heimdahl, M., J. Thompson, and B. Czerny "Specification and Analysis of Intercomponent Communication" *IEEE Computer* Magazine, April 1998

[5] Lutz, R., "Targeting Safety-Related Errors during software Requirements Analysis," *Proc. of First ACM SIGSOFT symposium on Foundations of Software Engineering*, ACM Press, New York 1993, pp95-106

[6] Object Management Group, "The Common Object Request Broker: Architecture and Sepcification" revision 2.2, 1998 http://www.omg.org/corba/corbaiiop.html

[7] Brown, A., and K. Wallnau, "The Current State of CBSE", *IEEE Software*, Sept./Oct. 1998, pp37-46

[8] Kruchten, P. "Modeling Component Systems with the Unified Modeling Language", *First Int'l Workshop on Component-Based Software Engineering*, in conjunction with ICSE'98, Kyoto, 1998

[9] Tai, S., "A Connector Model for Object-Oriented Component Interaction", *First Int'l Workshop on Component-Based Software Engineering*, in conjunction with ICSE'98, Kyoto, 1998

Component Evolution in Product-Line Architectures

Jan Bosch & PO Bengtsson
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden
e-mail: [Jan.Bosch|PO.Bengtsson]@ipd.hk-r.se
www: http://www.ipd.hk-r.se/[~bosch|~pob]

Abstract
The results of a case study investigating the experiences of component-based software development in the context of a product-line architecture are presented. The case study involves two companies, i.e. Axis Communications AB and Securitas Larm AB that employ product-line architectures. The paper discusses the differences between the academic and the industrial view on software components, the problems associated with using reusable components in product-line architectures identified in the case study and, finally, a cause analysis.

1 Introduction
Reusable components have been a goal of the software engineering research community for several decades. Over the years, much research effort has been spent on achieving this goal, e.g. [Weck et al. 97] and [Weck et al. 98]. At least, two important lessons with respect to component-based software development have been learned over the years. First, opportunistic reuse of components does not work and any form of component reuse requires a managed and explicit effort. Second, bottom-up reuse, i.e. plugging together previously unrelated components, has proven not to work in practice, and a top-down approach, i.e. providing a context for components in the form of a component framework [Szyperski 97] or a software architecture, is a necessary ingredient of any successful reuse program.

Within industry, one can identify an increasing use of and interest in, so-called, product-line architectures. Product-line architectures define a common software architecture for a family of products and an associated set of reusable components. These components are generally relatively large entities, up to 100 KLOC, but define an interface and provide several points of variation to cover the product-specific requirements. The components may be commercially bought, but most are developed within the organization. Since the requirements for the products change over time, the requirements on the product-line architecture and, consequently, on the reusable components, change accordingly. This requires the components to evolve, which leads to a number of problems. In this paper, we present the results from a case study in which we have investigated what problems are associated with the evolution of reusable components in product-line architectures.

The remainder of this paper is organized as follows. In the next section, the case study and the case study companies are presented. Section 3 compares the academic and industry views of software components. The problems related to the evolution of reusable components in product-line architectures that were identified during the case study are presented in section 4 and the underlying causes are analysed in section 5. The paper is concluded in section 6.

2 Case Study
The goal of the study was to get an understanding of the problems and issues surrounding the use of reusable components that are part of a product-line architecture in ‘normal’ software development organisations, i.e. organisations of small to average size, i.e., tens or a few hundred employees, and unrelated to the defence industry.

The most appropriate method to achieve this goal, we concluded, was through interviews with the system architects and technical managers at software development organisations. Since this study marks the start of a three year government-sponsored research project on composition and evolution problems of reusable components involving our university and three industrial organisations, i.e. Axis Communications AB, Securitas Larm AB and Ericsson Mobile Communications AB, the interviewed parties were taken from this project. The third organisation, a business unit within Ericsson Mobile Communications, is a recent start-up and has not yet produced product-line architectures or products. A second reason for selecting these companies was that, we believe them to be representative for a larger category of software development organisations. The organisations develop software that is to be embedded in products also involving hardware and mechanics, are of average size, e.g., development departments of 10 to 60 engineers and develop products sold to industry or consumers.

Axis Communications AB develops IBM-specific and general printer servers, CD-ROM and storage servers, network cameras and scanner servers. Especially the latter three products are built using a common product-line architecture and reusable components, i.e. a set of more than ten object-oriented frameworks. The organisation is more complicated than the standard case with one product-line architecture (PLA) and several products below this product-line. In the Axis case, there is a hierarchical organisation of PLAs, i.e. the top product-line architecture and the product-group architectures, e.g. the storage-server architecture. Below these, there are product architectures, but since generally several product variations exist, each variation has its own adapted product architecture.

Securitas Larm AB, earlier TeleLarm AB, develops, sells, installs and maintains safety and security systems such as fire-alarm systems, intruder alarm systems, passage control systems and video surveillance systems. The company's focus is especially on larger buildings and complexes, requiring integration between the aforementioned systems. Therefore, Securitas has a fifth product unit developing integrated solutions for customers including all or a subset of the aforementioned systems. Securitas uses a product-line architecture only in the fire-alarm products and traditional approaches in the other products. However, due to the success in the fire-alarm domain, the intention is to expand the PLA in the near future to include the intruder alarm and passage control products as well. Different from most other approaches where the product-line architecture only contains the functionality that is shared between various products, the fire-alarm PLA aims at encompassing the functionality in all fire-alarm product instantiations. A powerful configuration tool, Win512, is associated with the EBL 512 product that allows product instantiations to be configured easily and supports in trouble-shooting.

3 Comparing Research and Industry Views

An important issue we identified during this case study and our other cooperation projects with industry is that there exists a considerable difference between the academic perception of software components and the industrial practice. It is important to explicitly discuss these differences because the problems described in the next section are based on the industrial rather than the academic perspective. It is interesting to notice that sometimes the problems that are identified as the most important and difficult by industry are not identified or viewed as non-problems by academia.

For components, one can identify a similar difference between the academic and industrial understanding of the concepts. In table 1, an overview is presented comparing the two views. The academic view of components is that of black-box entities with a narrow interface. The industrial practice shows that components often are large pieces of software, such as object-oriented frameworks, with a complex internal structure and no explicit encapsulation boundary. Due to the lack of an encapsulation boundary are software engineers able to access any internal entity in the component, including the private entities. Even when only using interface entities, the use of components often is very complex due to the sheer size of the code. Variations, from an academic perspective, are limited in number and are configured during instantiation by other black-box components. In practice, variation is implemented through configuration, but also through specialisation or replacement of entities internal to the component. In addition, multiple implementations of a component may be available to deal with the required variability. Finally, academia has a vision of components that implement standardized interfaces and that are traded on component markets. To achieve this, there is a focus on component functionality and formal verification. In practice, almost all components are developed internally and in the exceptional case a component is acquired externally, considerable adaptation of the component internals is required. In addition, the quality attributes of components have at least equal priority, when compared to functionality.

Table 1: Academic versus industrial view on reusable components

Research	Industry
• Reusable components are black-box.	Components are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks.
Components have narrow interface through a single point of access.	The component interface is provided through entities, e.g., classes in the component. These interface entities have no explicit differences to non-interface entities.
Components have few and explicitly defined variation points that are configured during instantiation.	Variation is implemented through configuration and specialisation or replacement of entities in the component. Sometimes multiple implementations (versions) of components exist to cover variation requirements
Components implement standardized interfaces and can be traded on component markets.	Components are primarily developed internally. Externally developed components go through considerable (source code) adaptation to match the product-line architecture requirements.
Focus is on component functionality and on the formal verification of functionality.	Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.

4 Problems

Based on the interviews and other documentation collected at the organisations part of this case study, we have identified a number of problems related to reusable components that we believe to have relevance in a wider context than just these organisations. In the remainder of this section, the problems that were identified during the data collection phase of the case study are presented. The problems are categorized into three categories, related to multiple versions of components, dependencies between components and the use of components in new contexts.

Multiple versions of components

Product-line architectures have associated reusable components that implement the functionality of architectural entity. As discussed in the previous section, these components can be very large and contain up to a hundred KLOC or more. Consequently, these components represent considerable investments, multiple man-years in certain cases. Therefore, it was surprising to identify that in some cases, the interviewed companies maintained multiple versions (implementations) of components in parallel. One can identify at least four situations where multiple versions are introduced.

- **Conflicting quality requirements**: The reusable components that are part of the product line are generally optimised for particular quality attributes, e.g., performance or code size. Different products in the product-line, even though they require the same functionality, may have conflicting quality requirements. These requirements may have so high priority that no single component can fulfil both. The reusability of the affected component is then restricted to only one or a few of the products while other products require another implementation of the same functionality.
- **Variability implemented through versions**: Certain types of variability are difficult to implement through configuration or compiler switches since the effect of a variation spreads out throughout the reusable component. An example is different contexts, e.g., operating system, for an component. Although it might be possible to implement all variability through, e.g., `#ifdef` statements, often it is decided to maintain two

different versions.

- **High-end versus low-end products**: The reusable component should contain all functionality required by the products in the product-line, including the high-end products. The problem is that low-end products, generally requiring a restricted subset of the functionality, pay for the unused functionality in terms of code size and complex interfaces. Especially for embedded systems where the hardware cost play an important role in the product price, the software engineers may be forced to create a low-end, scaled-down version of the component to minimize the overhead for low-end products.
- **Business unit needs**: Especially in the organizational model used by Axis, where the business units are responsible for component evolution, components are sometimes extended with very product-specific code or code only tested for one of the products in the product-line. The problems caused by this create a tendency within the affected business units to create their own copy of the component and maintain it for their own product only. This minimizes the dependency on the shared product-line architecture and solves the problems in the short term, but in the long term it generally does not pay off. We have seen several instances of cases where business units had to rework considerable parts of their code to incorporate a new version of the evolved shared component that contained functionality that needed to be incorporated in their product also.

Dependencies between components

Since the components are all part of a product-line architecture, they tend to have dependencies between them. Although dependencies between components are necessary, often dependencies exist that could have been avoided by another modularization of the system or a more careful design. >From the examples at the studied companies, we learned that the initial design of the architecture generally defines a small set of required and explicitly defined dependencies. It is often during evolution of components that unwanted dependencies are created. Based on our research at Axis and Securitas, we have identified three situations where new, often implicit, dependencies are introduced:

- **Component decomposition**: With the development of the product-line architecture generally also the size of the reusable components increases. Companies often have some optimal size for a component, so that it can be maintained by a small team of engineers, it captures a logical piece of domain functionality, etc. With the increasing size of components, there is a point where a component needs to be split into two components. These two components, initially, have numerous relations to each other, but even after some redesign often several dependencies remain because the initial design did not modularise the behaviour captured by the two components. One could, obviously, redesign the functionality of the components completely to minimize the dependencies, but the required effort is generally not available in development organizations.
- **Extensions cover multiple components**: Development of the product-line architecture is due to new functional requirements that need to be incorporated in the existing functionality. Often, the required extension to the product-line covers more than one component. During implementation of the extension, it is very natural to add dependencies between the affected components since one is working on functionality that is perceived as a unit, even though it is divided over multiple components.
- **Component extension adds dependency**: As mentioned, the initial design of a PLA generally minimizes dependencies between its components. Evolution of a component may cause this component to require information from an earlier unrelated component. If this dependency had been known during the initial PLA design, then the functionality would have been modularised differently and the dependency would have been avoided.

Components in new contexts

Since reusable components represent considerable investments, the ambition is to use components in as many products and domains as possible. However, the new context differs in one or more aspects from the old context, causing a need for the component to be changed in order to fit. Two main issues in the use of components in new context can be identified:

- **Mixed behaviour**: A component is developed for a particular domain, product category, operating context and set of driving quality requirements. Consequently, it often proves to be hard to apply the component in different domains, products or operating contexts. The design of components often hardwires design decisions concerning these aspects unless the type of variability is known and required at design time.
- **Design for required variability**: It is recommended best practice that reusable components are designed to support only the variability requested in the initial requirement specification, e.g., [Jacobsen et al. 97]. However, a new context for a component often also requires new variability dimensions. One cannot expect that components are designed including all thinkable forms of variability, but components should be designed so that the introduction of new variability requires minimal effort.

5 Cause Analysis

The problems discussed in the previous section present an overview over the issues surrounding the use of reusable components in product-line architecture. We have analysed these problems in their industrial context and have identified, what we believe to be, the primary underlying causes for these problems. Below, these causes are briefly discussed:

- **Early intertwining of functionality**: The functionality of a reusable component can be categorized into functionality related to the application domain, the quality attributes, the operating context and the product-category. Although these different types of functionality are treated separately at design time, both in the design model and the implementation they tend to be mixed. Because of that, it is generally hard to change one of the functionality categories without extensive reworking of the component. Both the state-of-practice as well as leading authors on reusable software, e.g., [Jacobsen et al. 97], design for required variability only. That is, only the variability known at component design time is incorporated in the component. Since the requirements evolve constantly, requirement changes related to the domain, product category or context generally appear after design time. Consequently, it then often proves hard to apply the component in the new environment [Bosch 99b].
- **Organization**: Both Securitas and Axis have explicitly decided against the use of separate domain engineering units for engineering reusable components. The advantages of separate domain engineering units, such as being able to spend considerable time and effort on thorough designs of components were generally recognised. On the other hand, people felt that a domain engineering group could easily get lost in wonderfully high abstractions and highly reusable code that did not quite fulfil the requirements of the application engineers. In addition, having explicit groups for domain and application engineering requires a relatively large software development department consisting of at least fifty to a hundred engineers.
- **Time to market**: A third important cause for the problems related to reusable components at the interviewed companies is the time-to-market pressure. Getting out new products and subsequent versions of existing products is very high up on the agenda, thereby sacrificing other topics. The problem most companies are dealing with is that products appearing late on the market will lead to diminished market share or, in the worst case, to no market penetration at all. However, this all-or-nothing mentality leads to an extreme focus on short-term goals, while ignoring long term goals. Sacrificing some time-to-market for one product may lead to considerable improvements for subsequent products, but this is generally not appreciated.
- **Economic models**: As mentioned earlier in the paper, reusable components may represent investments up to several man years of implementation effort. For most companies, a component represents a considerable amount of capital, but both engineers and management are not always aware of that. For instance, an increasing number of, especially implicit, dependencies between components is a sign of accelerated aging of software and, in effect, decreases the value of the component. However, since no economic models are available that visualise the effects of quick fixes causing increased dependencies, it is hard to establish the economic losses of these dependencies versus the time-to-market requirements. In addition, reorganisation of software components that have been degrading for some while is often not performed, because no economic models are available to visualize the return on investment.
- **Encapsulation boundaries and required interfaces**: Although many of the issues surrounding product-line architectures are non-technical in nature, there are technical issues as well. The lack of encapsulation boundaries that encapsulate reusable components and enforce explicitly defined points of access through a narrow interface is a cause to a number of the identified problems. In section 3, we discussed the difference

between the academic and the industrial view on reusable components. Some of the components at the interviewed companies are large object-oriented frameworks with a complex internal structure. The traditional approach is to distinguish between interface classes and internal classes. The problem is that this approach lacks support from the programming language, requiring software engineers to adhere to conventions and policies. In practice, especially under strong time-to-market pressure, software engineers will access beyond the defined interface of components, creating dependencies between components that may easily break when the internal implementation of components is changed. In addition, these dependencies tend to be undocumented or only minimally documented.

A related problem is the lack of *required interfaces*. Interface models generally describe the interface provided by a component, but not the interfaces it requires from other components for its correct operation. Since dependencies between components can be viewed as instances of bindings between required and provided interfaces, one can conclude it is hard to visualize dependencies if the necessary elements are missing.

6 Conclusions

The use of reusable components in product-line architectures provides one of the most promising approaches for improving the state-of-the-art in component-based software development. However, the use and evolution of these components still has a number of problems associated with it. In this paper, we have identified these problems based on a case study that we performed at two swedish software development companies. We have analysed the problems and identified the primary causes underlying these problems.

A considerable body of research exists that addresses, at least up to some extent, the problems and causes discussed in this paper. Due to reasons of space, we are unable to include a discussion of related work. However, we refer to [Bosch 99a] for an overview.

References

[Bosch 99a] Jan Bosch, 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study', *Proceedings of the First Working IFIP Conference on Software Architecture,* 1999.

[Bosch 99b] Jan Bosch, 'Superimposition: A Component Adaptation Technique,' Accepted for publication in *Information and Software Technology,* February 1999.

[Jacobsen et al. 97] I. Jacobsen, M. Griss, P. Jönsson, *Software Reuse - Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Szyperski 97] C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1997.

[Weck et al. 97] W. Weck, J. Bosch, 'Proceedings of the Second Workshop on Component-Oriented Programming,' TUCS general publication Nr. 5, September 1997.

[Weck et al. 98] W. Weck, J. Bosch, C. Szypersky, 'Proceedings of the Third Workshop on Component-Oriented Programming,' TUCS general publication Nr. 10, October 1998.

Position paper:

Managing Standard Components in Large Software Systems

Ivica Crnkovic
Mälardalen University, Department of Computer Engineering
Box 883, SE-721 23 Västerås, Sweden
ivica.crnkovic@mdh.se
+46 21 103183

Magnus Larsson
ABB Automation Products AB
SE-721 67 Västerås, Sweden
magnus.ph.larsson@seapr.mail.abb.com
+46 21 342666

Abstract

This position paper consists of two parts. The first part gives an overview of a research project started by ABB and Mälardalen University. The project is concentrated on use of standards technologies and standard components in real-time industry-process systems. The main goal of the project is to increase the knowledge about software development based on standard components from both theoretical and practical points of view. The project can be an interesting case of practices of adopting CBSE, a case of an approach in managing component-based engineering.

The second part of the paper points to some important aspects in use of components at the development, run-time, and maintenance phases. It is a problem of identification and configuration of components in software systems. Configuration Management (CM) disciplines, such as Version Management, Configuration and Build Management, Change Management, etc., are well established for the conventional development. In a component-based development some new requirements on identification and version management arise, and some new methods, similar to those from conventional CM, and possible extensions of the existing methods, should be objects of further research. We propose an extension of CBSE-handbook with a new sub-chapter which is related to component identification and configuration and which could be a part of the Technology for supporting CBSE chapter.

1. A Case study - Use of Standard Technology in Industrial Applications

ABB is a global $30-billion engineering and technology company serving customers in electrical power generation, transmission and distribution, in oil, gas and petrochemicals and, in general, in industrial automation products. ABB employs 200 000 people in over 100 countries.

ABB Automation Products, a $340-million company, is responsible for developing automation products inside ABB and employs 2000 people. The automation products encompass several families of industrial process-control systems including both software and hardware.

The main characteristics of the products are reliability, high quality and compatibility. These features are results of responses to the main customers requirements: The customers need stable

products, running round the clock year after year, which can be easily upgraded without impacts to the existing process. In the recent years the requirements have been, however, somewhat changed. Customers require integration with standard technologies and use of standard applications in the products. This is a high trend on the market but low awareness about the possible problems exists. A improper use of standard component can cause severe problems, especially in the distributed real-time and safety-critical systems, with long-period guarantees. In addition to these new requirements, time-to-market demands become very important factor.

These factors and other changes in software and hardware technology [AOY-98] have introduced a new paradigm in the development process: In the middle of the eighties, ABB control products were complete proprietary monolithic systems with internally developed hardware, basic and application software. In the beginning of the nineties, standard hardware components and software platforms were bought while the real-time additions and application software were developed internally. Now the development process is focused on the use of standard and de-facto standard components, outsourcing, COTS and producing components. At the same time, the final products are not any longer the closed monolith systems, but are instead component-based products that can be integrated with other products available on the market.

This new paradigm in the development process and the marketing strategy has put new problems and questions in the focus:

- The development process has been changed. The developers are not only designers and programmers, they are integrators and marketing investigators. Are the new development methods established, are the developers properly educated?
- What are the criteria for selecting of a component? How can we guarantee that a standard component fulfills the product requirements?
- What are the maintenance aspects? Who is responsible for the maintenance? What are expectations for updating and upgrading of components? How can we manage the compatibility and reliability requirements?
- What is the trend on the market? What can we expect to buy not only today but also what will be present the day we start to deliver our product?
- When developing a component, how can we guarantee that the "proper" standard is used? Which standard will still be valid in five, ten years?

In order to find some answers and to give a theoretical base to the new methods, ABB Automation Products together with Mälardalen University have started a project for research of use of Standard Technology in Industrial Applications (STINA). The main goal of the STINA project is to increase the knowledge of software development based on standard components from both theoretical and practical points of view. The research results will be used both at the university and in the industry. At the university, the accumulated knowledge will be used for further education in order to prepare the students for new aspects in system development. The industry will benefit with direct implementation of methods and knowledge built up in the research activities and well educated students.

STINA is staffed with people both from ABB and the Mälardalen University. In its first phase the research activities are being defined for a three-year period. During that period, or later, some other ABB companies will actively participate in the project. The project is in the initiation

phase, i.e. the activities, milestones and the project results are being planned.

The possible subjects of the research are:

- Technologies for using and developing components. E.g.COM/DCOM, Java Beams, CORBA, Web, Windows 2000 and Linux;
- On-line update;
- Development, run and compose-time configuration management;
- Use of standard components for real-time applications/system;
- Quality assurance and maintainability aspects;
- Reusability of components in both real-time and non real-time systems.

The work will result in several "State of the art" reports, courses, research papers, prototypes and finally Ph.D. thesis.

Our expectations of CBSE are to find standards, disciplines and guidelines so we can investigate them and apply them on the real-time systems of the industry. One objective is also to feed back comments and experience from the industry to the academic world.

2. Standard Components and Configuration Management

In the conventional development/maintenance process Configuration Management (CM) plays an important role. The main purpose of the CM is:

- Identify and manage different versions of source code in a multi-user environment (Version Management and Work Space Management);
- Configure the components and build them (Configuration and Build Management);
- Keep control of changes at a logical level (Change Management).

The software systems based on standard components are results of a combination of pure development and integration of components. The requirements on conventional use of CM remain and new requirements related to component management appear in all phases - in the design, integration and run-time. Especially the integration part becomes important.

We can expect that the source code management becomes less critical, because we expect less internal development. The integration part, i.e. configuration, and version management of the components becomes essential. Change management keeps the same role, but the implementation of the process is different.

The importance of CM, and challenges in research and implementation of CM support, are emphasized of the 1998 CBSE workshop [BRO-98], as quoted: "In particular, high composeability in a product line setting amounts to mass customization and this introduces tremendous configuration management challenges and support challenges."

2.1 Version Management and Configuration Management

In the conventional development, we recognize two different phases - development and run-time.

In the development phase, we design and build configurations that will be used in the run-time. Those parts (typically source code and documentation), which are under intensive change process, are placed under version control. Each item version is identified by a name, version number and different attributes. There is information about who has made a change, when, and often why. A change performed on an item causes a generation of a new item version. In the integration phase, the particular versions of items are selected, defined as elements in a baseline, and used for the system building. In the building process, the new objects, called derived objects, are created from items under version control and from items that make the complete environment (for example, different system libraries, tools, operating systems, etc.). In simpler tools, such as Make, the identification of items outside version control is neglected. In more sophisticated tools, such as ClearMake [LEB-94], we have a possibility to identify the items used in the build process which are outside the version control. This control is however not so precise as for versioned items. It is not supposed that they are changed very often.

In the component-based development we are facing a new situation: The dominant or, at least, the significant parts of building items are components. They can be identified by their names, or by some internal representation, but usually there are difficulties with their version identification. In some cases of components, for example ActiveX, there are possibilities to define a version property, but the management of these properties are limited and not standardized.

Components can easily be replaced and the replacement can be made in an uncontrolled way if not performed cautiously. One common situation is when a new component version automatically replaces another component, which itself is used by some other parts of the system. We can be in a situation where different parts of a software system use different versions of the same component, which can lead to unpredictable system behavior (Figure 1). Even more, a component can be replaced directly in the run-time environment.

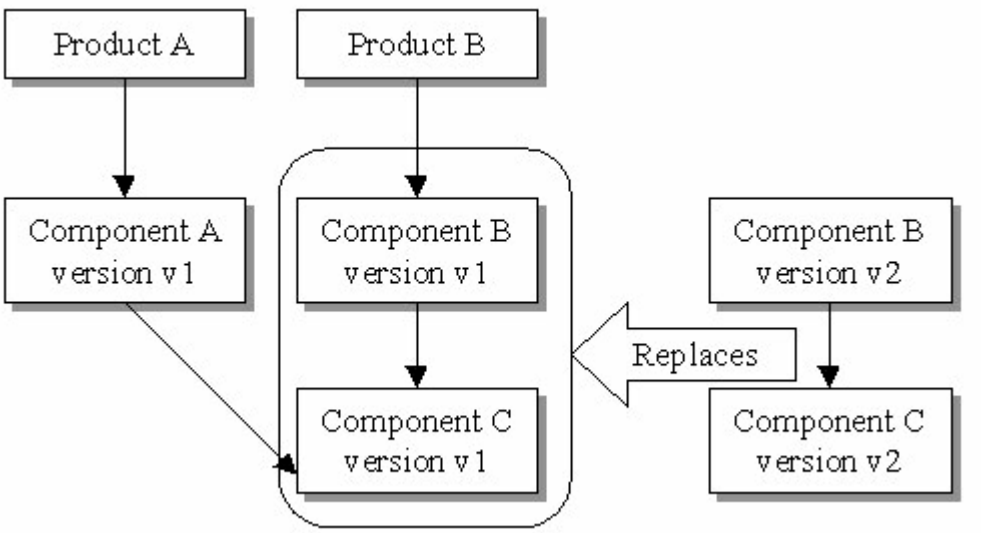


Figure 1. The new version of component B adds version 2 of component A into the system

To minimize the risk to fall into unpredictable situations, we need CM, not only for the parts internally developed, but also for the external components. A minimal requirement is to have a

uniform version identification of the components. The version identification can be applied on components placed in a repository. Such a repository would also include information about other features of components [NAD-98]. The component identification required in the design phase is not sufficient. We also need a mechanism for components identification in the integration (composition) phase, and finally in run-time phase. As different versions of a component may be included in the system, it should be possible to generate a component dependency graph and a non-consistent use of components should be indicated.

2.2 Change Management

The change management process is focused on the logical change introduced in the system and its relation to the physical changes. One purpose of using change management is to simplify the integration process and ensure a logical consistency of changes (i.e. ensuring that all parts being embraced by a change are involved in the integration process). Another purpose is to provide information about the changes introduced in the system - information used by management, quality people, etc. In a component-based development the use of first part will be simplified - it is not expected to be so many items (files) as in conventional development. A change related to a component is in practice a replacement of a component or a component version. The second part becomes more complex. Change Management must consider the questions, such as: What are the reasons for a change, and what are possible consequences of the change?

A component might be updated just because another product uses the new component or it can be required by a new component introduced in the system. A new component version might be added to introduce new functions in the system, or only to change its behavior, (better performance, better stability), but keeping the same interface. When replacing a component or a component version we must consider which type of change is allowed, and which type of compatibility are required.

There are different levels in the compatibility:

- Input and Output compatibility. A component requires input in a specific format and produces result in a defined format. What are the internal characteristics of the component is not of interest. An example of such type of compatibility we can find in different word-processors producing the same document format.
- Interface compatibility (at development time and at run time). The interface remains the same, and the implementation can be different. Typical examples is different implementations of ActiveX objects, with the same interface.
- Behavior compatibility. Internal characteristics of the components, such as the performance, requirements on resources, etc., must be preserved. Such requirements can be appropriate for real-time systems.

The compatibility criteria can be used in the decision process if a component can be replaced or not. This decisions can be especially important by a replacement "on the fly" in a run-time environment. It is important to keep the requited level of the compatibility to avoid a possibility to interrupt the whole system.

3. Conclusion

Configuration Management becomes a more significant part in CBSE. The development is radically reduced, but more efforts on integration, thus CM activities, is required. The CM methods used in the conventional development can be taken as a starting point, but new methods have to be investigated and introduced in CM to efficiently use components. For this reason we propose a new item in the Handbook of CBSE, in chapter 3, Technology for supporting CBSE - Development support, or under Maintenance and Reengineering support.

We expect that the STINA project may contribute to such a work.

4. References

[AOY-98] M. Aoyama: New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development, 1998 International Workshop on CBSE

[BRO-98] Alan W. Brown, Kurt C. Wallnau: An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering, 1998 International Workshop on CBSE

[LEB-94] David B. Leblang, The CM Challenge: Configuration Management that Works, Configuration Management, edited by Walter F. Tichy, Johnn Wiley & Sons, ISBN 0 471 94245-6

[NAD-98] Nader Nada, David C. Rine, A Validated Software Reuse Reference Model Supporting Component-Based Management, 1998 International Workshop on CBSE

Characterizing a Software Component

Sherif Yacoub, Hany Ammar, and Ali Mili
CSEE Department
West Virginia University,
Morgantown, WV 26506

Abstract

This paper discusses the problem of characterizing a software component, which is essential to understand what components are and how they can be classified. First, we classify what should be characterized about a component into three categories: *Informal Descriptions*, *Internals*, and *Externals*. Then, using this classification, we derive a set of features to characterize a software component.

Introduction

CBSE is emerging as a beneficial software development paradigm because it delivers on many of the promises of software reuse. Many would refer to components as Javabeans, CORBA or DCOM objects [12, 4], other would refer to components as fragments of source code or a functional procedure. Several definitions of a software component have emerged. A component can be considered an independent replaceable part of the application that provides a clear distinct function [1]. A software component is a unit of composition, with pre-defined dependencies on other components [13]. Business components represent reusable conceptual artifacts that can be implemented and deployed in large business systems. A component can be a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system[6].

Brown and Wallnau acknowledge in their workshop summary [1] the multitude of definitions of software components and further analyze the variances in terms of granularity, context-dependencies, and autonomy. Our position paper seeks to build on this analysis by proposing an orthogonal set of features that we submit as means to characterize a component for CBSE purposes. Han [15] proposed an object-oriented characterization of components, here we envisage that characterization of components could be further abstracted beyond object oriented definitions and include non-technical features as well.

This paper proposes a set of features to characterize a software component. This set is by no means exhaustive, and shall further mature based on peer discussions in the workshop. Given those features we can understand what are the different types of components, how they are involved in the development lifecycle, how they are reused, and what technologies are required to adopt a component in an application development.

Characterizing a Software Component

We distinguish what needs to be characterized about a component under three main categories: the Informal Description, Externals, and Internals. For each of those sections we define a set of features to characterize a component.

A) Informal Description

This section describes the component human-related issues. Components can be considered as mentally-driven building blocks of applications and hence they should be preserved as solution prescriptions to application development problems. This section would encompass the following characteristics:

1. Age
This is a characteristic of a component that reflects its stability and maturity. When a component is first introduced, there is a high risk associated with its usage. After several instances of usage, the component becomes more mature and hence the reuse risk decreases. This property is adopted in design patterns (or in the pattern community), the rule-of-three establishes the maturity of a pattern by defining three instances in which the pattern was instantiated.

2. Source
Components can be characterized by the source supplying the component, for example, Commercial Off the-Shelf (COTS), Military Off the-Shelf (MOTS), or Government Off the-Shelf (GOTS)

3. Level of Reuse
A component can be reusable at different phases of the development life cycle. The nature of the components plays an important role in determining where in the development phase is the component reusable. For example, a fragment of code is used during development, a Dynamic Link Library (DLL) is reused during run-time, and a static library is reused during integrating the application.

4. Context
The situations in which the component can be used. It is usually a difficult task to determine all situations, general and specific ones, for a component usage. Target domains and applications should be considered when characterizing the context for a component.

5. Intent
This characterizes the purpose for which this component is documented. This includes the problem (or set of problems) that the component solves (or participates in solving).

6. Related Component
This feature defines other well-known components that solve the same (or similar) problems.

B) Externals

Component external define its interactions with other application artifacts and with the platform

on which the component resides. This section would encompass the following characteristics:

1. Interoperability
Interoperability characterizes *Application Interfaces*, i.e. how the component interacts with other components in the same system. A component should be interoperable because it lives with other components in an application environment. The interoperability property necessitates a clear interface definition. This property determines how to call/invoke a component and the interaction direction (Client/Server). We must be able to integrate components with each other and with legacy software to achieve application functionality. Components should be developed as independent as possible from other components, however, it is not usually the case because they are only meaningful in a given interaction context (interaction with other components). The autonomy of a component describes how independent are the components from each other (somehow a measure of de-coupling) which also characterizes the self-containment of the component.

2. Portability
Portability characterizes *Platform Interfaces*, i.e. how the component interact with the underlying platform(Hardware, operating system, communication subsystem, etc.). It is the property of a component that specifies which platform the component runs on and how it can be ported to other platforms. For code components, it is necessary to specify the necessary language compiler. For executable components and link libraries it is essential to indicate the underlying operating system.

3. Role
A component is characterized by its role that identifies what it provides to (requires from) other application artifacts. There are two roles that can be played by a component:

- Active Role: Affect and is affected by other artifacts. ex: GUI
- Passive Role: Affected by/invoked by other artifacts. ex: Databases.

This characterization is essential in determining which component is in control of execution at a given instance of time, a characteristic that helps solving control issues as discussed by Garlan et. al.[8]

4. Integration Phase
This characterizes when a component is integrated to the application. A component can be integrated at the development phase or at run-time. In the development phase it is inserted in the application and compiled to produce the binary executable form of the application. At run-time, components can be loaded and unloaded from the application allowing a on-line replacement.

5. Integration Frameworks
Components are integrated together to form applications, sometimes components do not directly interact with one another but they interact through an underlying framework that virtually connect components with each other [examples 12, 4]. A component should be characterized by identifying the underlying framework on which it runs.

6. Technology

This characterizes the technology on which the component is dependent and the restrictions on the technology for developing the component and for reusing it in other applications. A specification component can be considered technology independent. A general-purpose design pattern as those in [7] are object-oriented technology dependent. Dependability on technology is further discussed in [10]

7. Non Functional Features
Sometime it is essential to denote the non-functional aspects[called *ilities* in 16] of a component specifically for mission-critical domains. Such characterization would include: the component security assumptions (access control, authentication), performance issues (processor requirements and run time requirements), and component reliability.

C) Internals

This section reflects the internal aspects of a component which encompass the following characteristics

1. Nature
The nature of a component determines where the component can be used in the development process. A component can abstract a function, data, package, cluster and system abstraction or a system structure [11]. A component can be a class, a fragment of code, a fragment of design, an executable module, a run time link library (Dynamic link libraries ex. DLLs), a static library, etc.. In general, we classify them components according to their nature as:

- *Design Components*: A component can be a design principle or idea. Design patterns like Observers, Strategy and others[7] can in fact be used as design building blocks in constructing object oriented frameworks [14].
- *Specification Component*: A specification can be considered a reusable component. Specifying the expected functionality and behavior of a component frees the developer to implement this component in a variety of programming languages. An example of using specifications as components was illustrated in [9].
- *Executables*: Executable components can be static libraries, dynamic link libraries (DLLs or VBXs), or executable pieces of an application. Many literatures refer to components of this nature. Usually the source code of those components is not available, and the executable components themselves are commercially available. Most vendors provide (sell) an executable component yet keep the design and implementation code as a proprietary product.
- *Code*: Components can also be code that has been tested and executed in other projects. In many cases, these components will be an in-house library of assets of code that have been used in other applications in the same domain or within the same organization.

The following table shows the development phases and the nature of the components used in each phase. The dependencies on platform, technology, implementation and specification languages are tabulated in each phase.

	Dependency				Component Nature
	Platform (OS, H/W)	Implementation Language	Technology	Specification Language	
Analysis and Specification	N	N	N	Y	Specs
Design	N	N	Y	Y	Design Component
Detailed Design	N	Y	Y	Y	Design Components
Implementation and Coding	Y	Y	Y	Y	Code or Executable module, or Library (DL

Table 1 Nature of a Component

2. Granularity
Digre [5] proposed a possible characterization of components granularity from the business perspective. He classified components as enterprise, domain subsystem, domain object and semantic primitives. Other granularity classification can be based on the size of a component or the phase in which they can be reused.

3. Encapsulation (Decision Hiding)
A non-trivial component should hide (encapsulate) one or more decisions. The decision can be a design decision (design component), an implementation decision (executable component), or a specification decision. A software component should have a significant aggregate functionality and complexity [2]

4. Structural Aspect
The component structure describes the internal participants of the component and how they collaborate. For example, a simple component in object oriented technology can be viewed as a class. For example, a larger grained component such as a design pattern is a macro-component [3] that has the structure of collaborating classes and objects.

5. Behavioral Aspect
The component behavior should characterize two aspects: stateless behavior (the component response to specific set of inputs), and retrospective behavior (the component response to sequence of actions, i.e. components with state)

6. Accessibility to Source Code

Many researchers advocate that components should not be modified because they lose a percentage of trust that was established by the component provider. However, the availability of the component source code is sometimes essential for verification and testing the component at the customer side (the party acquiring the component). The availability of source code determines the accessibility and modifiability of the component. How to use a component and the form of reuse also depend on what is available of the component, usually the question regarding availability of the code is an issue of concern. The following table shows possible reuse component nature and reuse methods depending on availability of code and possibility of modifications.

Source Code Available?	Yes	Yes	No
Source Code Modifiable?	No	Yes	No
How to reuse the component	Extension (Specialization)	Modification of component according to application specific requirement, hot spots, ..etc.	Composition
Form of Reuse	Code (Blackbox)	Code (Whitebox)	Executables,Libraries, (Blackbox)

Table 2 How to use a component depending on the availability and modifiability of source code

Impacts of characterizing a software component

- *Better Cataloguing*: Understanding the characteristics of a component plays a major role in documenting, cataloging, and classifying the wide literature of available components.
- *Better Usage*: Understanding the characteristics and properties of a specific component means better usage and ensures usage of the correct component in particular application development.
- *Better Retrieval* from component libraries: Characterizing components could facilitate selection, acquiring, and acquaintance of components.
- *Better Understanding* of architecture problems: Identifying characteristics of components helps in solving architecture mismatch problems such as the nature of the component and its control mode [8].

References

[1] Brown, W., and K. Wallnau "The Current State of CBSE", *IEEE Software*, October 1998, pp37-46

[2] Brown, A., and K. Wallnau, "Engineering of Component-Bases Systems" in *Component Based Software Engineering* , Alan W. Brown (edt.) Software Engineering Institute, IEEE Computer Society, 1996

[3] Castellani, X., and S. Y. Liao, "Development Process for the Creation and Reuse of Object-Oriented Generic Applications and Components", *Journal of Object Oriented Programming*, June 1998, Vol 11, No.3, pp24-31

[4] Component Object Model home page http://www.microsoft.com/com/dcom.asp

[5] Digre, T.,"Business Object Component Architecture" *IEEE Computer* , Sept/Oct 1998, pp60-69

[6] D'Souza, D. F., and Alan C. Wills "Objects, Components, and Frameworks with UML : The Catalysis Approach" , ISBN 0-201-31012-0 Addison-Wesley, 1998

[7] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Object-Oriented Software," Addison-Wesley, 1995.

[8] Garlan, D., R. Allen, and J. Ockerbloom, "Architecture Mismatch or Why it's Hard to Build Systems out of Existing Parts," Proc. 17th *International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, Ca., April 1995, pp179-185

[9] Iglesias, A., and J. Justo, "Building System Requirements with Specification Components", *Proc. of Joint conference of Information and Computer Science*, JICS'98 , Vol III, pp499-502, Oct. 1998

[10] Kroeker, K., "Component Technology", *IEEE Computer* , Vol 31,132-133, Jan 1998.

[11] Meyer, B. , "On To Components", *IEEE Computer* , Jan 1999, pp139-140

[12] OMG, Corba 2.0/IIOP Specification, OMG Formal/97-09-01, Sept. 1997, http://www.omg.org/corba/corbiiop.htm

[13] Szyperski, C., "Component Software: Beyond Object Oriented Programming", Addison Wesley Longman, 1998

[14] Yacoub, S., and H. Ammar, "Towards Pattern Oriented Frameworks", To appear in *Journal of Object Oriented Programming* JOOP, 1999

[15] Han, J., "Characterization of Components", *First Int'l Workshop on Component-Based Software Engineering*, in conjunction with ICSE'98, Kyoto, 1998

[16] Thompson, C.,"System Wide Properties or Ilities" *Workshop on Compositional software Architectures*, Monterey, USA, Jan. 1998 ,
http://www.objs.com/workshops/ws9801/report.html#II-3

A Position Paper for Second International Workshop on Component-Based Software Engineering

Strawman section – Principles of Adopting Component-Based Software Engineering (CBSE)

1 Introduction

This paper speaks to section 2 "Practices of Adopting CBSE" of the strawman outline. Most of the topics covered fall within Organisational Issues but one candidate new (sub)section is identified.
Top of the Document

2 Which Components First?

Broadly, components can be divided into two groups (as can most of the rest of human experience, at least by someone). The author proposes:

- Business Components - those that the business itself would recognise - e.g. customer, invoice, account, assembly.
- Infrastructure Components - those which (probably) only the IS community would recognise - e.g. audit (in transaction terms), security, error message handling.

CBSE will really show returns for an organisation when its Business Components are identified and "engineered". Unsurprisingly, there are several issues to be addressed.

Firstly, the business needs to be educated about components and the benefits they bring. Secondly, there may be a cultural barrier to overcome about the role of software viz. the need to see software as infrastructure (notwithstanding the author's classification of components above).

Once these two have been overcome, there is a problem, which has analogous forms in more traditional forms of software development. In an organisation of any size, getting disparate user groups to agree on the definition of some Business Components will be a non-trivial task. For example, in a distribution company or a postal authority, there will be several views on an address or even on a postal code. Similarly, in a railway company, the concept of "train" can be an engine (to the maintenance yard) and the 07:15 Monday-Friday from Sheffield to London (to traffic planners and passengers).

The author does not claim this list is exhaustive.

Infrastructure Components may avoid some of the problems associated with Business Components (e.g. there will undoubtedly be a group of developers in an organisation who will want to bring new technology in) but they have their own traps.

The first of these is the effect on the business users. If too much emphasis is made of the introduction of infrastructure components, the business may see components as the latest technical fad. Thus, the introduction of infrastructure components must be handled carefully. However, if managed well, this approach could show a technical "proof of concept". Also, components which are rightly seen as infrastructure but which do interact with business users (e.g. authentication) should render themselves capable of explanation in a way to help convince the business of the benefits of CBSE.

The next problem for the infrastructure component approach is funding. This is also a problem for Business Components but the difficulties are similar in both cases, and may be more difficult to solve if Infrastructure Components are the first to be done. It is likely that a component will cost more to develop than the same functionality written in more traditional methods. This will be difficult to sell to the first project (and others) that uses this component. A possible model comes from a technology (or product) development budget approach where a central (within the IT function) body carries the "risk". This is discussed in more detail in Costs/Benefits/Funding below.

The current marketplace is populated far more with technical (e.g. ORBs) components than with business components. This could suggest that it makes sense to start with infrastructure components. However, as outlined above, the real returns for an organisation will come when business components are deployed. In the end, the approach must be determined by the benefits sought by a project and/or organisation, and the starting point for the migration to CBSE.
Top of the Document

3 Costs/Benefits/Funding

3.1 Objectives

It is vital to be thorough on investigating costs when putting together the business case for the adoption of CBSE. It is also important not to overestimate the benefits, especially for the return on investment. If the costs and benefit are not carefully researched and presented, subsequent CBSE projects may find it very difficult to get funding, and the adoption of CBSE may fail.

An organisation adopting CBSE must also be very clear on objectives. For example, reuse looks very attractive, and if CBSE is done properly, it will come. However, it almost certainly will not come early on. Similarly, flexibility for change should come from successful adoption, but not immediately. The best way to frame objectives is to find out what other organisations have done, where they have succeeded (and struggled, or even failed) and work out what is best for your organisation from these experiences.
Top of the Document

3.2 Possible funding structure

To a large extent, funding structures for CBSE will depend on how the organisation typically invests in IT change and infrastructure (including people and methods).

What will be hardest to achieve, and least likely to succeed, is to ask individual project teams to deliver components out of their funding. The demands on projects make this an almost impossible goal, from the outset. It is better to have some centrally funded activity, whichever way the organisation decides to start with components. (See discussion in Staff/Organisational Issues below).

If the organisation starts on infrastructure components with a single specialist team, (see also the discussion on transition in Staff/Organisational Issues below) it could charge project teams. The charge to projects must be a little less than the team would have spent developing the functionality in a more traditional way. The other approaches can be funded in similar ways, but crucially relieving the project teams of the burden.

For any of these to work, the CBSE team needs some investment until it starts "making a profit". This should come from a technology or product development budget.

A further aspect of funding to be considered is for maintenance costs. See 5 below for a brief discussion of Service Management aspects of CBSE.
Top of the Document

4 Staff/Organisational Issues

4.1 Education and Training

There may be management resistance to education and training - especially as it will involve users at some stage. Technical staff will need a change of mindset as well as technology education - even if they are already using Object Oriented techniques. Giving this sort of training will bring with it a risk of staff turnover. Management must be convinced that the problem can be managed, but the risk still exists and needs to be managed once the adoption is underway. Project Managers will also need training, because CBSE is different for them, as well as for technical staff.
Top of the Document

4.2 Organisation

4.2.1 An Object View

There are different models of adoption from which a choice must be made. Alan O'Callaghan of De Montfort University in the U.K. has written extensively on the issue of making the transition from traditional methods to Object Technology, and the principles apply in CBSE as well. In particular, one paper speaks to the organisational issues to be faced [1]. Where this paper refers to O'Callaghan's article, no attempt to "convert" to from objects to CBSE has been made. However, the principles still apply where an organisation has not moved to Object technology but is considering moving from traditional structured methods straight to Components. This is not as rare as might be imagined. Object technology has not penetrated the market for administrative or commercial software and many such organisations are still deploying Client-Server applications.

The models are variants on a centre of expertise being formed which over time effects skills transfer into the rest of the organisation.

"Loading" the pilot team
Here, a pilot project is chosen with some key well-respected developers, along with more junior staff doing well on

other projects. The project is chosen to be important to the organisation's goals, without being one by which the organisation lives or dies. After the project is complete, the team members will "spread the word" as they disperse through the organisation.

Technology Reception Team
Here, the pilot team approach is used in a much more specialised way. As well as the goals above, the team starts out from the very beginning to train in-house "mentors" for the new approach. This needs to be part of a well-formed migration plan for the organisation. The plan will include not only how the reception team will function, but also the technology (in its widest sense) transfer mechanisms to spread the new methods into the rest of the development (and business) organisation. This transfer, in turn, will need to define what skills and competencies will be needed not only for the staff who work in the new way, but also (perhaps crucially) for those who will effect the transfer. The best technicians might not make the best mentors!

The Object Centre approach
Here, a specialised group is set up as a centre of excellence in the new technology for the organisation. As well as getting the new approach started, over time staff will rotate through the Object Centre. This will spread the new skills etc. through the organisation.
Top of the Document

4.2.2 The "Component Centre"

The CBSE analogue of the Object Centre is the "Component Centre", and the analogy is fairly direct. When considering CBSE specifically, the Component Centre may also "harvest" and "mine" new (even legacy) applications. This takes the burden off the project team to think too hard about components. Instead, the Centre works alongside them, or even slightly after deployment, and examines the application for potential components. The Centre then carries out the work to convert the identified element to a component.

One way of achieving this is to put Component Centre staff into the project team at the equivalent rank of Project Manager and giving them a brief both for harvesting code for reuse and for spotting opportunities for reusing existing code. While the project team itself concentrates on meetings its own application/system deadlines, the CBSE person offers to "buy" suitable components for rework to make them reusable, and to "sell" existing repository components to the project. The implication is some kind of cost centre relationship between the project and the CBSE team.

In addition, the Component Centre can take on the management of the in-house component repository. Indeed, during the transition, it should do so, and there are reuse exemplars that speak to this.

The Centre will keep an eye on the component marketplace so that components can be bought, and, where appropriate, sold for the organisation. This runs alongside the Centre's role in ensuring the order in which an organisation seeks to source each "new" component. Acquisition, then adaptation of existing components should always be considered as preferable to construction.

Top of the Document

4.3 Staff Sensitivities

Some existing roles will almost certainly change, while new roles will be created. People dislike change (not a new insight, the author wishes to point out) so these new structures and practices must be sold to staff. Watch out, for example, for the impact of the Component Centre member seconded into a project team as described in 4.2.2 above.

Life is never simple, however, so organisations making this change may well find that staff feel uncomfortable about change, and yet jealous of those being given the opportunity of new technology. This is, however, not a new

problem. Management experience shows it recurs, with, perhaps, only the backdrop being new.
Top of the Document

5 Support Issues

The Service Management (in its broadest sense – for example as defined in the IT Infrastructure Library) function will be concerned. There are issues about how defects in production components are dealt with when the components are bought in. This problem differs from the issues connected with supporting a bought-in package.

A model for the maintenance costs of components needs to be developed before any components go into production. This will be complicated by the concept of ownership of components, especially once they start to get reused.

A further complication arises in the field of Change Management. Most production defects can be attributed to change in some shape or form, so organisations thinking about Service Management in any depth are trying to understand and manage change. Change will be more difficult to understand, as components get more and more widely used. And bought in components may, at first inspection, make it even more difficult. As with all development methods, this just serves to underline that so-called non-functional requirements need to be considered as early as functional requirements in any project.

The Strawman outline would benefit from a Service Management section.
Top of the Document

6 Project Management

Other workshop participants' papers under the Project Management section may discuss these issues. The issues the author sees with CBSE adoption here result from the fact that the paradigm is so new. There will be a need for new "rules" for managing software projects because there will be new roles and some merging and mixing of existing roles.

The key point here is that Project Managers will need to be convinced that CBSE is different, and then they will need the relevant training.
Top of the Document

7 Metrics

There is also an issue (not that the software industry has got this right yet!) in estimating. By its very nature, estimating is only good when there is a good body of data from which to extrapolate. By definition, a new paradigm cannot have such data, especially within one organisation. Some mechanism for sharing estimates (and actuals, of course) across the industry must be found.

Having said that, some existing techniques may still be valid with, perhaps, just a change of scale. Also if an organisation does construct small components, a database of useful past experience may be built up quite quickly.

Top of the Document

8 References

[1] “Organising for the ‘Personal Shift’”, Alan O’Callaghan, Object Expert 1(3)

Chris Woodhouse
Post Office Research Group
The Post Office
Manor Offices
Old Road
Chesterfield
S40 3QT
U.K.

Tel: +44 (0) 1246 214830
Fax: +44 (0) 1246 523325
Email: woodhouc@postoffice.co.uk

(End of document) Top of the Document

Implementation of CBSE in Small Businesses

William T. Councill
Mannatech, Inc.
600 South Royal Lane
Coppell, TX 75019
bcouncil@mail.mannatech-inc.com

A position paper is proposed for the Second International Workshop on Component-Based Software Engineering

Introduction

The U.S. Small Business Administration defines small businesses as having fewer than 500 employees. Over 99 percent of all U.S. businesses are small businesses, and from 1992 to 1996, small businesses created all of the net new jobs [1]. Yet, reference to the literature on software engineering suggests strongly that implementation of software engineering process models, such as the Software Engineering Institute's Capability Maturity Model (CMM) and ISO 9000, occur among large organizations and on Department of Defense projects.

Additionally, object-oriented and componentware methodologists—such as Grady Booch, Ivar Jacobson, Bertrand Meyer, and James Rumbaugh, among others—generally conduct consulting assignments that eventuate in their articles and books within organizations having more than 500 employees. Numerous small, independent software vendors (ISVs) and information technology (IT) organizations operating in small businesses will adopt component-based development or solutions.

The benefits of software engineering are undeniable. Likewise, once component-based development (CBD) is skillfully adopted, and after the appropriate infrastructure is established, the summary of the first workshop on component-based software engineering [2] instructs that some goals of CBSE are to:

- Enhance reuse of core functionality across applications;
- Permit the flexible upgrade and replacement of parts of software independent of their production (manufacture); and
- Provide the means for establishment of collective organizations' best practices so that extant processes may be replaced because of changing business or market-driven scenarios.

Many, if not most, of the business components developed for numerous industries will be developed by small businesses as defined by the U.S. Small Business Administration. The issue is this: How can small business software organizations assure quality and trusted components that comport with all established or prevailing standards, whether those components are internally developed or purchased from third parties?

Software Engineering and Small Business

Small companies, especially those in the range of 200 to 500 employees, often develop complex software to assure their companies' competitive positions. This software is often perceived as mission-critical; that is, businesses cannot succeed without the software. Software engineering in small companies, especially entrepreneurial organizations, must realize the needs of continually more erudite market experience. To survive, many small companies interpret market changes as requirements for modifications to business processes and business rules, thus necessitating frequent, rapid revisions to the companies' core

software.

Software engineering processes, including the CMM, generally are based on experiences with DOD contractors and large corporations for whom change occurs more slowly than in small companies. In large organizations that adopt change management, change is perceived as a deliberate, methodical process. In small businesses, change is considered a "make-or-break" situation; the competitive window of opportunity is perceived—often rightly so—as narrow and one that must be traversed immediately.

Nevertheless, software engineering is required to assure consistent access to the software. Strict software engineering guidelines--for example, adherence to the key process areas (KPAs) of the CMM--are considered necessary to assure, at a minimum, a repeatable, controlled, and well understood, software development process. Many small companies, especially those that do not function as subcontractors for DOD projects, consider software engineering a hindrance to efficient, realizable development of software.

Additionally, the roles identified by Paul Allen in his article, "Planning Team Roles for CBD," [3] are daunting for small businesses. Allen identifies an array of roles that, ideally, should be assumed by members of any component development team and support personnel to achieve optimal results [4]. That article relies greatly on work published by the Dynamic Systems Development Method consortium (DSDM) [5].

The roles are effectively delineated, and Allen states that roles may be assumed by various employees at different times. The problem is that many small organizations have insufficient numbers of employees to assure that employees assigned to various roles will function in each role well. For example, an experienced configuration management engineer likely will have assumed too many existing roles to learn to function effectively in the new role of reuse librarian.

Budgets in many small companies, despite the commitment to component technologies, will not support the recruitment of a reuse librarian. Yet, many small companies necessarily will adopt CBD. How can CBSE develop so that small companies can assure themselves, as well as purchasers of their components, of the confidence and trust in their components?

CBSE and Small Business

CBD for small businesses makes sense. Many small businesses are unlikely to purchase enterprise resource planning (ERP) software because of the great expense and significant number of personnel required for implementation. Small companies, for example, may anticipate the purchase of a third party order processing component to build into a proposed new application. Or, such companies may develop a business component and, to further fund their IT functions, likely will develop generic versions of the business component for resale to other, non-competing companies with similar business processes.

Nevertheless, purchasers of third party componentware must trust their business processes to components developed by others. Components can only be trusted when, especially considering components developed using object-oriented techniques, the following requirements converge:

1. The architecture demonstrates adherence to the basic laws and principles of object-oriented analysis, design, and programming [6]:

    a. The Law of Demeter – Avoidance of coupling a client to knowledge of indirect objects and the internal representations of direct objects.

    b. Principle of Low Coupling – Assurance that a class is not dependent on too many other classes. Enhances reusability.

    c. Principle of High cohesion – A measure of how strongly related and focused the responsibilities of a class are. Enhances maintenance, comprehension, and reusability.

2. Services can be accessed only through a consistent, detailed, published interface that represents a contract between the provider of the capability and potential consumers. This is routinely referred to as component encapsulation.

3. New interfaces can be defined to service new requirements with minimal disruption to the component's internals and to component consumers.

4. CBSE is demonstrable through:

    a. Repeatable processes (at a minimum, as defined by CMM, Level 2);

    b. Documentation of all phases of the software development life cycle;

    c. Configuration management of code; and

    d. Separate, verifiable component management, with automated search capabilities on metadata.

The Challenge

Components will require a great degree of trust; otherwise, as Bertrand Meyer states, "the spread of less-than-optimal components could lead to a *worsening* of the [software development] situation [7]. In that article he refers to another of his articles, concerning the Ariane project, where reuse of an improperly specified component (module) created an industrial disaster [8]. Code libraries from sources other than compiler vendors generally will be reviewed with suspicion; the code is likely to be readily investigated. The question that must be answered by Workshop participants is: Against what standards will components with an acceptable, published interface be adjudged, thereby permitting the purchaser a reasonable sense of trust?

The author proposes that the admonition "Caveat Emptor" ("Let the buyer beware") is an insufficient standard and one that promotes lawsuits in the event the purchased software does not work as published. Rather, it is recommended that all vendors of business components establish a warranty attesting that basic software engineering principles have been adopted for the general design, construction, and testing of the component. Since many business components, it is proffered, will be developed by ISVs and other small businesses, how can the quality, merchantability, and "fitness for a particular purpose" be assumed, without recourse to the courts?

The Butler Group has introduced the concept of CBD levels of maturity [9]. The issues are similar to those for the CMM. However, what is the assurance that ISVs and small business IT organizations will adopt the proprietary model, or even become aware of the Butler Group's attempts to propose such a model?

The author proposes for the Workshop's consideration an approach for certification of components analagous to the international Underwriter's Laboratories, Inc. model. For those vendors that volunteer to participate in component certification, publication of compliance, as well as all known defects, should be prescribed. The organization's approach to software engineering, and especially the testing of components, likewise would be published.

A separate standards body would establish minimal standards for acceptable types of defects and the numbers of errors in their internals, interfaces, and metadata. Components then could be tested in UL-type certification laboratories according to standards promulgated by approved national or international standards' organizations. Rigid testing would assure vendors' or publishing users' intent to offer for consumption a trusted component that is reliable and reusable.

Summary

Small businesses require implementation of software engineering processes to assure continuous access to reliable software. Yet, the business dynamics of small businesses necessitate a review and likely

revision of software engineering practices that support their needs for frequent, rapid change. The development or purchase of reliable, reusable components has the potential to assist small businesses with dynamic and sudden business, and therefore, software change. Component technology can assure quality software, but only once standards for trusted components are established and testing facilities are established for those organizations, both small and large, that desire to submit to the voluntary process of certifying their components.

References

[1] Small Business Administration, "Small Business Profile, 1998." www.sba.gov/ADVO/stats/profiles/98us.html

[2] Brown, A. W. and K.C. Wallnau, "An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering," 1998. www.sei.cmu.edu/cbs/icse98/summary.html

[3] Allen, P. And S. Frost, "Planning Team Roles for CBD," *Component Strategies*, (August 1998). www.selectst.com/downloads

[4] Allen, P., "Practical Strategies for Migrating to CBD," *Cutter IT Journal*, Vol. 11, No. 12 (December 1998).

[5] DSDM Consortium, *DSDM Version 3*, Tesseract Publishing, Farnham, UK, 1997.

[6] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

[7] Meyer, B., C. Mingins, and H. Schmidt, "Trusted Components for the Software Industry." www.eiffel.com

[8] Jezequel, J-M. and B. Meyer, "Design by Contract: The Lessons of Ariane." www.eiffel.com

[9] Wilkes, L. and D. Sprott, "Understanding Component-Based Development Through the Butler Group Maturity Model: What Level Are You At?" *CBD Forum Journal* (June 1998).

Solution Concepts for the Optimal Selection of Software Components

Salah Merad and Rogério de Lemos
Department of Computer Science
University of Newcastle upon Tyne, NE1 7RU, UK
{salah.merad, r.delemos}@newcastle.ac.uk

Abstract

We consider the problem of the optimal selection of software components from a library of components with respect to some non-functional attributes. Two potential solutions are presented, one based on utility theory, and the other based on a game theoretic formulation of the problem.

1. Introduction

In this position paper, we consider the problem of selection from a library of several components which have the same functionality, but differ in their non-functional attributes (NF-attributes). Examples of NF-attributes are reliability, performance, availability, which together specify the quality of service provided by a component, and cost. In the selection process, a trade-off between cost and quality of service has to be made.

A similar multi-criteria decision problem was addressed by Kontio /Kontio 96/ for which he recommended using the Analytic Hierarchy Process developed by Saaty /Saaty 90/. We propose two solution procedures: the first one is based on the construction of a combined utility function that quantifies the user's preference patterns over the available components, and the second one is the result of a game theoretic formulation of the problem. In the latter, a group or subset of components that have the same functionality is to be selected, and we propose an index which is a measure of the trade-off between the overall quality of service to be provided by the group and the total cost of the software to be built. We briefly discuss the conditions under which each of the solutions is applicable and the criteria which need to be considered to evaluate the merits of each solution procedure.

2. Optimal selection of components

Given a user's non-functional requirements (NF-requirements), the problem is to select the "best" component or subset of components from a library of components with respect to their NF-attributes.

2.1. The model

Let $\mathbb{A} = \{A_1, A_2, \ldots, A_M\}$ be the set of components which satisfy the user's requirements. Let $X_1, X_2, \ldots, X_N$ be N NF-attributes that specify the quality of a component. Let a_{mn} be the level of attribute X_n in component A_m, $1 \le m \le M$ and $1 \le n \le N$. Then, vector

$\mathbf{a}_m = (a_{m1}, a_{m2}, \ldots, a_{mN})$ represents the profile of component A_m.

We distinguish two types of NF-requirements, strict and flexible. Without loss of generality, we define these terms for an attribute X_n in which the user's preference is for high levels over low levels. A NF-requirement for attribute X_n is strict when the level of the attribute in a component has to be equal or higher than a critical level cv_n, and it is flexible when the level can be lower than cv_n, though the user aspires for higher levels. The user's strength of preferences over the levels of attribute X_n are summarised in a utility function $u_n(x_n; cv_n)$ whose expression is found empirically. The user's NF-requirements are specified by the vector $\mathbf{cv} = (cv_1, cv_2, \ldots, cv_N)$ and the individual utility functions $u_n(x_n; cv_n)$, $1 \le n \le N$.

Let c_m be the cost of component A_m, and c^* the total amount of money available for expenditure on components. Without loss of generality, we assume that $c_m \le c^*$ for all $1 \le m \le M$.

2.2 Solution procedures

This is a multi-attribute decision problem for which there exists a large literature /Fishburn 70, Keeney 76/. For this problem, there is no obvious solution concept because there are many attributes, some of which are conflicting. We consider two solutions: one is an extension of the Von Neumann one-dimensional utility function to several dimensions, and the other is the result of a game theoretic formulation of the problem.

2.2.1. Combined utility function

It is shown that, under some independence assumptions between the attributes, it is possible to represent the user's preferences between alternatives with a combined utility function (CUF) $U(x_1, x_2, \ldots, x_N)$ which has a simple expression. For example, under the assumption of preference independence, in which the user's preference pattern over a subset of attributes is independent of the levels of the complementary subset, the CUF has the additive form

$$U(x_1, \ldots, x_N; cv_1, \ldots, cv_N) = \sum_{n=1}^{N} k_n u_n(x_n; cv_n),$$

where $k_1, k_2, \ldots, k_N$ are scaling coefficients through which the user's value trade-offs between the attributes are evaluated empirically. Note that both the individual utility functions and the CUF are scaled from 0 to 1. The hardest task is the evaluation of the scaling coefficients which is done through what are called indifference experiments. We illustrate such experiments with two attributes X_1 and X_2 in which the lowest levels are x_1^I and x_2^I, respectively. Let x_1 be a possible level for attribute X_1. The user is then asked to find the level x_2 for attribute X_2 such that he/she is indifferent between the two components which have profiles (x_1^I, x_2) and (x_1, x_2^I), respectively. The profiles (x_1^I, x_2) and (x_1, x_2^I) are thus equivalent and yield the same utility. It is easy to verify that, by combining the equality of the utilities with the form of the utility function, we obtain a linear equation. For N attributes, we need N linearly independent equations, and hence at least N indifference experiments need to be conducted. Because the evaluation process is subjective, it often yields inconsistencies which are difficult to eliminate

even after repeating the process many times. Note that the cost of components is one of the attributes, and hence the trade-off between cost and quality is incorporated in the CUF.

Kontio considered a similar problem and investigated two solution methods, the Weighted Scoring Method (WSM) and the Analytic Hierarchy Process (AHP) /Kontio 96/. He recommended the latter method because it is theoretically sound, it constructs the user's utility function for each attribute and it has a strong empirical validation. It is true that the AHP is superior to the WSM, but it has some shortcomings. The AHP is a method which yields an additive aggregate utility function which has the same form as the combined utility function under preference independence given above. In this aggregate utility function, the scaling coefficients are replaced by the weightings of the attributes, and the scaled individual utilities of the CUF are replaced by utilities evaluated by pairwise comparisons of the alternatives for each attribute separately. These utilities are in fact weights which add to 1, and which represent ratio scale preferences between the alternatives. Hence, although the AHP can include the cost of the components in the aggregate utility, it does not incorporate the user's value trade-offs; it is a heuristic which approximates the user's true preferences under the preference independence assumption. The AHP has an advantage in that there are simple statistical tests to check for consistency in the evaluation of the weighting coefficients and the individual utilities, but the power of these tests decreases rapidly with the number of alternatives and attributes.

2.2.2. A game theoretic solution

As mentioned above, the combined utility function requires the tedious task of the evaluation of scaling coefficients. Moreover, the NF-requirements for some attributes may be determined by the environment and hence uncertain (e.g., memory available to run an application at a given time). For these reasons, we need to define another solution concept which is appropriate for dealing with the uncertainties associated with the environment of a component based software system.

Note that although there may be uncertainty regarding the NF-requirements of some attributes, we assume that the user can quantify their relative likelihood with a completely specified probability distribution.

In this solution, we assume that, subject to the user's purchasing power, a group of different components all fulfilling the functional requirements is to be selected according to some criterion. The main issue here is how to evaluate a group of components. To distinguish between the quality of service provided by a group of components and their total cost, the cost attribute is not included among the N attributes.

Let Ω be the set of subsets of A whose total cost does not exceed c^*. Let S be an element from Ω containing the K components $A_{m_1}, A_{m_2}, \ldots, A_{m_K}$, and C_S its cost. Note that C_S can be more than the total of its constituent components, because we need to add the cost of putting them together.

In any subset, there may be more than one component which satisfies the NF-requirements. If we try to optimise simultaneously all the attributes, then it can be shown that this selection problem can be formulated as a bargaining game whose solution achieves a compromise between the preference patterns of all the attributes /Merad 99/. The solution is obtained by maximising a function, called the Nash product, over part of the boundary of a well defined convex set - see /Nash 1950, Luce 57/. It is a randomised procedure in which the components of the subset are selected with some probabilities during run-time. The Nash product is

$$F(v_1, v_2, \ldots, v_N) = \prod_{n=1}^{N} v_n^{\omega_n},$$

where v_n represents the expected utility for attribute X_n, and ω_n is the weight allocated to attribute X_n, with $0 \le \omega_n \le 1$ and $\sum_{n=1}^{N} \omega_n = 1$. These weights are evaluated empirically.

Let $\Delta_S^*(\mathbf{cv}) = \left(\delta_{m_1}^*(\mathbf{cv}), \delta_{m_2}^*(\mathbf{cv}), \ldots, \delta_{m_g}^*(\mathbf{cv}) \right)$ be the resulting optimal selection strategy within the subset during run-time under NF-requirements $\mathbf{cv}$. The elements of vector $\Delta_S^*(\mathbf{cv})$ are the probabilities with which the respective components are selected. We propose to characterise the subset S by the index

$$I_S(\mathbf{cv}) = \left(\prod_{n=1}^{N} \left(E_{\Delta_S'(\mathbf{cv})}(x_n) \right)^{\omega_n} \right) / C_S,$$

where $E_{\Delta_S'(\mathbf{cv})}(x_n)$ is the expected level of attribute X_n with respect to probability distribution $\Delta_S^*(\mathbf{cv})$.

If the subset contains only one component, or only one component in the subset satisfies the NF-requirements, then $E_{\Delta_S'(\mathbf{cv})}(x_n)$ must be replaced by the level of the component for attribute X_n. If no component in the subset is feasible under requirements , then the index takes value 0. The index under all possible requirements, denoted by I_S, is then the average index under all possible requirements, that is

$$I_S = E\left(I_S(\mathbf{cv}) \right),$$

where E is an expectation with respect to the distributions of the unknown requirements. If there are no attributes for which the NF-requirements are unknown, then the index is $I_S(\mathbf{cv})$.

Note that this index is to be computed during the design phase, and the subset with the highest index in the one that should be purchased.

Example: Suppose that there are two attributes specifying the quality of service provided by a component, each of weight 0.5:
X_1 represents the reliability of a component, and it is measured in average failures during a given length of time;
X_2 represents performance, and it is measured in the time taken to execute a number of tasks.

The following table gives the profiles of 3 components, together with the user's critical values and utilities which appear between brackets in bold:

	Reliability X_1	**Performance** X_2
Critical Values Components	1	0.8
A_1	0.85 **(1)**	0.80 **(0)**
A_2	0.90 **(0.5)**	0.75 **(0.5)**
A_3	0.95 **(0)**	0.70 **(1)**

Consider the two subsets $S_1 = \{A_2\}$ and $S_2 = \{A_1, A_3\}$. In the latter subset, the Nash product is maximised by choosing each of components A_1 and A_3 with probability 0.5 /Merad 99/. The expected level of reliability is 0.9 and the expected level of performance is 0.75. The indices of the two sunsets are: $I_{S_1}(1,0.8) = (0.9 * 0.75)^{1/2} / C_{S_1}$, and $I_{S_1}(1,0.8) = (0.9 * 0.75)^{1/2} / C_{S_1}$. If the cost of component A_2 is less than the total cost of subset S_2, then component A_2 is preferred.

3. Conclusion and future work

We have proposed two solution concepts to the optimal selection of components under scarce resources, and there are others simpler but ad-hoc procedures /Merad 99/. The solution concept that should be adopted depends on the information the user can provide, the practicality of the computations, especially during run-time, and most importantly the trade-offs between the computational effort and the gain in utility over simpler but cruder methods. But the best way of carrying out such an evaluation is a properly designed experiment in a real situation using real decision makers /Chankong 83/.

References

/Chankong 83/ V. Chankong and Y.Y. Haines. *Multiobjective Decision Making*: *Theory and Methodology*. North-Holland, 1983.

/Fishburn 70/ P.C. Fishburn. *Utility Theory for Decision Making*. Wiley, New York, 1970.

/Keeney 76/ R.L. Keeney and H. Raiffa: *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley, New York, 1976.

/Kontio 96/ J. Kontio. "A Case Study in Applying a Systematic Method for COTS Selection". *Proceedings of the 18th International Conference on Software Engineering*. Berlin, Germany, March 25-30, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996. pp. 201-209.

/Merad 99/ S. Merad, R. de Lemos and T. Anderson. *Dynamic Selection of Software Components in the Face of Changing Requirements.* Technical Report No 664, Department of Computing Science, University of Newcastle, UK, 1999.

/Nash 50/ J.F. Nash: The Bargaining Game. *Econometrica*, 18, 1950. pp. 155-162.

/Saaty 90/ T. L. Saaty. *The Analytic Hierarchy Process.* McGraw-Hill, New York, 1990.

/Von Neumann 47/ J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Wiley, 2nd edition, 1947.

Building Maintainable Component-Based Systems

Dr. Mark R. Vigder
National Research Council Canada
mark.vigder@nrc.ca

Abstract

Maintaining large software systems that are constructed from pre-built components is expensive and one of the major cost drivers for these systems. By identifying the activities associated with maintaining component-based systems, and then designing systems that facilitate these activities, maintenance costs can be reduced. Designing maintainable systems requires a specific set of design criteria that are to be followed, and a checklist of items that can be used during design inspections in order to verify that the design criteria are being adhered to.

1. Introduction

The traditional approach to a discussion of maintenance issues focuses on the ideas of adaptive, corrective, preventative and perfective maintenance. These discussions, and any conclusions reached, assume that the maintainer has full access to the source code from which the system is constructed. However, many modern software systems are a built from software components where these components are a combination of custom built software and Commercial Off-the-shelf (COTS) software that are acquired and evolved over a long period of time. With the use of third-party components the system manager does not have access to source code (often a blessing) nor direct access to the developers.

Maintenance of these software systems has become a significant cost driver in an organization's overall IT costs. Many of the maintenance activities are made more difficult due to the use of components. Maintaining the system involves managing a number of black-boxes that are supplied by many different vendors that were not necessarily designed to work together.

Maintenance of component-based systems differs from maintenance of custom-built systems in the following ways:

System developers do not have access to the source code. Without access to source code, maintainers must find novel ways for troubleshooting and supporting a system, test the system, and change and modify the functionality of a system.

Maintenance and evolution of the component is controlled by a third party. System developers are one more customer of the component developer. It is the component owners who control the evolution and maintenance of the individual components.

Maintenance is done at the component level rather than the source code level. Rather than fixing

components and modules, maintenance of component-based systems involves replacing/adding/deleting components rather than source code changes.

This position paper discusses three main issues related to maintenance: Section 2 identifies the major maintenance activities of component-based systems; Section 3 describes architectural properties that can be built into a system that support the maintenance activities; and Section 4 outlines how an inspection checklist can be used during system construction to assist in building a maintainable component based system.

2. Component-based maintenance activities

The activities of the maintenance process for component-based systems are activities associated with the component level rather than with source code level. These activities include the following.

Gluing and wrapping. Components are generally not "plug-and-play". Significant effort is required to build wrappers around components and the "glue" between components in order that they can work together. As systems evolve this wrapper and glue code must be maintained.

Tailoring. Components provide generic functionality. Organizations must tailor this generic functionality to correspond to their unique business requirements. Such tailoring can be done through various technologies without touching the source code of component, e.g., scripting, plug-ins, and frameworks. As businesses modify and update their processes, maintenance of the software system must be performed to reflect the modified business processes.

Fault identification and isolation. When a system fails, system maintainers can no longer fix the problem by changing the source code. Fixing the code is the responsibility of the component builders. The maintainers must deal with the failure by identifying the component (or set of components) that have the fault, and then by isolating the fault and finding workarounds in order to continue using the system.

Updating component configuration. One of the major activities of maintaining component-based systems is the effort required to upgrade component configurations. This includes: replacing components with newer versions as they are released by the component developer; substituting similar components from different vendors; adding or deleting components as the requirements of the system change.

Monitoring and auditing system behaviour. Any system must be monitored in an ongoing manner in order to evaluate issues such as performance, process improvements, failure detection, and usage as well as to assist in troubleshooting. For component based systems this requires a system manager to be able to monitor the load on each component, its failures, component performance, how components are being used, and where they are being used.

Component testing. System maintainers are continually adding and upgrading components within a system. Before adding components, system maintainers are required to perform extensive component testing to determine the effect of integrating the component into the system. The testing must be done to determine behaviour of the component, differences from

previous version, performance, resource requirements, etc.

3. Software architectures that facilitate maintenance activities

Overall system lifecycle costs of component based systems can be reduced by building systems in such a way that they facilitate the maintenance activities that were outlined in Section 2. By building a system with appropriate architectural properties, maintainers will have techniques available to perform the activities. Following is a summary of architectural properties that should be built into component-based systems to enhance their maintainability.

Service level tuning. Maintenance personnel must be able to easily modify the user services of the system. As an organization fine tunes its business process, it must update the software that embodies the process, i.e., maintainers must be able to tune the services that the system provides. Since components cannot generally be modified directly (and should not be modified) it is the responsibility of system designers to build component-based systems where the user services can be tuned without component modification. This can be done through the selection of components that allow for tuning (e.g., through scripting or plug-ins) or by a software architecture that separates generic functionality provided by components from the specific business process functionality that can be provided by modifiable mediators or scripts.

Flexible component configuration. A flexible component configuration refers to the ease with which maintainers can add, delete, replace, and upgrade components. Factors affecting the flexibility include the level of component coupling, interconnection architecture among components, and the use of wrappers and glue code to isolate components.

Visibility. In order to assist in fault identification and isolation, troubleshooting, and system monitoring, it is necessary that system maintainers have visibility into system behaviour. Since components themselves are black boxes, visibility must be provided at the edges of the boxes, in the wrappers and glue code. Developers must build systems in such a way that monitoring and instrumentation facilities are provided at appropriate points in the architecture.

Fault isolation and exception handling. A software architect cannot control when or how a component will fail, but they can control how failures are detected and what happens in the event of a failure. For maintenance purposes, it is important to design systems such that failures are detected as soon as they occur, are isolated to the components in which they occur, an exception is generated for every failure, and recovery and reporting is handled by the system architecture.

Open system. An open system, in this context, is one that can be expanded and included as part of a larger system. Openness is achieved through a number of means, for example use of standards for interfaces, standards for middleware, exported and documented interfaces and data schemas, etc.

Appropriate integration architecture. System builders do not have ownership of the components of the system; they do however have ownership of the system architecture. Architects must focus on constructing a system that assists integration by: minimizing component coupling; allowing for concurrency control; and provide instrumentation capability.

4. Constructing maintainable component based systems

In order to build a system with the properties outlined in Section 3, a mechanism must be put in place that verifies the existence of the properties during the construction and evolution of the system. One mechanism that can be used to verify these properties is to identify a specific checklist of items that can be verified in the design and implementation of the system. A checklist can be used during the inspections or walkthroughs of the software design; if the inspection verifies the items on the checklist are included then the system will possess the desired properties.

Categories that should be included as part of the inspection are listed below. A more detailed description can be found in [1,2].

Connector infrastructure. Since connectors provide the infrastructure for communication between components, they impact the ease of integration and operations of the system. The objectives in inspecting the connector architecture are: determine whether components can be easily moved to allow for reconfiguration of the system; verify that components can be easily added, removed, and substituted; maximize the selection of third party component suppliers who can provide components that are compatible with the system; determine the level of visibility the connector provides for the purposes of troubleshooting and testing.

Interconnection topology. The objective of evaluating the interconnection topology is to minimize the number of interconnections and to simplify the interconnection patterns. Understanding the topology is a necessary condition for understanding component dependencies and performing operations such as developing test plans and system troubleshooting. Simplifying the topology eases the task of substituting components and integrating new components into the system. It also provides an insulating mechanism between components by isolating functionality.

Interfaces. The objective of evaluating the interfaces is to verify that interfaces provide the following: interfaces remain independent of the underlying component so that component substitution is possible; interfaces facilitate the instrumentation, monitoring, testing, and troubleshooting of systems; and interfaces are documented, versioned, and under configuration management.

Tailorability. The objective is to have a system that can be quickly and easily tailored to meet new or updated requirements. A checklist must verify: the tailorability of individual components; tailorability of the glue code; and tailorability of the architectural style.

Architectural style. The objective of the architectural style evaluation is to verify that: an architectural style for the system has been defined and documented; the style is appropriate for the long-term evolutionary goals of the system; components used to construct the system are consistent with the architectural style; the architectural style is being maintained during the initial development as well as during the maintenance of the system.

Run-time instrumentation. The purpose of evaluating the instrumentation is the following: determine whether integrators and users have an adequate instrumentation capability; determine

what level of instrumentation is available from the architecture; determine the usefulness of standards for instrumentation.

Collaborations. The objective in inspecting the component collaborations is to determine the dependencies between components and to verify that system services can be added and modified.

Configuration management. Since much of the maintainability of component based systems depends on an appropriate configuration management process to manage the evolution and configuration of components, it is important to evaluate the CM processes and tools. It must be verified that the CM process can: determine the component versions installed at each site; track the history of updates to components at each deployed site; and record known compatible and incompatible sets of components.

Component substitution Evaluation of the architecture should include a number of component substitution exercises to verify that component substitution is possible, and determine the level of effort required to substitute components.

5. Conclusions

Maintenance activities associated with component based systems are different from traditional systems in that maintainers are dealing with components that are black-box, they do not control, and which they exercise minimal control over how they evolve. In order to reduce life cycle costs, architects must design the system in such a way to facilitate the activities associated with maintaining component based systems. This can be done by identifying the architectural properties that are desirable in such a system and then developing a detailed checklist that can be used during inspections and/or walkthroughs to verify that the properties are being built and preserved in the system.

[1] M. Vigder, *Inspecting COTS Based Software Systems, Verifying an Architecture to Support Management of Long-Lived Systems*, NRC Report No. 41604, 1998. (Available at http://wwwsel.iit.nrc.ca/projects/cots/COTSpg.html).

[2] M. Vigder, *An Architecture for COTS Based Software Systems*, NRC Report No. 41603, 1998. (Available at http://wwwsel.iit.nrc.ca/projects/cots/COTSpg.html).

The Magma Approach to CBSE

Svein Hallsteinsen and Geir Skylstad,

SINTEF Telecom and Informatics

N-7034 Trondheim

Norway

{svein.hallsteinsen, geir.skylstad}@informatics.sintef.no

Abstract

In order to meet the need for flexibility and rapid application development facing COTS software manufacturers, high levels of reuse are necessary. This position paper claims that this is only possible by means of CBSE and outlines the Magma approach to CBSE. This approach is based on domain specific architectures, a 3-tier software engineering process model and object-oriented analysis and design.

Background

Off-the-shelf software vendors experience an increasingly dynamic environment for their products. Both underpinning technology and user demands are in constant movement and challenge the ability of the vendors to evolve their products accordingly.

The Magma project was initiated by the association of the Norwegian software industry (PROFF), to aid their member companies to meet this challenge. To this end a Software Engineering Handbook is being developed. The project is led by PROFF and is carried out as a cooperation between SINTEF Telecom and Informatics and a handful pilot companies. A draft version of the handbook is currently being tested in the pilot companies. The pilot companies are relatively small companies (from a few tens to a few hundreds developers) that are developing and selling software products basically off-the-shelf, although in some cases there is a modest amount of customization involved with each sale.

The Magma Software Engineering Handbook is based on the belief that extensive reuse is necessary to meet the demands for flexibility and rapid response to new needs and new

technology enforced by the marketplace and that CBSE is the only way to bring about the necessary level of reuse.

Position

In the following we briefly outline the Magma approach to CBSE.

Problem domains

Most software systems are there to support real life processes, for instance business processes or production processes. A set of related real life processes fulfilling a given purpose we call a problem domain. Typically the business idea of an off-the-shelf software company is to provide software solutions for a particular problem domain

Domain architectures

We envision the structure of the software system that supports a domain as consisting of applications and components.

Applications are the parts that interact with the supported real life process, and typically support a particular process or a particular type of actor.

Applications use components to implement the end user functionality that they offer.

The literature offers several definitions of what a component is. We sympathize with the following definition by Kruchten [BROWN98]:

A software component is a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

This structure of applications and components and the way they interact constitutes *the domain architecture*.

Normally components are shared in the sense that the services they offer are being used by several applications or other components. Sharing takes place both at the type level (code reuse) and at the instance level (data sharing).

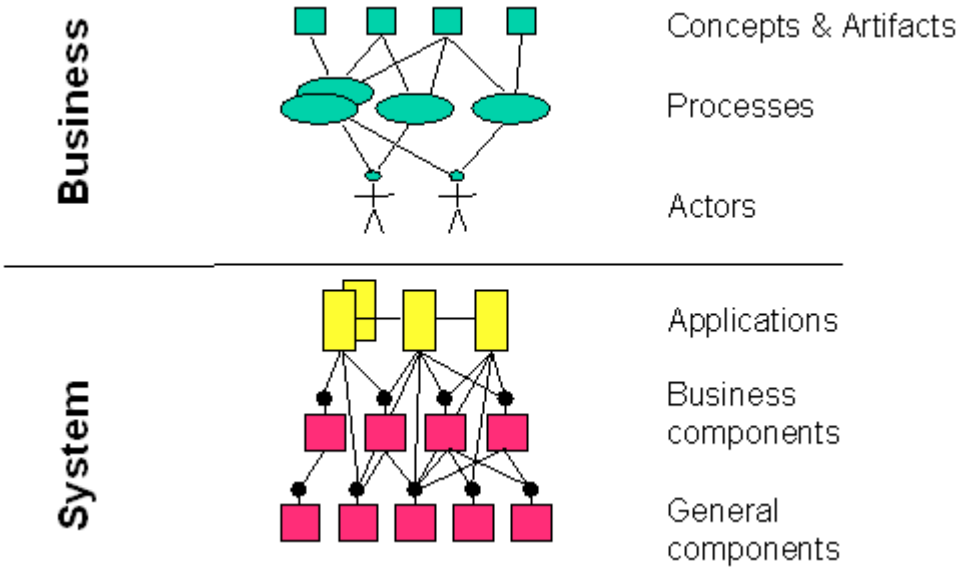


Figure 1 The fundamental structure of a domain and its supporting software systems

We distinguish two main classes of components; *business components* and *general components*. Business components implement concepts of the problem domain. General components implement the infrastructure of computational mechanisms needed by the business components to fulfill their task. This includes components dealing with UI controls, persistent storage, concurrency, transactions, communication etc., and is where component middleware like MS COM, CORBA and JEB fits in.

Object oriented modelling with UML

Modern software systems and the problems they solve are far too complex for any human being to understand as a whole. Therefore we need to build models that allows us to view and contemplate both the problem domain and the software systems at different levels of abstraction and from different points of view. A fundamental idea is that of building a complete picture by synthesis from partial models. This is recommended in various forms throughout the handbook.

We have adopted the object oriented paradigm for modelling, which means that we think of and model the real world as well as software systems as consisting of a set of collaborating objects. We do this primarily because we think it is a more powerful modelling paradigm, and not because we think CBSE is impossible without it. The preferred notation for object oriented models is UML [FOWLER97] [AMIGOS98].

Object oriented components

Our adoption of the object oriented modelling paradigm also has an impact on our view of

components, which we generally conceive as objects, although on a coarser level of granularity the objects typically found in analysis models and in source code. Normally a component encapsulates a cluster of such finer granularity components of the same or of different classes.

Software engineering process architecture

In accordance with the picture drawn above, the Magma process architecture defines three main software engineering processes: domain engineering, component engineering and application engineering This is illustrated in Figure 2. This three-tier process architecture is similar to the one proposed by Jacobson et al. in [JACOBSON97]

Component based systems are built through the concerted effort of such processes, where the domain engineering process provides the domain knowledge and the architectural context for component and application engineering.

Domain Engineering

The domain engineering process is a collection of activities aiming to establish and maintain the domain knowledge and technical infrastructure necessary to effectively develop and maintain competitive software products for a given problem domain.

To this end the domain engineering process produces a set of domain models that transcends transcend individual applications and components and focus on different aspects of the domain.

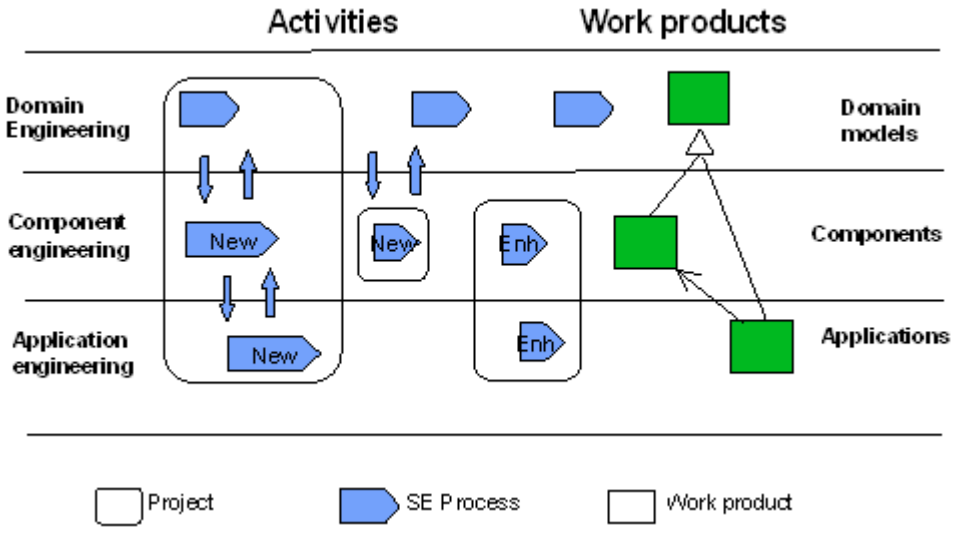


Figure 2 The Magma process architecture

The *business model* focuses on the business as such and seeks to model the processes, actors,

concepts and artifacts of the business and the relation between them. The *needs model* focus on the needs for software support in the domain. The *architecture model* defines a common domain architecture, that is it identifies applications and components and the ways they collaborate to satisfy the needs specified by the needs model. The domain architecture is a key work product as it provides the common architectural context for components that we think is instrumental to CBSE.

Domain engineering is not something you do once and then is done with. Continuous activity is necessary to keep track of the evolution of the domain and keep the models up to date.

An important aspect of the domain engineering activity is to identify and accommodate the anticipated variation in the domain, both the variation caused by different needs of different users or user groups, and the variation caused by evolution.

Component Engineering

The *component engineering* process develops the reusable components from which applications are built. Each instance of this process develops one component. The starting point is the interfaces and responsibilities assigned to the component in the architecture model.

Ideally components generalize over the anticipated variation in requirements as defined by the domain models, and this is a major concern in the implementation of components. This is not always feasible, however, and then one can build a generic component that can be instantiated with different properties depending on the specific requirements of the context in which the components is being used.

In addition to the implemented component, the component engineering process also produces a set of abstract models. The *interface model* defines collaborator roles and collaborations into which the component may engage and object and relation types visible in the interface. The *customization model* defines how the component can be customized to meet varying requirements. The *design model* focuses on the internal design of the component.

Application Engineering

The *application engineering* process develops individual applications, that is the programs that the users see. By definition the components implement the functionality that the applications need to offer to the users. Therefore the main challenge of the application engineering process is to select the specific functionality adequate for the particular type of business process or role the application is meant to support and to design an appropriate user interface.

Development projects

A development project has clear objectives to be achieved in a given time and with given resources. To this end the project instantiates and coordinates software engineering processes as necessary. For instance a project, whose primary objective is to develop an application and thus instantiates an application engineering process, may also need to instantiate domain engineering activities to refine the domain models and component engineering processes to develop not yet

available components. Typically in the early stages of transition to CBSE, where domain models and components have to be established, there will be an overweight of domain and component engineering, while in the more mature stages, relatively mature domain models and component implementations will exist and application engineering will dominate.

An incremental project process model with use case based increments has been adopted, which is similar to RUP [KRUCHTEN99]. This type of project process seems best suited to match the need for flexible development and is also well suited for coordinating the parallel engineering processes.

References

[JACOBSON97] I Jacobson, M Griss, P Jonsson,
Software Reuse – Architecture, Process and Organisation for Business Success
Addison Wesley Longman / ACM Press, Object Technology Series, 1997.

[FOWLER97] Martin Fowler, Kendall Scott: UML Distilled
Addison-Wesley, 1997.

[AMIGOS98] Grady Booch, Jim Rumbaugh, Ivar Jacobson,
Unified Modeling Language User Guide
Addison-Wesley, October 1998.

[BROWN98] Alan W. Brown, Kurt C. Wallnau: The Current State of CBSE.
IEEE Software, September/October 1998.

[KRUCHTEN99] Philippe Kruchten: Rational Unified Process – An Introduction.
Addison-Wesley, 1998

The Role of Formal Methods in Component-Based Software Engineering

P.S.C. Alencar D.D. Cowan

Computer Science Department

University of Waterloo

Waterloo, Ontario N2L 3G1

{palencar,dcowan}@csg.uwaterloo.ca

Abstract.

In this chapter we intend to present a survey recent progress in the development and application of formal techniques for component-based software systems. This chapter may be included in part 4 or part 3 of the proposed book outline. As central issues, we discuss formal methods and their applications to component descriptions (such as COM) and architecture descriptions (such as UNICOM). As a starting point, we discuss formal models for the specification and verification of Module Interconnection Languages (MILs). Topics that we plan to discuss in this chapter include the precise specifications of component-based and architectural models and their properties, automated analysis of the models, and use of design constraints to provide guarantees about, for example, and the system structure and behavior. Finally, we discuss some general promising research directions in this area.

The main idea of this chapter is to describe and emphasize the current and future (potential) benefits of the interaction between formal methods and component-based software engineering. The content of the chapter, although ``formal'', should be presented in a ``user friendly'' way. The chapter may be adapted to fit the negotiated final content of the book; for example, we can restrict ourselves to component descriptions and omit section 5 on the formalization of ADLs.

1. Introduction

Component-based software engineering has been widely accepted to be an engineering discipline. Typically, all engineering disciplines today are based on some theoretical foundations, and these foundations are a basis for understanding the involved engineering tasks, rules, procedures, and processes. Also, component-based software engineering needs its own theoretical foundations like any other engineering discipline.

The goal of the following proposed sections is to describe the current state of the application of formal techniques to component-based software engineering and give some indication about promising related research directions in the area.

2. Formal Methods

In the context of software, the term ``formal methods'' refers to the use of techniques from formal logic and discrete mathematics in the specification, design, and construction of computer software [7,8,9,10]. By using formal or rigorous specifications, much of the ambiguity that is found inevitably in informal specifications can be eliminated. The use of formal specifications allows the software systems to be analyzed in various useful ways: many ways. Formal proofs eliminate the subjectivity and ambiguity of requirements by providing a logical and precise argument of the behavior of the requirements. The use of formal specifications and formal proofs also provide a systematic approach to analysis. Formal specifications and formal analysis can be applied at any life cycle phase, can be supported by computer-based tools, and can complement (formal) testing approaches [11,12].

In this section, we plan to briefly describe some representative formal techniques relevant to this chapter will be presented, such as model-based, state-based, and event-based.

3. Module Interconnection Languages (MILs)

Proposed as a significant step towards programming in a higher level of abstraction, Module Interconnection Languages (MILs) were originally proposed in [1]. Extensions, such as object-oriented MILs, were also later proposed [2].

MILs were formalized in [3]. A formal specification language, Z [15], was used to describe the MIL models. The model allowed the basic MIL components to be instantiated and connected to each other. The general MIL model was instantiated into two well-known MIL variants. Illustrative parts of the MIL models will be shown. See [14] for an alternative formalization. The benefits of these formalizations are discussed.

We also discuss how extended MIL models can be formally analyzed. We describe some work on analyzing the MIL models with respect to their structural properties and their configurations. We describe how reasoning about changes can be performed in this context [4,5].

4. Component Models

In this section we describe applications of formal methods related to representative component models. We plan to describe current efforts to specify general component models, as well formal aspects related to COM (CORBA, and Java BEANS) and formally analyzing these models. Illustrative COM formal specifications and verifications based on the formal models can be found in [20,31,32].

We also briefly describe some of the efforts towards component-based software composition [34-40,6] and other related instances where formal methods were applied [33].

5. Architectural Description Languages

In this section we describe applications of formal methods related to ADLs [13,16-19,21-24,29,30,42,43]. We plan to concentrate not so much on the description of the formal basis for some representative ADLs, but on applications taking advantage of the formal descriptions such as formal analysis and testing. Formal analysis includes verification techniques

(theorem proving and model checking [26,27]).

6. Research Directions

Promising research directions related to the application of formal methods in CBSE are presented. We will discuss some issues about the specification, formal design, refinement, formal analysis and testing (including specification-based testing), maintenance (including adaptation of components), and formal retrieval of components.

References

[1] DeRemer, F., Kron, H., Programming-in-the-Large versus Programming-in-the-Small, IEEE Transactions on Software Engineering, pp. 321-327, June 1976.

[2] Hall, P., Weedon, R., Object-Oriented Module Interconnection Languages, Advances in Software Reuse, Selected papers from the Second International Workshop on Software Reusability, pp. 29-38, March 24-26, Lucca, Italy, 1993.

[3] Rice, M., Seidman, S., A Formal Model for Module Interconnection Languages, IEEE Transactions on Software Engineering, 20, pp. 80-101, January 1994.

[4] Alencar, P. S. C., Lucena, C. J. P., A Logical Framework for Evolving Software Systems, Formal Aspects of Computing, vol. 8, pp. 3-46, 1996.

[5] Lucena, C.J.P., Alencar, P. S. C., A Formal Description of Evolving Software Systems Architectures, Science of Programming vol. 24, pp. 41-61, 1995.

[6] Alencar, P.S.C., Cowan, D.D., CASCON'98 Workshop on Component-based Software Composition, Dec. 3, 1998, Toronto, Ontario, IBM Report, 1999. Also, CASCON'9 Workshop on Patterns and Frameworks, 1997, Toronto, Ontario, IBM Report, 1998.

[7] Craigen, D., Gerhart, S., Ralston, T., An International Survey of Industrial Applications of Formal Methods, Volume I: Purpose, Approach, Analysis and Conclusions, National Institute of Standards and Technology (NIST), Gaithersburg, MD, March 1993.

[8] Craigen, D., Gerhart, S., Ralston, T., An International Survey of Industrial Applications of Formal Methods, Volume II: Case Studies, National Institute of Standards and Technology (NIST), Gaithersburg, MD, March 1993.

[9] Clarke, E.M., Wing, Jeannette, Formal Methods: State of the Art and Future Directions, ACM Computing Surveys 28, 4, pp. 626-643, December 1996.

[10] Rushby, J., Formal Methods and the Certification of Critical Systems, SRI-CSL-93-07, Menlo Park, California, SRI International, November 1993.

[11] National Aeronautics and Space Administration, Formal Methods for Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and

Technology Insertion, Office of Safety and Mission Assurance, Washington, D.C., July 1995.

[12]] National Aeronautics and Space Administration, Formal Methods for Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion, Office of Safety and Mission Assurance, Washington, D.C., July 1995.

[13] Shaw, M., and Garlan, D., Software Architecture: An Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[14] Dean, T.R. and Lamb, D.A., A Theory Model Core for Module Interconnection Languages, Proceedings of CASCON'94 – Integrated Solutions, Toronto, Ontario, Oct. 31-Nov. 3, pp. 1-8, IBM Centre for Advanced Studies, 1994.

[15] Spivey, J.M., The Z Notation: A Reference Manual, ISICS. Prentice-Hall, second edition, 1992.

[16] Clements, P. and Northrop, L., Software Architecture: An Executive Overview, Tech. Rep. CMU/SEI-96-TR-003, Software Engineering Institute, February 1996.

[17] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M. and Zelesnik, G., Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, vol. 21, no. 9, pp. 717-734, Sept. 1995.

[18] Luckham, D.C. and Vera, J., An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, vol. 21, no. 9, pp. 717-734, September 1995.

[19] Garlan, D. and Perry, D.R., Introduction to the Special Issue on Software Architecture}, pp. 269-274, 1995.

[20] Sullivan, K.J., Socha, J., Marchukov, M., Using Formal Methods to Reason about Architectural Standards, Proceedings of the 19th International Conference on Software Engineering, (ICSE'97), pp. 503-513, 1997.

[21]Allen, R.J., Garlan, D., Ivers, J., Formal Modeling and Analysis of the HLA Component Integration Standard, Proceedings of the 6th International Symposium on the Foundations of Software Engineering (FSE-6), November 3-5, 1998 (to appear).

[22] Moriconi, M., Qian, X., Riemenschneider, R., Correct Architecture Refinement, IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 356-372, April 1995.

[23] Allen, R. and Garlan, D., {\it A Formal Basis for Architectural Connection, ACM Transactions on Software Engineering and Methodology, July 1997.

[24] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J., Specifying Distributed Software Architectures, Proceedings of ESEC'95, September 1995.

[25] Zhang, X., A Rigorous Approach to Comparison of Representational Properties of

Object-Oriented Analysis and Design Methods, PhD Thesis, Queen's University, August 1997.

[26] Gordon, M.J.C. and Melham, T.F., Introduction to HOL: A Theorem Proving Environment for Higher Order

Logic}, Cambridge University Press, New York, 1993.

[27] Owre, S., Rushby, J., Shankar, N., PVS: A Prototype Verification Sytem, Proceedings of the 11th International Conference on Automated Deduction (CADE), Lecture Notes in Computer Science, vol. 607, pp. 748-752, Springer-Verlag, June 1992.

[28] Hoare, C.A.R., Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[29] Lichtner, K., Alencar, P.S.C., Cowan, D.D., Using View-Based Models to Formalize Architectural Description, Proceedings of the International Software Architecture Workshop (ISAW'98), FSE'98, ACM SIGSOFT, pp. 97-100, November 1998.

[30] Lichtner, K., Alencar, P.S.C., Cowan, D.D., Formalizing Architecturre Description Languages, Technical Report CS-98-07, Computer Science Department, University of Waterloo, 1998.

[31] Outhred, G., Potter, J., Extending COM's Aggregation Model, Component-Oriented Software Engineering Workshop (COSE'98), in conjunction with Australian Software Engineering Conference (ASWEC'98), to appear, November 1998.

[32] Ibrahim R., Component-Based Systems: A Formal Approach, , Component-Oriented Software Engineering Workshop (COSE'98), in conjunction with Australian Software Engineering Conference (ASWEC'98), to appear, November 1998.

[33] Bumbulis, P., Alencar, P.S.C., Cowan, D.D., Lucena, C.J.P., Validating Properties of Component-Based Graphical User Interfaces Proceedings of the 3rd Eurographics Workshop on Design, Specification, Verification of Interactive Systems, F. Bodart and J. Vanderdonckt (eds.), pp. 347-365, Namur, Belgium, June 5-7, DSV-IS'96, FNRS, Springer-Verlag, 1996.

[34] ATP Focused Program: Component-Based Software, National Institute of Standards and Technology Gaithersburg Maryland, http://www.atp.nist.gov/atp/focus/cbs.htm, http://www.atp.nist.gov/www/press/9706cbs.htm

[35] Nierstrasz O., Tsichritzis D., Object-oriented Software Composition Prentice Hall 1995.

[36] International Workshop on *Large-Scale Software Composition* held in conjunction with *DEXA'98 (Vienna, Austria, Aug. 24-28), 1998, http://www.iro.montreal.ca/~keller/Workshops/DEXA98/index.html.*

[37] CAiSE*98 Ws. on Component-based Inf. Systems Eng. (CBISE'98), Pisa, Italy, 8-9 June 1998, http://www.cs.waikato.ac.nz/~jgrundy/caise98_workshop/.

[38] Intl. Ws. on Component-based Software Engineering, Kyoto, Japan, April 25-26, 1998, http://www.sei.cmu.edu/cbs/icsewkshp.html.

[39] Workshop on Compositional Software Architectures, Monterey, CA, USA, 6-8 January, 1998, http://www.objs.com/workshops/ws9801/.

[40] A. Brown K. Wallnau, "Engineering of Component-Based Systems" in Component-Based Software Engineering, IEEE Computer Society Press, Los Alamitos, CA., 1996.

[41] Medvidovic, N. Rosenblum, D.S., Taylor, R.N., A Type Theory for Software Architectures, Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-CS-98-04, February 1998.

[42] Rosenblum, D.S., Challenges in Exploiting Architectural Models for Software Testing, Proceedings of the NSF/CNR Workshop on the Role of Software Architecture Testing and Analysis, pp. 49-53, July 1998.

[43] Penix, J. and P. Alexander, ‘‘Formal Specifications for Component Retrieval and Reuse,’’ *Hawai’i International Conference on Systems Sciences (HICSS 98)*, June 5, 1997.

[44] Penix, J. and P. Alexander, ‘‘Declarative Specification of Software Architectures,’’ *Automated Software Engineering Conference (ASE 97)*, November 2-5, 1997.

Componentware - Methodology and Process

Klaus Bergner, Andreas Rausch,Marc Sihling, Alexander Vilbig
{bergner|rausch|sihling|vilbig}@in.tum.de

Institut für Informatik
Technische Universität München
D-80290 München
http://www4.informatik.tu-muenchen.de

19th March 1999

Introduction: Componentware Methodology

Componentware is concerned with the development of software systems by using components as the essential building blocks. It is not a revolutionary approach but incorporates successful concepts from established paradigms like object-orientation while trying to overcome some of their deficiencies. Proper encapsulation of common functionality, for example, and intuitive graphical description techniques like class diagrams are keys to the widespread success of an object-oriented software development process. However, the increasing size and complexity of modern software systems leads to huge and complicated conglomerations of classes and objects that are hard to manage and understand. Those systems obviously require a more advanced means of structuring, describing and developing them. Componentware is a possible approach to solve these problems.

An analogy to the building industry illustrates a successful application of such a component-oriented approach: First, the building owner provides the architect with the functional and non-functional requirements in a more or less informal way. Examples are the number and function of rooms and the money he wants to spend. The architect then constructs a first, overall ground plan and several side views or even a computer-generated virtual model of the building. If these proposals meet the owner's expectations, the architect will elaborate a more detailed and technical construction plan. It describes the different components of the building, like walls and windows, and how they fit together. Now the architect invites tenders for these components and evaluates their offers. At last, the ''best'' component producers get the job, place the components to the architect's disposal, and integrate them into the building. During the whole process, the architect's construction plan is the basis of communication between all parties working on the building.

Although there already exist a variety of technical concepts and tools for component-oriented software engineering, the successful model from the building industry was not completely transferred to software development yet. In our opinion, this is partly due to the lack of a suitable componentware methodology. Such a methodology should at least incorporate the following parts:

- A well-defined conceptual framework of componentware is required as a reliable foundation. It consists of a mathematical *formal system model* which is used to unambiguously express the basic definitions and concepts. The contained definitions and concepts should be as simple as possible, yet sufficiently powerful to capture the essential concepts and development techniques of existing technical component approaches.
- Based on the formal system model, *description techniques* for components are required. They correspond to the building plans of architecture and are necessary for communication with the customer and between the developers. Examples for description techniques are graphical notations like class diagrams and state transition graphs from modeling languages like UML as well as textual notations like interface specifications expressed in CORBA IDL, C++, or Java. Well-defined consistency criteria between the different description techniques allow to verify the correctness of different views onto a system with the help of specialized tools.
- Development should be organized according to a *process model* tailored to componentware. This includes in particular the assignment of discernible development tasks to individuals or groups in different roles, for example, a software architect responsible for the overall design of a system, and component developers who produce and sell reusable components.
- The description techniques and the componentware process model should be supported by *tools* . At least, these tools should be able to generate an implementation of the system as well as corresponding documentation. Furthermore, they could facilitate the verification of critical system properties, based on the formal system model.

A more extensive discussion about these fundamental parts of a componentware methodology can be found in [BRSV98b]. In the following sections we focus on the process model for componentware in detail. Such a process model supports system development by clearly defining individual development tasks, roles and results as well as the relationships between them. We first cover the essential aspects that distinguish a component-oriented process model from more traditional approaches. Subsequently, we introduce new development roles associated with componentware and propose a suitable process model for component users and developers. Then we discuss some specific component developer issues. A short conclusion ends the paper, referring to the strawman outline of CBSE'99 [CBS99].

Requirements for a Component-Oriented Process Model

The characteristics of a componentware methodology as described in Section 1 require a suitable process model. Such a process model should itself follow the componentware pardigm: it should consist of a box of building blocks which can be individually tailored to the specific needs of the actual project. There should also be a strong focus on reuse and architectural issues.

New Tasks and Roles

To leverage the technical advantages of componentware and to support the reuse of existing components, the introduction of new tasks accomplished by individuals or groups in new roles is immanent. This pertains to roles like *Component Developer* and

Component Assembler , and to tasks like searching for existing components and evaluating them before their integration into the overall system architecture. The initial elaboration and the continued development of such an architecture requires further tasks like architecture design to be performed by special roles like system architects.

Adaptability and Flexibility

The rigidity of traditional, prescriptive process models is widely felt as a strong drawback, and there is common agreement about the need to adapt the process to the actual needs. A flexible process model should be more modular and adaptable to the current state of the project, much in analogy to the essential properties of components and componentware systems themselves. To provide the necessary flexibility, our approach uses so-called ‘‘Process Patterns’’ (cf. [Cop94,DW98,BRSV98a]).

Combining Top-Down and Bottom-Up Development

With componentware, the successful combination of top-down and bottom-up development is essential. On the one hand, one has to take into account the initial customer requirements, breaking them down into components in a top-down fashion until the level of detail is sufficient for implementation. On the other hand, one has to reach a high reuse rate of existing components. Hence, one starts with existing, reusable components, which are then iteratively combined and composed to higher-level components in a bottom-up fashion. Obviously, neither pure bottom-up nor pure top-down approaches are practical in most cases. New process models, like the Rational Unified Process [Iva99] or the German V-Modell [IAB98] already try to resolve these both aspects by defining an iterative and incremental process.

Evolutionary Approach

The introduction of new roles and tasks is a key aspect of a process model tailored to componentware. However, this doesn’t imply that the process model in question has to be completely new and revolutionary. After all, componentware is itself an evolutionary approach based on the technical foundations of earlier paradigms like object-orientation. Therefore a proposed process model for componentware should represent an adapted and improved version of established practice. In [ABD+99] we have outlined how the German V-Modell standard can be tailored to componentware, focusing on reuse issues and process patterns.

Roles of a Component-Oriented Process Model

The distinction between the roles of *Component Vendors* and *Component Users* is a key aspect of a component-oriented development process. It is a necessary prerequisite for the rise of a market for specialized, reusable components of high quality that are needed to build large, reliable and highly complex systems. Other, more mature industrial branches have known this separation for a long time [Hin97]. We expect the following, specialized roles to evolve in the context of component-oriented software development:

Component Developer:

Components are supplied by specialized component developers or by in-house reuse centers as a part of large enterprises. The responsibilities of a *Component Developer* are to recognize the common requirements of many customers or users and to construct

reusable components accordingly. If a customer requests a particular component, the *Component Developer* offers a tender and sells the component.

Component Assembler:
Usually, complicated components have to be adapted to match their intended usage. The *Component Assembler* adapts and customizes pre-built standard components and integrates them into the system under development.

System Analyst:
As in other methodologies, a *System Analyst* elicits the requirements of the customer. Concerning componentware, he also has to be aware of the characteristics and features of existing systems and business-relevant components.

System Architect:
The *System Architect* develops a construction plan and selects adequate components as well as suitable *Component Developers* and *Component Assemblers* . During the construction of the system, the *System Architect* supervises and reviews the technical aspects and monitors the consistency and quality of the results.

Project Coordinator:
A *Project Coordinator* as an individual is usually only part of very large projects. He supervises the whole development process, especially with respect to its schedule and costs. The *Project Coordinator* is responsible to the customer for meeting the deadline and the cost limit.

Process Model for Componentware

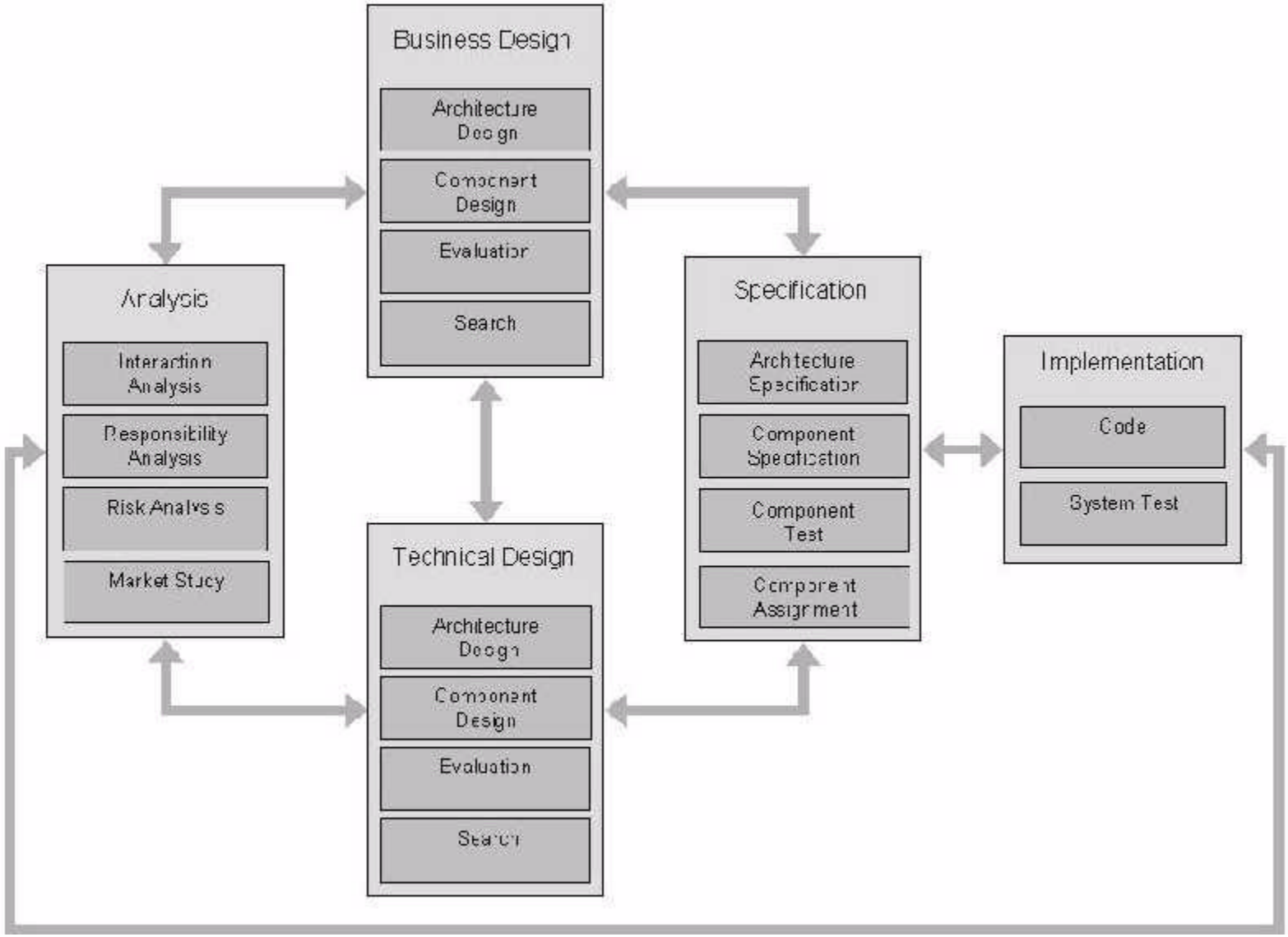
Figure 1 illustrates our proposal for a flexible, component-oriented process model. It shows the different tasks of a componentware development process. Each of these tasks produces results. Thus, a process model for componentware contains a hierarchical *task structure* resp. *result structure* . Note that the presented concepts apply to *Component Users* , i.e. the developers of component-oriented systems, as well as for *Component Vendors* shipping components to *Component Users* .

Figure 1: Component-Oriented Process Model (*figure appears on the next page*)

The main parts are resembling the phases of conventional process models although we explicitly separate business-oriented design from technical design: *Analysis*, *Business Design*, *Technical Design*, *Specification*, and *Implementation*. All main development tasks, like *Business Design*, consist of subtasks like *Architecture Design*, *Component Design*, *Evaluation*, and *Search* each of which is requiring and/or producing development results. For instance, during some task a so-called *Component Design Document* is created which may contain several diagrams using the description techniques mentioned in [BRSV98b] and which is reflected in the result structure shown in Figure 1.

The produced documents and other development artifacts serve as interfaces of the main tasks, analogous to ‘‘real’’ interfaces of software components. The connections between the tasks, namely, the consistency conditions and the flow of structured development information, are visualized by thick, grey arrows in Figure 1.

Note that there are no arrows between the subtasks (resp. subresults) in a main task (resp. result). This is due to the fact that these subtasks are even closer coupled than the main tasks


Business Design
Architecture Design
Component Design
Evaluation
Search
Analysis
Interaction Analysis
Responsibility Analysis
Risk Analysis
Market Study
Specification
Architecture Specification
Component Specification
Component Test
Component Assignment
Implementation
Code
System Test
Technical Design
Architecture Design
Component Design
Evaluation
Search

and are usually developed together. As componentware is based on reusing existing software, it is not plausible, for example, to design the technical architecture (subtask *Architecture Design* of main task *Technical Design*) without searching for and evaluating existing technical components (subtasks *Search* and *Evaluation* of the same main task).

In contrast to traditional process models, we do not define any particular order on the temporal relationship between the development tasks and their results. We believe that a truly flexible process should be adapted to the current state of the project which is partly determined by the current state of the development documents. According to this state, a given development context, and a set of external conditions, we define so-called process patterns which provide guidelines about the next possible development steps. Details about the proposed result structure and the pattern-based approach can be found in [BRSV98a]. In the following, we describe the tasks (resp. results) and involved roles in more detail.

Analysis:

The *Analysis* main result resp. task contains the specification of the customer requirements. The subresult *Interaction Analysis* is concerned with the interaction between the system and its environment. It determines the boundary of the system, the relevant actors (both human and technical systems), and their usage of the system to be developed. Contained may be parts like an overall *Use Case Specification*, a *Business Process Model*, *Interaction Specifications* including *System Test Cases*, and an explorative *GUI Prototype*.

The subresult *Responsibility Analysis* specifies the expected functionality of the system with respect to the functional and non-functional user requirements. It describes the required services and use cases of the system in a declarative way by stating *what* is expected without prescribing *how* this is accomplished. Contained are parts like *Service Specifications*, *Class Diagrams*, and a *Data Dictionary*.

The subresult *Risk Analysis* identifies and assesses the benefits and risks associated with the development of the system under consideration. In the context of componentware, this requires a *Market Study* with information about existing business-oriented solutions, systems, and components.

Note that *Analysis* usually not only covers functional and non-functional requirements, but also technical requirements restricting the technical architecture of the system to be built. While the functional requirements must be fulfilled by the *Business Design* main result, the non-functional and technical requirements must be compliant with the *Technical Design* main result. Furthermore, the *Implementation* must pass the *System Test Cases*.

Business Design:

Business Design defines the overall business-oriented architecture of the resulting system and specifies the employed business components. The subresults *Architecture Design* and *Component Design* are comparable to the *Interaction Analysis* and *Responsibility Analysis* subresults of *Analysis*. However, they do not address technical issues, but instead provide a detailed specification of the business-relevant aspects, interactions, algorithms, and responsibilities of the system and its components. *Search* corresponds to a preselection of potentially suitable business components and standard business architectures that are subject

to a final selection within the *Specification* main result. Within *Evaluation*, the characteristics of the found components and architectures are balanced against the criteria identified in *Architecture Design* and *Component Design*.

Technical Design:

Technical Design comprises the specification of technical components, like database components, for example, and their overall connection architecture which together are suited to fulfill the customer's non-functional requirements. As this result deals with technical aspects of the system like persistence, distribution, and communication schemes, *Technical Design* represents a dedicated part of the development results that should be logically separated from *Business Design*.

In the context of componentware, however, the applied development principles are the same for both areas. Consequently, the involved subresults are analogous to those of *Business Design*.

Specification:

The main results *Business Design* and *Technical Design* are concerned with two fundamentally different views on the developed system. The *Specification* main result merges and refines both views, thereby resulting in complete and consistent *Architecture* and *Component Specifications*.

As said above, both *Business Design* and *Technical Design* cover an evaluation of existing components from the business and technical point of view, resulting in a preselection of potentially suitable components for the system. The *Specification* subresult *Component Test* contains the results and test logs of these components with respect to the user requirements and the chosen system architecture. Note that such tests should be performed as soon as the specification of a component is available in order to avoid problems during system integration.

Some of the desired components may simply be ordered whereas other components are not available at all and must be developed. The *Component Assignment* subresult specifies which components are to be developed in the current project and which components are ordered from external component suppliers or in-house profit centers. If a component is to be developed outside of the current project, a new, separate result structure has to be set up. Note the close correspondence between *Architecture Specification* and *Component Specification* on the one hand, and *Interaction Analysis* and *Responsibility Analysis* on the other hand. It allows for a clear hand-over of a component specification to a component developer outside the project.

Implementation:

The most important subresult of the *Implementation* main result is of course the *Code* of the system under consideration. It comprises source code as well as binary-only components. The other subresult covers the *System Test* results.

Note again that all subtasks mentioned in the above sections may be performed concurrently

and influence each other mutually. For instance, it might be advisable to implement and test critical subsystems early in order to reduce the development risk.

Component Developer Issues

A *Component Developer* implements and ships components to his customers, the component users. These component users may be end-users, i.e. *Project Coordinators* , *System Architects* , and *Component Assemblers* . The corresponding process model for a *Component Developer* is rather similar to the process model for component users, as described in Figure 1:

A *Component Developer* does also receive requirements from his customers, although these requirements are specifications which are usually more formal than the requirements provided by end-users. The *Component Developer* screens his stock for a component which suits the customer's requirements. In some cases an existing component merely has to be adapted during the corresponding design and implementation tasks. The resulting development process is rather fast and the component can soon be delivered.

In other cases, the *Component Developer* has no suitable component in stock, and consequently performs a complete development process as described in the previous section. During *Analysis*, the *Component Developer* should consider customer requirements from a more abstract point of view in order to develop components that may also be (re)used in a different context by different customers. Therefore, a special marketing department should perform an according market study.

During *Business Design* and *Technical Design* the *Component Developer* specifies the component architecture. Possibly, a combination of smaller components within this architecture already fulfills the given requirements. Otherwise, the *Component Developer* has to design and implement the component from scratch. Subsequently, the developed component is tested against the more abstract requirements provided by the market study. Finally, the *Component Developer* will adapt the component, test it against the original requirements provided by the original customer, and deliver it after a successful test.

Conclusion and Further Work

In this paper, we have outlined an overall methodology for componentware consisting of four essential building blocks: a *Formal System Model* , *Description Techniques* , a *Process Model* , and *Tools* . We have presented a modular and adaptable process model suited for componentware based on an overall result resp. task structure and a set of new roles and tasks performing process patterns to fill the result structure.

We think that this work could be part of the CBSE handbook--Chapter 2.1 and 2.2 [CBS99]. We know that we still need to provide further work, especially with respect to refining and elaborating a more detailed result structure. The final result structure will be a combination of the result structures in existing process models, like the RUP [Iva99] or the V-Model [IAB98], tailored to the specific needs of componentware. It will also be elaborated and enhanced with additional aspects, especially with respect to economical and management-related aspects. Based on this result structure, we have to work out the life-cycle

and typical activities in componentware projects. Then we have to elaborate a clear understanding about the different roles and responsibilities in component-based development projects.

Finally, the proposed process model and its accompanying pattern language (cf. [BRSV98a,ABD+99]) are far from being complete--both structure and content of the pattern catalog are not sufficiently elaborated. For the CBSE handbook we have to expand and improve the patterns. Furthermore, we have to present existing process models as process patterns, as we already did in [ABD+99]. Thus, the CBSE handbook will contain a set of process patterns which can be tailored individually to the specific needs of the current project.

=

References

ABD+99

Dirk Ansorge, Klaus Bergner, Bernd Deifel, Nicholas Hawlitzky, Andreas Rausch, Marc Sihling, Veronika Thurner, and Sascha Vogel.
Managing componentware development - software reuse and the v-modell process.
In *Proceedings of CAiSE '99*, Lecture Notes in Computer Science. Springer, 1999.

BRSV98a

Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig.
A componentware development methodology based on process patterns.
In *PLOP'98 Proceedings of the 5th Annual Conference on the Pattern Languaes of Programs*. Robert Allerton Park and Conference Center, 1998.

BRSV98b

Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig.
An integrated view on componentware - concepts, description techniques, and development process.
In Roger Lee, editor, *Software Engineering : Proceedings of the IASTED Conference '98*. ACTA Press, Anaheim, 1998.

CBS99

CBSE'99.
`http://www.sei.cmui.edu//cbs//icse99//strawman.html`, 1999.

Cop94

J. O. Coplien.
A development process generative pattern language.
In *PLoP '94 Conference on Pattern Languages of Programming*, 1994.

DW98

Desmond D'Souza and Allan Wills.
Objects, Components, and Frameworks with UML: The Catalysis Approach.
to appear, `http://www.iconcomp.com/catalysis`, 1998.

Hin97

Dietrich Hinz.

Die neue HOAI.
Forum Verlag Herkert GmbH, Merching, 1997.

IAB98
IABG.
Das V-Modell, `http://www.v-modell.iabg.de/`, 1998.

Iva99
Ivar Jacobson and Grady Booch and James Rumbaugh.
The Unified Software Development Process.
Addison Wesley, 1999.

=

About this document ...

Componentware - Methodology and Process 

This document was generated using the **LaTeX**2`HTML` translator Version 97.1 (release) (July 13th, 1997)

The command line arguments were:
latex2html `position.tex`.

The translation was initiated by Andreas Rausch on 3/22/1999

Andreas Rausch
3/22/1999

The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective

R. A. Riemenschneider and Victoria Stavridou
Dependable System Architectures Group
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Abstract

Most research on architecture description languages (ADLs) has focussed on the use of architecture description to guide conventional, top-down development of systems. This paper describes one approach to bringing the benefits of formal architecture description to the world of component-based development (CBD), where system developement consists primarily of connecting instances of pre-existing components.

1. Using Architecture Descriptions to Achieve Dependability

The principal focus of our group's ADL work has been on describing architecture implementation patterns (i.e., rules for implementing abstract architectural constructs in terms of more concrete constructs), developing techniques for proving that patterns preserve dependability properties of interest (e.g., security, fault-tolerance), and using those patterns to incrementally generate implementation-level descriptions of architectures from abstract, easily analyzable architectural descriptions. For example, we have developed a reference implementation of an architecture for secure distributed transaction processing (SDTP) -- an extension of X/Open's DTP architecture -- and proven it secure by showing that an abstract description of the architecture is secure and that the patterns used to generate the implementation of the architecture preserve security. (See M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong, ''Secure software architectures'',*Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 84-98, and F. Gilham, R. A. Riemenschneider, and V. Stavridou, ''Secure Interoperation of Secure Distributed Databases'', to appear.)

In our view, the scope of architectural description is quite broad. We consider any description of a software system in terms of *components*, joined by *connectors*, that satisfy a collection of *constraints* to be an architectural description. Components correspond to the ''boxes'' of the ubiquitous ''boxes and arrows'' diagrams used for informal architecture description. Connectors correspond to the ''arrows''. What distinguishes architectural descriptions from other sorts of system description is that only the externally visible interfaces of components are described; the

details of the internal functioning of components is left unspecified. The architectural description of the system characterizes the ‘‘glue’’ that binds the components together. An architectural description and descriptions of the components jointly constitute a complete system description.

In contrast to our broad sense of architectural description, some members of the ADL community consider a high level of abstraction to be essential to architectural description. They treat architectural description as the first phase of system design, a basis for more detailed lower level designs. This difference in viewpoint is not merely terminological. The broad view encourages the use of ADLs to describe systems, and parts of systems, in terms of connected components at all levels of abstraction, from highest to lowest. The architectural components in a low level description can naturally be identified with the reusable components of the CBD paradigm, providing a link between ADLs and CBD. Our group has recently focussed on ways of strengthening this link, by adapting our ADL toolset to better support CBD, with a goal of providing technology for building dependable systems from off-the-shelf components.

2. Predictably Dependable Computing with Off-the-Shelf Components

One of the challenges in applying the component-based development (CBD) paradigm to critical software systems is guaranteeing dependability. (A *dependable* system is one that can be justifiably relied upon to provide its services.) If a component is nothing more than executable code, how can a developer gain confidence that it will satisfy his requirements? We are currently investigating one approach to answering this question.

In CBD, the developer’s task is to assemble a (large) software system from (a large number of) reusable components, and perhaps a comparatively small amount of code written for the particular application. In general, the developer does not have access to the source code for the components. We shall assume, however, that each component is accompanied by a *specification*. At a minimum, this specification describes the component’s interface, its externally observable behavior, and any constraints that must be satisfied in order for it to behave as specified. The specification is expressed in a formal notation designed for the purpose.

Such specifications can serve several functions. First, they provide a way for the component developer to advertise the component’s capabilities to potential users. Second, they can support sophisticated component search and retrieval capabilities. (See, e.g., A. M. Zaremski and J. M. Wing, ‘‘Specification matching of software components’’, *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT ’95)*, pp. 6-19.) Third, the behavior of the system can be determined from the specified behavior of the components -- assuming, of course, that the components satisfy their specifications -- together with a description of how the components are connected.

From a practical point of view, a system specification constructed by simply conjoining the specifications of a large number of components is not sufficient for analysis purposes. The complexity of such specifications are far beyond the bounds of intellectual tractability. In addition, tools used to establish dependability properties -- such as theorem provers and model checkers -- are equally incapable of dealing with such complex specifications. Simpler, more abstract, descriptions are required by humans and CASE tools alike.

3. Generating Architecture Descriptions by Abstraction

The key observation in adapting out ADL tools to support CBD was that the same architecture implementation patterns we have been using to generate concrete descriptions from abstract descriptions can equally well be used to generate abstract descriptions from concrete descriptions. Our patterns can be thought of as having the form

- If an abstract architectural description matches template *T*
- And it satisfies the constraints C_1, C_2, ..., C_n,
- Then it can be implemented by the concrete description that matches template *U*,
- Provided that the concrete description satisfies the constraints D_1, D_2, ..., D_m.

(perhaps together with a collection of ‘‘soft’’ constraints that determine when it is advisable to apply the pattern). A pattern can be used for refinement by matching an architecture against *T*, checking C_1, C_2, ..., C_n, generating a more concrete architecture that matches *U*, and adding D_1, D_2, ..., D_m to the constraints that must be maintained on further refinement. But the same pattern can be used for abstraction by matching against *U*, checking D_1, D_2, ..., D_m, generating a match for *T*, and subsequently maintaining C_1, C_2, ..., C_n. Consider, for example, a pattern that says an abstract component can be replaced by an interconnected collection of concrete components, provided the public interface is maintained. This pattern supports the familiar sort of ‘‘bubble decomposition’’ (i.e., horizontal component refinement) common to all refinement-oriented development methodologies. However, it equally supports treating a collection of components as a single component at a higher level of abstraction.

Generation of abstract architectural descriptions from concrete descriptions resembles reverse engineering of systems from source code. In terms of the ‘‘bubble decomposition’’ example, generating the abstraction requires identifying a group of components that can usefully be treated as a single abstract component for purposes of some dependability analysis, much as reverse engineering requires identifying pieces of code that can usefully be treated as modules. The problem of module identification based on analysis of source code is known to be extremely difficult. Is there any reason to hope that architecture-based abstraction will be more tractable? We believe that there is. First, since even low-level architectural descriptions are much simpler than source code, because the internals of the components have already been abstracted away. Second, the behavioral descriptions of the connectors are highly structured. Third, our libraries of architectural implementation patterns are ‘‘domain-specific’’. Our belief is that development organizations that reuse components will tend to reuse them in similar ways, so that abstraction patterns that have proven useful in past analyses can be reapplied in similar ways in present analyses.

We are currently working on a demonstration of the feasibility of this approach to generating abstract architectural descriptions from a description of the system as a collection of concrete components (in the CBD sense) linked by concrete connections (e.g., via CORBA). Abstractions in several different ADLs will be generated, each will be analyzed using the tools provided by the ADL, and the pattern instances linking the abstract descriptions to the concrete implementation will be shown to preserve the results of the analysis (exactly as in the case of

refinement).

4. Conclusions

ADLs have been shown to provide valuable system analysis capabilities, especially when high levels of dependability are required. But the problem of how to make effective use of these capabilities when systems are constructed from reusable components -- especially when development is principally bottom-up, rather than top-down -- has not been adequately addressed. We believe that out approach to generating abstract architectural descriptions from concrete component-based descriptions can bring the benefits of architectural analysis to the world of CBD.

Issues in Component-Based Software Engineering

Kyo C. Kang
Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)
San 31 Hyoja-Dong, Pohang, 790-784, Korea

Introduction

Software reuse is generally considered as one of the most effective ways of increasing productivity and improving quality of software. To make software reuse happen, however, there should be a change in the way we develop software: software must be developed for reuse and with reuse, and there must be a paradigm shift from the notion of specific application "development" to that of "integration." Component-based software engineering (CBSE)[3] is an emerging software engineering paradigm in which applications are developed by integrating existing components. Here, components refer to any units of reuse or integration, including computational (i.e., functional) components, interface components, communication components, and architectures.

In order to maximize the productivity gain and cost reduction, CBSE must be based on domain-oriented components as well as generic components. In general, the productivity increase from the reuse of domain-oriented components is higher than the productivity increase from the reuse of generic components, although reusability of domain-oriented components might be lower than that of generic components. Therefore, CBSE should happen in the context of domain-orientation. SAP R/3 [7], Baan IV [1], and Oracle Applications [6] are good examples of domain-oriented CBSE.

Domain-oriented CBSE is not for any application domains. In immature and unstable domains, there may not be much domain-oriented components to reuse and CBSE may be limited to the reuse of generic components. Therefore, CBSE should be applied in mature and stable application domains, or in an organization where a family of closely related products is produced.

To develop an application by integrating components, there must be components that can solve the problems of the given application. Therefore, CBSE must address not only the issues of how to integrate components but also the issues of how to produce integratable components. CBSE can only be successful when the issues of both producer’s and consumer’s are resolved.

In section 2, a CBSE framework is presented in which major activities of both producing and using components are identified. Section 3 summarizes some of the important component engineering principles. Some of non-technical but important issues underlying CBSE are discussed in section 4. Section 5 concludes this position paper.

CBSE Framework

To develop software by integrating components, components must be developed for reuse. Therefore, CBSE must address both the development of reusable components and the development applications using the reusable components, as shown in Figure 1 [4].

Application-oriented reusable components must be developed for the common needs of the application domain not for a specific application to maximize reusablity. Domain engineering, therefore, consists of activities for identifying commonalities of the applications in the domain (i.e., domain analysis), developing alternative architectures, and developing reusable code components for the architectures. Once domain engineering is performed for a domain and reusable components are created, application engineering may proceed applying the reusable components. User requirements analysis may be performed using the domain commonality model and an architecture appropriate for the application may be selected. Based on the selected architecture, the application software can be created by integrating reusable code components.

Figure 1: CBSE Process (*figure appears on the next page*)

One of the most important elements for successful CBSE is the development of reusable (i.e., implementing common needs/functions, adaptable, maintainable) components. Some of the important component engineering principles are discussed in the following section.

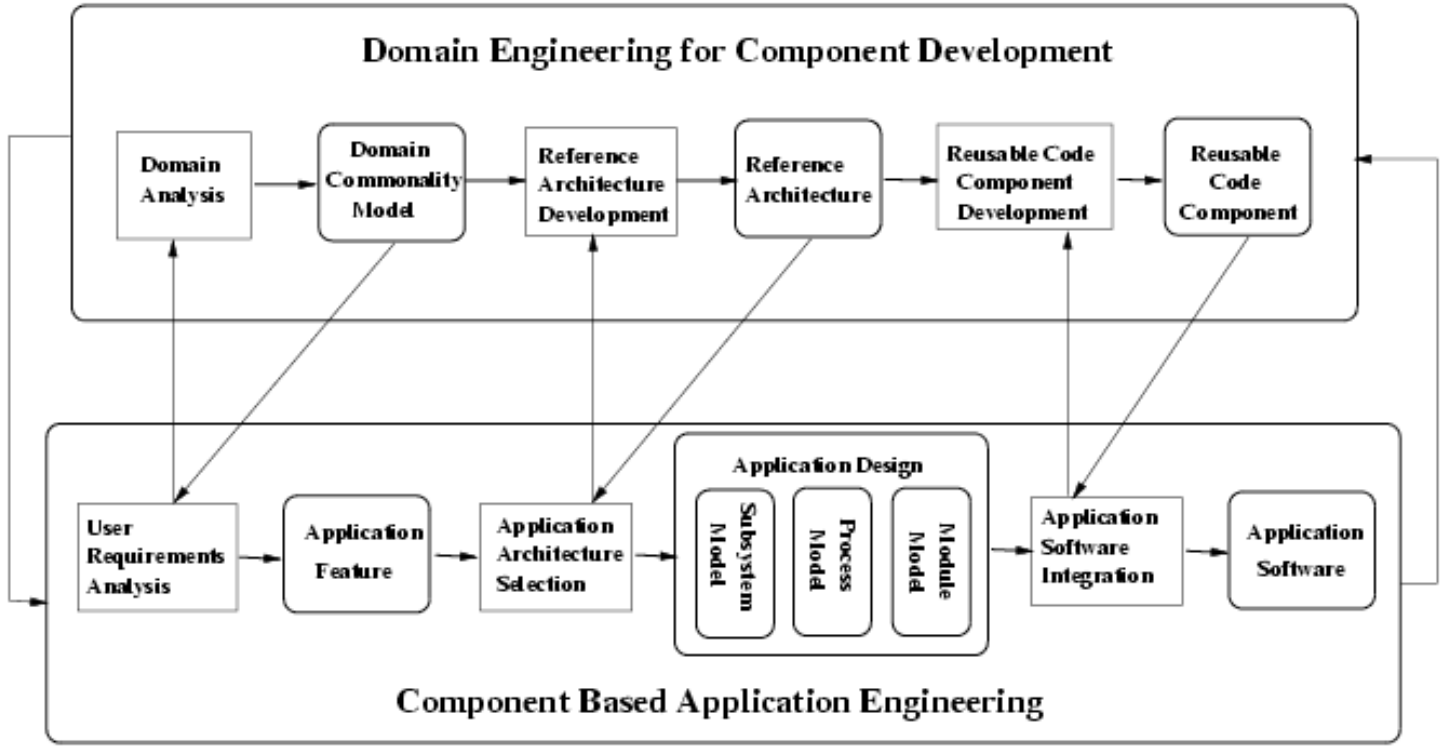
Engineering Principles for CBSE

There are principles for the development of components to support component integration. Some of important component-engineering principles are discussed below.

Domain Orientation: Software reuse may be the most effective ways of increasing productivity and reducing maintenance as well as development cost of software. To achieve successful software reuse, commonalities of related systems must be discovered and represented in a form that can be exploited in developing similar systems. Domain orientation is one such approach. It attempts to discover commonalities of systems in an application or a technical area (i.e., a domain) and then develop models or components that can be used in developing systems in the domain. This approach will help evolving an application as well as developing a family of applications.

Although domain orientation is believed to be the key element in achieving successful CBSE, most domain-oriented engineering technologies are still in their infant stage. There are many technical issues that must be resolved before we can mature these technologies.

Separation of Concerns: This engineering principle is one of the key engineering principles supporting CBSE. Components must be designed so that each performs a unique singular


Domain Engineering for Component Development
Domain Analysis
Domain Commonality Model
Reference Architecture Development
Reference Architecture
Reusable Code Component Development
Reusable Code Component
Application Design
Subsystem Model
Process Model
Module Model
User Requirements Analysis
Application Feature
Application Architecture Selection
Application Software Integration
Application Software
Component Based Application Engineering

function. Also, each functional component must be designed independent of the interface mechanisms it employs to communicate with other components. This principle implies that selection of a particular functional component or an interface method/mechanism does not impose any restriction on selection of other components.

Abstract Virtual Machine Interface: Interface of a component must be designed as a virtual machine. A component must provide a complete, non-redundant (i.e., minimal) interface. The interface must also hide internals (i.e., implementation decisions) of the components so that different implementations can be made for the component.

Postponement of Context Binding: In the design of components, we need to strive for the development of "context free" components focusing only on the core functionality. To the extent it is possible, binding with particular contextual parameters such as data type, storage size, implementation algorithms, communication methods, operating environment, etc. should be postponed until the component integration time when performance optimization is made.

Design Reuse: For component-based software integration to happen, design (i.e., component development context) must be shared and reused among the potential users. That is, design reuse must happen before component reuse. An architectural design shows the allocation of functionality to components and, for a component integration, reuse of the underlying architecture based on which components were developed must occur.

Hierarchically Layered Architecture: Architectures and code components must be designed for maximum flexibility in composing components. Figure 2 includes a layered architecture model [4] which seperates application task-oriented components, which do controlling and activity coordination, from functional components, which do mostly computations. Implementation techniques that are commonly used in the domain are separated from the functional components as implementation techniques can change for the same functions. Data communication models (e.g., message queue, task synchronization) are also separated from the technologies that implement various communication models. This architecture model allows postponement of the binding of particular implementation techniques until the component integration time when performance considerations are made.

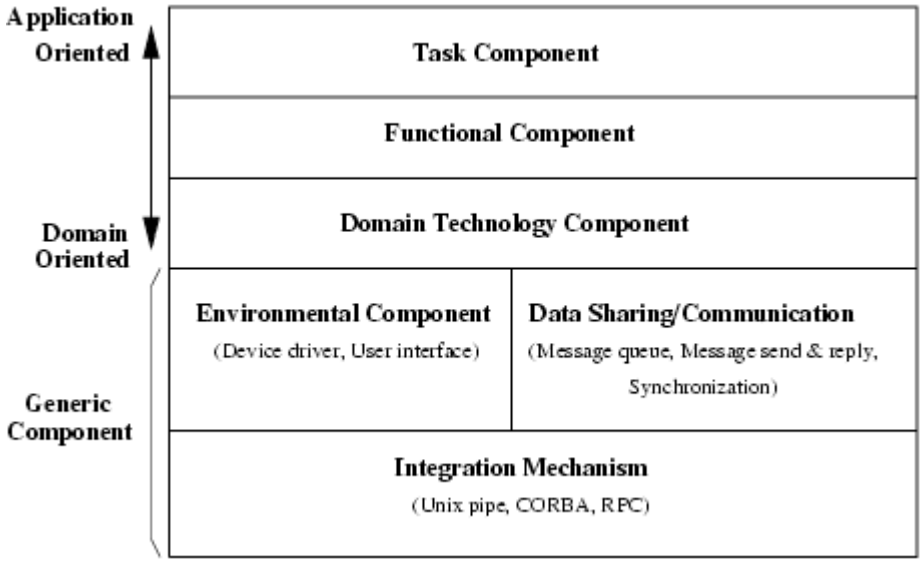


Figure 2: Layered Architecture

Non-technical Issues

One of the most serious problems that impede CBSE is the cost involved in producing reusable components. The cost of producing reusable components is substantially higher than the cost of producing a single application without reuse consideration, as high as five times the cost of producing a single application in some cases [2], few will make this large investment for others or for the future. This approach may be considered economically viable in the commercial software development where frequent customization is needed or in the environment where a family of systems are developed and maintained. It is likely that it will take some time before CBSE becomes the common practice in software engineering. Some of the economics issues are discussed in [5].

Developing software for future reuse or for others is not an easy matter in any corporate environment where software development is managed in terms of projects. It is difficult to see any development projects that were not under schedule pressures or not in shortage of resources. Giving incentives (e.g., award, promotion) to contributors or reusers doesn't usually make much difference unless those incentives are significant. One of the most effective ways to promote CBSE in a corporate environment might be to have a central support organization dedicated to: (1) identify common needs, (2) develop, maintain, and advertise reusable components, and (3) teach and support reuse.

This support organization should consists of highly skilled software engineers who are able to perform domain analyses for the application areas of the development organizations (projects) and develop reusable components applying techniques such as meta-programming, application generator, macro processing, and template. As they can oversee many projects, they should be able to identify common problems among projects and provide generic solutions. The cost of this organization could be amortized through reuse across development projects.

Conclusion

The systematic discovery and exploitation of commonality across related software systems is a fundamental requirement for achieving successful CBSE, and domain orientation is one of the most important elements of CBSE. Domain orientation aims at codifying the development knowledge in an application domain as models and components, and using these models and components in applications development. CBSE might be applied most effectively in mature and stable domains where domain-oriented reusable components can be identified.

For CBSE to become the common practice in software engineering, non-technical as well as technical issues for both producers and consumers of reusable components must be addressed. The development cost of reusable components is especially high and CBSE may not happen unless there is a social/organizational infrastructure supporting the production and exchange of components and amortize the development cost. In a corporate environment, a central support organization dedicated to the development, maintenance, and support of reusable components might serve as a support infrastructure.

References

1 Baan Co., http://www.baan.com
2 Basset, P., Netron Inc., Toronto, Canada (private conversation)
3 Brown, A.W., Editor, Component-Based Software Engineering, IEEE Computer Society, 1996.
4 Kang, K.C., et al., "FORM: A feature-oriented reuse method with domain-specific architectures", Annals of Software Engineering, Vol. 5, pp. 143-168, 1998.
5 Kang, K.C., Levy, L.S., "Software methodology in the harsh light of economics", The Economics or Information Systems and Software, (Veryard, R., editor), pp. 183-203, Butterworth-Heinemann, Ltd, 1991.
6 Oracle Corp., http://www.oracle.com
7 SAP-AG, http://www.sap.com

About this document ...

Issues in Component-Based Software Engineering

This document was generated using the **LaTeX**2HTML translator Version 96.1-h (September 30, 1996) Copyright © 1993, 1994, 1995, 1996, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:
latex2html `cbs.tex`.

The translation was initiated by on Tue Apr 13 16:40:22 KST 1999

Tue Apr 13 16:40:22 KST 1999

On Software Components and Commercial ("COTS") Software

Kurt C. Wallnau

Software Engineering Institute

Pittsburgh, PA, 15206

Abstract: The software industry is struggling to understand the meaning and implications of component-based software. A number of different perspectives have emerged concerning the nature of "software components", but one perspective that is particularly strong is that software components will be (*or are already*) commercial software products. In this paper I argue that if this commercial perspective is valid, then the goal of replaceable, standard components will never be achieved because component providers will resist might and main the emergence of the commodity-style markets implied by component substitutability. However, this strong assertion must be reconciled with manifest evidence that a marketplace of commercial software "components" is emerging. This reconciliation takes the form of a reference model that describes and relates different market niches for software components and engineering skills in an emergent component-based software paradigm.

Motivation

In reviewing the literature for component-based software engineering (CBSE) a.k.a. component-based software (CBS) a.k.a. component-based development (CBD), I have been struck by the recurring theme of *component marketplace*. This is true of submissions to the ICSE'99 workshop on CBSE, at least eight of which assume without comment that components

are purchased "commercial off-the-shelf" (COTS). The correlation of component concepts with marketplace concepts is also a fundamental precept of Szyperski's influential (if not seminal) book on software components [szyperski-98].

Unfortunately, there is an inherent mismatch between one frequently stated key objective of software components, the ability to easily replace components, and the interests of COTS software vendors. Substitutability implies that one vendor's product can be substituted for another, and this, in turn, implies a commodity market of software products. But commodity markets are dominated by price competition, which is not in the interest of software product vendors. Why? Because software production is just as difficult, expensive and risky for producers of software products as it is for their consumers. These and other software production factors are not consistent with the emergence of a commodity market for software components.

The problem is that component substitutability expresses the interests of consumers without taking consideration of the interests of producers. One frequently encounters claims such as how software components will provide investment protection for consumers, allowing a measure of isolation and insulation from a fast changing technology marketplace. Of course, component providers have an interest in protecting *their* investments, and these interests take the form of non-standard and innovative product features. Of course, non-standard and innovative features encourage the opposite of product substitutability--*vendor lock*, the ideal condition for protecting a vendor's investment in their product line.

Consumers appear all too willing to participate in this form of component market. If the reader doubts this, then perhaps a few thought experiments might be convincing. First, is it more reasonable to assume that product selection decisions are made on the basis of the *unique* capabilities provided by the selected product, or on the basis of features that a selected product provides that are also provided in equal measure by all competing products? Second, is it reasonable to assume that the software market is behaving *irrationally* in producing software products that are difficult to integrate or substitute? Is it reasonable to ascribe these difficulties to *technology* deficiencies? The author is comfortable that performance of these experiments will sustain the premise that the component marketplace does not--and will not--result in replaceable software components.

And yet the arguments by Szyperski and others that commercialization of software components is ongoing and inevitable are compelling, and the evidence of this emergence is manifest to any practicing software engineer. It is, therefore, self evident that a thorough understanding of the relationship of software component technology to the component marketplace is essential if we are to fully understand the meaning and implications of CBSE. To this end I propose a reference model for relating software components and CBSE engineering skills to various market niches. This model is intended to complement Jim Ning's excellent reference model that describes and relates various technical aspects of software components [ning-99].

Concept Diagram

The component/marketplace reference model is depicted in Figure 1. The iconography is as

follows. Things that appear in *solid* boxes are components, in the sense that they are *binary units of independent production, acquisition and deployment* [syperski-98]. The hammer-in-circles are engineering skills. Both have *value* in the marketplace that is related to technical aspects of software components. There is some risk in combining skills with components in the same reference model. However, the overall theme of this position paper is that the marketplace has its own internal logic that actively resists the emergence of plug-replaceable components. This necessarily requires the intervention of an external agency, i.e., skilled engineers, to overcome obstacles to plug-replaceability introduced by component vendors. Thus, it is difficult to disentangle components from the engineering skills needed to integrate them.

The positioning of icons suggests relationships between these marketable components and skills; these relationships are made more explicit in the prose description that follows. The description is structured as a glossary of the acronyms used to label the icons. In this description it is assumed that the overall market context is enterprise information systems. This is not an arbitrary choice, but reflects another conjecture (not justified in this position paper) that enterprise-level computing is and will continue to drive software component technology, and that the application of CBSE to other domains is possible but will nevertheless represent a specialty market.

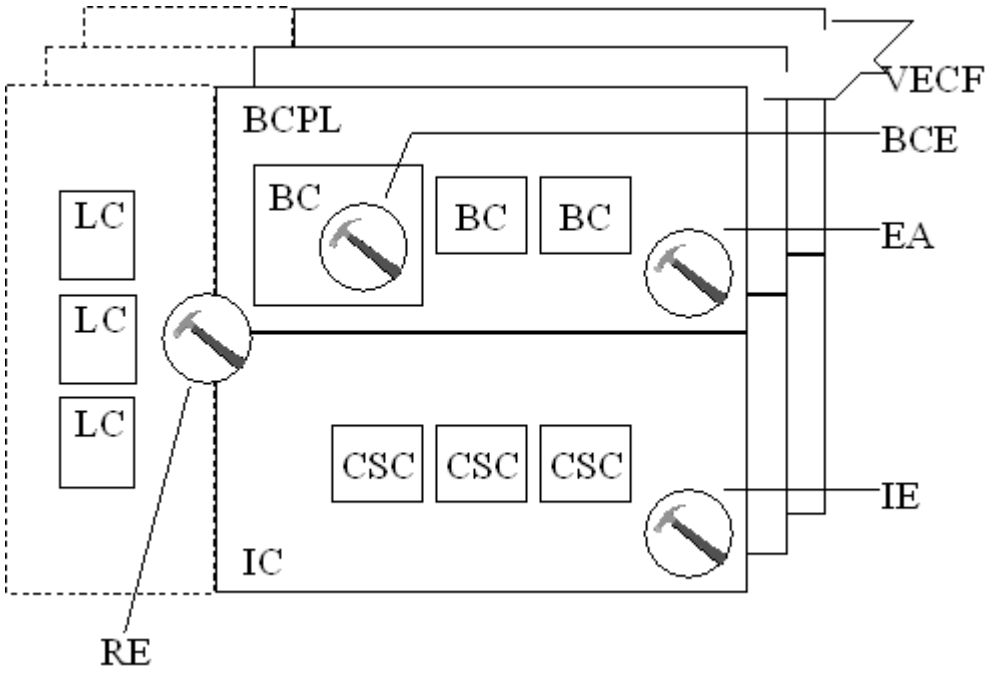


Figure 1: Components and Marketplace

Reference model glossary, organized to facilitate the discussion of related issues:

- **CSC--COTS Software Component**. Modern enterprise systems are composed from these types of components, which include relational database, transaction monitor and other middleware, web/intranet, security and a host of others. This partial extensional definition

may not be theoretically satisfying, but it will satisfy engineers who are experienced in building enterprise systems. This is a robust and unstable market that will likely continue to behave in the future as it has in the past--momentary but only partial convergence on integration standards, followed by periods of technology disruption.

- **IC--Infrastructure Component**. The cost and complexity of integrating CSCs, and the high-level of sophistication required of application programmers who use integrated CSCs, has resulted in a new class of component--the off-the-shelf enterprise architecture and integration framework, generically referred to as IC. A class of *component-friendly* ICs is now emerging, specifically Microsoft's COM+/MTS and Sun et.al.'s Enterprise JavaBeans (EJB). This is an emerging market whose success depends upon evidence that the *switchover* costs from conventional ICs to component-friendly ICs is justified by production efficiencies (e.g., faster time to market, decreased maintenance costs, easier incorporation of innovation). Note that this market will remain fragmented even if EJB succeeds, for EJB vendors are no more inclined to produce plug-replacable *servers* and *containers* than any other vendor of commercial software, and there is evidence to support this assertion [dorda-99]. In the remainder of this paper it is assumed that ICs are component friendly.
- **IE--Infrastructure Engineering**. The skills required to integrate "best of breed" CSCs is daunting. Not only is each CSC complex in its own right (as it must be if it will be "bought" rather than "made") and unstable (as it must if the vendor is to add innovative features to outpace competitors who invariably adopt the successful features of competing products), but their integration into *ensembles* of products is also quite complex [seacord-99]. The engineering skills and capacity for keeping abreast of changing technologies are therefore quite specialized, and very distinct from those skills often associated with the production of business applications.
- **BC--Business Component**. Assuming the emergence of a market for component-friendly ICs, *business* components will be developed. The definition of business component provided by Wojtek Kozaczynski [brown-98] as a "software implementation of an autonomous business process" is sufficient for the purposes of this paper. Actually, it is the author's opinion that there will *never* emerge a marketplace of individual business components because a) business processes tend to be unique, and, b) even if they could be made generic they would not become plug-replacable with other business components for precisely the same market reasons that inhibit plug-replacable CSCs. Thus, a market of individual BCs is highly unlikely.
- **BCPL--Business Components Product Line**. More plausible is the emergence of a *product-line* of business components--at least, there is at least one vendor who thinks this is a plausible scenario [theory-99]. Note that this discussion finesses two issues. First, there is the distinction between a component and configuration of components. Second, there is the distinction between a product line, and the specific product configured from the product line. These are important distinctions, but need not detain us as they do not invalidate the overall thrust of the observation, which is that the success of this market, if it is decoupled from *vertically-integrated components* (see next), depends on whether BCPLs can be deployed into a variety of ICs. This, as noted earlier, is dubious if the IC market continues to fracture, for example if EJB proves to be insufficient as a basis for any reasonable level of portability of Enterprise JavaBeans..
- **VECF--Vertically-Integrated Component Framework.** This corresponds to systems such as Baan, PeopleSoft, OracleFinancial and other well-known enterprise resource

management (ERM) systems. These are "vertically integrated" in the sense that the frameworks package off-the-shelf business processes as well as all the software needed to adapt, maintain, deploy and execute these processes. The current generation of VECFs are decidedly not component-based, and if anything these large-scale systems are aggressive in their pursuit of "vendor lock." However, these vendors are, bit by bit, being forced to make their systems more adaptive to 3rd-party integration (and therefore component-level substitutability) in order to make their products more cost competetive in "mid-level" enterprises.

- **LC--Legacy Component**. Legacy components are existing business applications, whether they are unique to an enterprise (in which case there is no market), or whether they are COTS business applications. The marketplace for these components is not of interest to CBSE except as they introduce *repair engineering* as a necessary and marketable skill. Such repairs are implied by the need for new component-based systems to interoperate with existing systems.
- **RE--Repair Engineering**. Assuming the emergence of a market of component-friendly ICs, a substantial problem remains about how to integrate LCs using component technology. This integration requires the removal of various mismatches between LCs, BCs and component-friendly ICs. There are two classes of mismatches, both of which are non-trivial to diagnose and repair. The first class is *semantic* mismatches at the business process level. Diagnosing and repairing these mismatches requires a thorough understanding of the application domain. The second class is *mechanical* mismatches between the LC and the IC. Diagnosing and repairing these mismatches requires a thorough understanding of operating system level primitives (processes, protocols, etc.) and the IC component model (component life cycles, state transitions and other component-level protocol issues). Engineers skilled in dealing with both kinds of repairs are a rare breed.
- **BCE--Business Component Engineering**. Desinging and implementing BCs involves new skills. In addition to the ability to map an understanding of business processes and concepts to software abstractions (a skill that is needed to build even current-generation enterprise applications), new skills include understanding and management of modular requirements, familiarity with component concepts and infastructures, fault diagnosis in distributed systems executing "black box" components, and the ability to track and accommodate fast changing and frequently non-robust products that underlie the IC.

- **EA--Enterprise Architectecting.** Enterprise architectures enbody large-scale corporate decisions regarding business objectives, and the information technology strategies needed to attain these objectives. The challenges of enterprise architecting have always been substantial, requiring a deft combination of business and technology savvy. A number of additional skills are required if architects are to accommodate the market imperatives implied by component technology, including designing for resiliance in the face of change, maintenance of technology competency, managing uncertainty and loss of control of production factors, and so forth [wallnau-99].

The above discussion is meant to be suggestive rather than exhaustive. It is intended to describe the large-scale implications of market imperatives on software component technology. It also indulges in a bit of prognostication.

Summary

If the premise is valid that the benefits of component technology are derived from a component marketplace, as is eloquently argued by Szyperski and tacitly assumed by others, then the premise that the benefits of software components are derived from their plug-replaceability is surely at risk. This literal dilemma suggests that software researchers need to better understand and relate market factors to the technical concepts underlying component-based software. To this end I have proposed a reference model--doubtless incomplete and flawed--as a starting point for building this understanding.

References

[brown-99] Alan Brown, Kurt Wallnau, "The Current State of CBSE," IEEE Software, September/October 1998, pp.37-46.

[dorda-99] Santiago Comella Dorda, John Robert, Robert Seacord, "Theory and Practice of Enterprise JavaBean™ Portability," SEI Technical Note, in preparation.

[ning-99] Jim Ning, "A Component Model Proposal," in proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE99 http://www.sei.cmu.edu/cbs/icse99/papers/index.html

[seacord-99] Robert Seacord, Kurt Wallnau, John Robert, Santiago Comella Dorda, Scott Hissam, "Custom vs. Off-The-Shelf Architecture," SEI Technical Note, in preparation, submitted to EDOC'99.

[szyperski-98] Clements Szyperski, Component Software Beyond Object Oriented Programming, Addison-Wesley, 1998.

[theory-99] Web site for TheTheoryCenter, www.theorycenter.com.

[wallnau-99] Kurt Wallnau, "COTS Software: Five Key Implications for the System Architect," in Software Tech News, DoD Data and Analysis Center for Software, http://www.dacs.dtic.mil/awareness/newsletters/technews2-3/toc.html