Do we need Component-Oriented Analysis & Design methods ?

Geert Poels[§], *Guido Dedene*

Management Information Systems Group
Dept. Applied Economic Sciences
Katholieke Universiteit Leuven
Naamsestraat 69, B-3000 Leuven, Belgium
{geert.poels, guido.dedene}@econ.kuleuven.ac.be

1. Context and purpose

This is a pre-session position paper of the organisers of Think Tank session *Moving from OOAD towards COAD* (Object Technology 2000 conference, Wednesday 29 March, 14:30 - 15:45). Its purpose is to state, clarify and delimit the problem that is to be discussed during the session and to outline the session's process. The ideas, definitions and assumptions presented in this paper are meant to guide the 'thinking' and to structure the discussion.

The success of the Think Tank depends to a considerable degree upon its problem description. We hope that the conceptual framework sketched in this paper helps you focusing on a common understanding of the problem. The concepts and thoughts presented here are however also meant to provoke your thoughts and reactions. We do not expect you to agree to the assumptions we make. They are the result of our own perception of the problem. In fact, absolutely nothing that is in this paper must be taken for granted, and in principle, everything is open to discussion (we especially welcome questioning the main problem statement itself), as long as the session will not end in chaos.

2. What is the issue ?

We cannot deny the fact that talking about 'components' has become fashionable these days (although not as fashionable as e-commerce). The term 'object' seems to be more and more replaced by 'component'. Is this just fashion, or is there more to it? As an example, in the business objects community, people are now talking about 'business object components' instead of 'business objects'.[1] As another example, Bertrand Meyer notes the name change of the Object Magazine into Component Magazine.[2] But has the concept of a business object, or the contents of Object Magazine changed as well?

From a technological perspective, we cannot turn a blind eye on recent developments. Major organisations (Microsoft, Sun, OMG) have all invested in component technologies (COM, JavaBeans, CORBA), and these technologies are becoming widespread and better known. Technology now allows us to specify a component using a common language (i.e. an interface definition language) that is independent

[♣] Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)(F.W.O.).

[1] J. Sutherland, "The Emergence of a Business Object Component Architecture", OOPSLA'99 Workshop on Business Object Component Design and Implementation, 1999 http://jeffsutherland.com/oopsla99/Sutherland/sutherland.html.

[2] B. Meyer, "A Really Good Idea", IEEE Computer, December 1999, pp. 144-147.

from the language used to implement the component. Component middleware allows distributing components geographically, and amongst platforms. Component architectures allow dividing responsibilities amongst components. Etc…

However, in our perception, methodology hasn't caught up on the move towards component software. Our position is that to build a component software system, technology alone is not enough. We need methodological support as well. There are literally dozens of Object-Oriented Analysis & Design (OOAD) methods,[3] but has anyone heard from Component-Oriented Analysis & Design (COAD) yet? We haven't. We acknowledge that OO has been around for many, many years, and that the component paradigm is still in its infancy (is it?). But isn't it time to take the component paradigm more serious?

But perhaps we are wrong…, and you think there is no need for COAD, or the contemporary OOAD methods are adequate enough, or they are already focused on components, or you just question the usefulness of moving towards component software, …

Perhaps you think we are right. But then, what are the shortcomings of today's OOAD methods (in terms of methodological support for component orientation)? What methodological support would you expect from a COAD method?

Before structuring the many questions related to this Think Tank's issue, let us first make some assumptions regarding 'the world of components'.

3. Conceptual framework and terminology

The obvious trap of this Think Tank is that we all talk about components without understanding each other, because each of us has its own (experience-based or otherwise acquired) reference framework for components. We wish to avoid getting caught in this trap. Hence, this section presents a conceptual framework and terminology for use during the session. It is based on a set of concept definitions, assumptions and perceptions. You may feel that this section overly simplifies the complexities of the 'world of components'. However, we hope it will help focusing on the problem we would like to see addressed.

A. What kind of components do we consider?

There are many flavours of components, as witnessed by the wide variety of definitions that have been proposed.[4] For the purpose of this Think Tank we propose to see a component as an abstract entity, as used during Analysis & Design (A&D). In 'traditional' OOAD methods, the central concept in terms of which we analyse a problem and design a solution to that problem is the object. In a COAD method, the central notion would be the component.

[3] See for instance Cetus links: http://www.rhein-neckar.de/~cetus/oo_ooa_ood_methods.html.
[4] M. Broy *et al*., "What characterizes a (software) component?", Software Concepts & Tools, Vol. 19, No. 1, 1998, pp. 49-56.

This corresponds to the idea of 'component-based software engineering (CBSE) with abstract components'.[5] During A&D, a system is thus modelled in terms of components. Once the application has been designed, it will become necessary to replace these 'abstract' components by executable components, either by implementing the components or by acquiring them (i.e. reuse).

Using the notation of UML, the A&D view of a component can be represented as a <<component subsystem>>: a type of package realising one or more interfaces (Fig. 1).[6] The elements of such a component subsystem are the following:

a. Each lollipop represents an interface, which is a collection of services provided by the component subsystem. There must be at least one interface provided by a component subsystem;
b. The box represents the realisation of the interface(s);
c. Dependency relationships represent the (external) interfaces that are required by the component subsystem in order to realise its own interface(s).

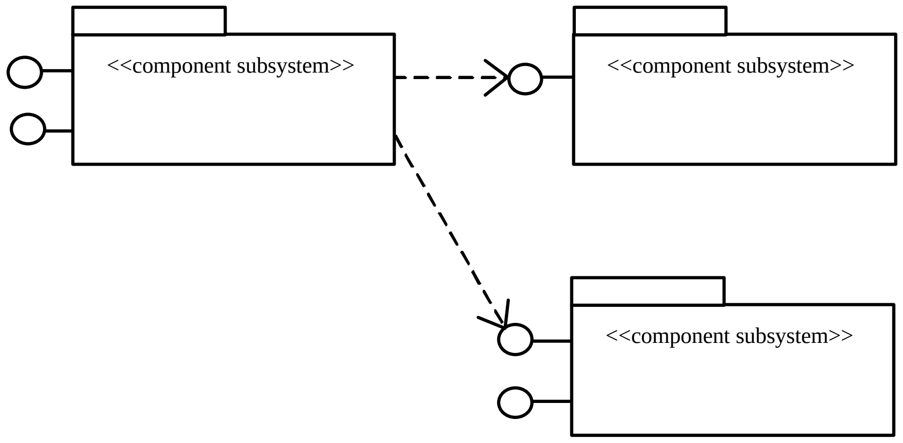

Fig. 1: component subsystems

Important characteristics of a component subsystem are the following:

a. Encapsulation: everything that is needed to realise the interface(s) is within the box, given that the required external interfaces are made available;
b. Information hiding: the functionality of a component subsystem is offered only through its interface(s);
c. Substitutability of realisation: the contents of the box can be replaced by other contents, as long as the same interface(s) are supported and no additional external interfaces are required.

[5] A.W. Brown and K.C. Wallnau, "The Current State of CBSE", IEEE Software, September/October 1998, pp. 37-46.

[6] P. Kruchten, "Modeling Component Systems with the Unified Modeling Language", Proceedings of the 1st International Workshop on CBSE, Kyoto, Japan, April 1998, http://www.sei.cmu.edu/cbs/icse98/papers/index.html.

B. Object-Oriented Analysis & Design and component subsystems

A crucial question now: if a component subsystem is used to represent the A&D view of a component, what are then the interface(s), the realisation of the interface(s), and the dependencies upon external interfaces?

Let us start from (traditional) OOAD and equate for a moment the notion of object with component. The static and dynamic properties of an object are specified in an object type definition. These object type definitions become gradually more refined, i.e. they are transformed into class definitions. It is of course a subtle point where design ends and implementation starts, but let's assume that an OOAD method minimally identifies the domain object types needed in an application, identifies their static interrelationships (e.g. specialisation, aggregation, etc.), identifies their functionality in terms of provided services, specifies the signature of these services, and specifies the interaction protocol between objects, for instance in terms of contracts for the services (i.e. pre-conditions, post-conditions and invariants). We may also assume that some work is done concerning class definitions, for instance the design of methods and the data structures of the class. The interface of an object would then be the set of services it offers, along with their signatures and contractual specifications. The realisation of the interface consists of class methods and data structures. The dependencies upon external interfaces consist basically of all uses of services offered by other objects.

So, in principle, an object in OOAD can be represented as a component subsystem. Class definitions can be 'wrapped' in order to conform to the component technology of choice. Analogously, at the A&D level, objects can be 'wrapped' to conform to the notion of components. It has been argued that this strategy offers advantages as compared to traditional object-orientation.[7,8] Basically, without going into details at this moment, it forces good design practices upon developers, i.e. encapsulation, strict visibility rules and explicit dependencies.

However, to reap the promised benefits of component software (e.g. reusability, maintainability, ease of assembly, etc.), more is needed. Components may be used to introduce a notion of <u>subsystem</u> in OOAD. A component is then essentially seen as a package of objects, specified using type or class definitions, which collaborate to offer a set of services, which are grouped into one or more interfaces.

Working with subsystems in OOAD offers several advantages. First, the A&D models become <u>more manageable</u>:

a. The granularity of the basic building stone in the models increases. The overall effect is that models will contain less building stones and, more important, less relationships between building stones.

[7] G. Wang and H.A. MacLean, "Architectural Components and Object-Oriented Implementations", Proceedings of the 1st International Workshop on CBSE, Kyoto, Japan, April 1998, http://www.sei.cmu.edu/cbs/icse98/papers/index.html.

[8] O.-C. Kwon *et al.*, "Component-Based Development Environment: An Integrated Model of Object-Oriented Techniques and Other Technologies", Proceedings of the 2nd International Workshop on CBSE, Los Angeles, Calif., USA, May 1999, pp. 47-53, http://www.sei.cmu.edu/cbs/icse99/index.html.

b. This effect can be further enhanced by careful selection of the objects that are assigned to a component. Objects that belong together should be put together (i.e. a component should be cohesive), and relationships across component boundaries should be minimised (i.e. reduce coupling).

Second, parts of the models become <u>more reusable</u>:

a. A component subsystem is a package of objects that work together to offer a set of services. If such (or similar) services are needed, then it is likely that the objects would have been reused as a whole anyway.
b. It is hard to reuse individual object type or class definitions if they are heavily coupled to other type or class definitions. A solution must be found for all functionality that a type (or class) inherits or uses from other types (or classes).

So, although working with subsystems (i.e. packages of type or class definitions) at the A&D level seems to offer advantages, Keith Short notices, in an early, highly influential paper on CBSE, that, in general, OOAD methods and tools have no notions with which to describe packages of classes.[9] There do exist notions for grouping classes (e.g. the cluster concept in Eiffel[10]), but they were introduced to better manage the development process, i.e. by dividing development work into packages that can be assigned to separate teams or individuals. Hereafter, we use the term <u>Component-Oriented Analysis & Design</u> (COAD) methods for A&D methods that focus on components positioned at the level of granularity of a subsystem of objects.

Note here that we qualify COAD methods starting from OOAD. The meaningfulness of having a notion of objects in COAD is an implicit assumption we wish to make. It is actually reflected in the title of this Think Tank session. Object-orientation is seen as a necessary step towards component-orientation.

C. Component-Oriented Analysis & Design

The central concept and building stone for modeling in a COAD method is the component subsystem. It represents the A&D view of a component (i.e. an abstract entity), which is positioned at the level of granularity of a subsystem (i.e. a package of objects).

COAD methods (may) play a role in:

a. Application engineering with (abstract) components: a problem is analysed and an application is designed in terms of component subsystems;
b. Component engineering: once component subsystems have been identified and specified, they must be further designed. Within the realisation box of a component subsystem, design work is needed for the collection of objects that realise the component's interface(s);
c. Domain (re-)engineering: even if it is not the intent to build a new application, COAD may play a role in selecting and generalising component subsystems, possibly starting from an already available OO domain model, with the purpose of building a reuse catalogue of component subsystems.

[9] K. Short, "Component Based Development and Object Modeling", White Paper, Version 1.0, Sterling Software, February 1997.

[10] B. Meyer, Object-Oriented Software Construction, 2nd edition, Prentice-Hall, 1997.

Note: Strictly spoken, we could have restricted the use of the term 'component-oriented' to item c (and also partially to item b). The A&D is here oriented towards finding reusable component subsystems. We could then have referred to item a as 'component-based' A&D. The goal is now to analyse and design an application in terms of component subsystems, which can later on be implemented (item b) or acquired (as a result of items b and c, but done previously). Given the assumption made in the previous sub-section, the design work needed with respect to item b could be carried out in the usual object-oriented manner.

Referring to the definition of a component subsystem, a COAD method should minimally specify the interface(s) and dependency relationships. Depending on its role, the realisation part can be further designed or left as a black box.

D. Component-Based Software Engineering methods

Where do CBSE methods like Catalysis[11] fit into the picture? Some of you will argue that Catalysis, or the Catalysis-like methodology underlying the CBSE-tools of Sterling Software[12] and Platinum[13], offers an answer to the problem with traditional OOAD methods sketched above. These methods do indeed support the notion of component subsystems. Moreover, the component subsystem, actually the set of interfaces supported by the component (cf. infra), is specified in an object-oriented manner, allowing these methods to be classified as object-oriented as well. The implementation of the component must not necessarily be object-oriented, although it is strongly advised to do so for newly built components.

However, in our opinion, these methods do not answer all questions. Catalysis(-like) methods focus mainly on domain modeling and interface modeling. They analyse a domain in terms of object types. The type specifications that are considered useful for an application are then further refined into interface specifications. During component modeling, interfaces are assigned to components, which are responsible for realising them. These components must be newly built or can be acquired if available in a component catalog. Sounds really component-oriented, but as far as we know, the methodological support for grouping interfaces, i.e. assigning interfaces to components, is minimal. Hence, we would rather call Catalysis and Catalysis-inspired CBSE methods 'interface-oriented'. The basic building stone used during modeling is the interface, and not the A&D view of a component, which, when represented as a component subsystem, may have several interfaces. Interfaces in Catalysis (-inspired methods) are positioned more or less at the same level of granularity as objects, but not at the same level of granularity as a subsystem.

Perhaps other CBSE methods (e.g., Magma[14], RUP (Rational Unified Process)[15,16,17]) are more 'component-oriented' than the Catalysis group of methods? We do not have

[11] D.F. D'Souza and A.C. Wills, Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.
[12] See http://www.tis.hr/Sterlingsof/Products/spexind.htm.
[13] See http://www.platinum.com/consult/crc/crc.htm.
[14] S. Hallsteinsen and G. Skylstad, "The Magma Approach to CBSE", Proceedings of the 2nd International Workshop on CBSE, Los Angeles, Calif., USA, May 1999, pp. 181-186, http://www.sei.cmu.edu/cbs/icse99/index.html
[15] Rational Software, Object Oriented Analysis and Design, Student Manual, 1998, http://www.rational.com/.

sufficient information regarding these methods at this moment, but perhaps some of you have?

4. What are the questions ?

Given this background, we can now formulate more specific questions related to the problem addressed in this Think Tank session. The questions are listed from general to specific, and they allow for alternative directions in the 'thinking' process. It is not our intent to force your opinions in one direction or another. Even the most fundamental questions are open to discussion.

The main question of course is

Question 1: Do we need Component-Oriented Analysis and Design methods?

Of course, the range of answers to this question will probably depend upon more basic questions, like

Question 2: Do we need components?
Question 3: Given that we wish to use component technology, do we need to bother with the concept of a component at the A&D level?

A "no" to these questions logically implies a "no" to the main question. You might for instance reject the move towards component software altogether, or you might accept this move, but see it as a transformation that can be (quasi automatically) handled on a technological level (e.g. wrapping object classes such that they conform to component standards).

On the other hand, a "yes" does not imply a "yes" answer to the main question. You might think that we need no specific COAD methods, since there are already OOAD methods that are sufficiently 'component-oriented'. You might for instance think so because you reject the positioning of a component at a higher level of granularity than an object. In addition (or alternatively), you might prefer the methods we called 'interface-oriented' in the previous section, because interfaces and not components are the key concept to work with during A&D.

If you do see the need for a COAD method, then we ask you to think about the methodological support you would expect from such a method. The following questions/issues might help you. It must be noted however that they merely reflect the organiser's concerns, and that they are shaped by the assumptions made in the conceptual framework. You are of course not required to stick to the following questions.

---

[16] W. Kozaczynski, "Composite Nature of Component", Proceedings of the 2nd International Workshop on CBSE, Los Angeles, Calif., USA, May 1999, pp. 73-77, http://www.sei.cmu.edu/cbs/icse99/index.html.
[17] P. Kruchten, ibid.

Question 4: Do we allow a component subsystem to provide more than one interface?
If no, why not?
If yes,

a. What good reasons are there for having more than one (provided) interface?
b. On what basis can services be grouped into interfaces?

A "no" answer to the above question trivially solves the problem of assigning interfaces to components (cf. our discussion of interface-oriented methods). However, the answer to the question does not necessarily determine the granularity of the interface or component.

Question 5: One of the key differences between objects and components is the notion of required interfaces.[18] What does 'requiring an interface' mean for a component subsystem? How to decide which interfaces are required given an A&D view of a component, especially in case the component realisation is still a black box?

Question 6: Should component subsystems be types, or just unique instances? Are there services that should minimally be present in each interface (e.g., for creating a new instance of the component subsystem)?

Question 7: What types of static relationships between component subsystems should be supported? What is the effect on the interface(s) of the component subsystems? What is the effect on the object(s) (type definitions, class definitions) in the realisation box (i.e. package)? What is the effect on the dependency relationships (i.e. required interfaces)?

Aggregation/composition (Fig. 2)

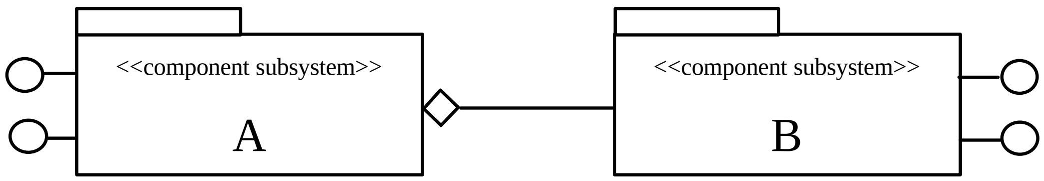

Fig. 2: component A aggregates component B

If not allowed, why?
if allowed,

a. Does A depend on the interfaces of B?
b. Are the interfaces of B a subset of those of A?
c. Is there a difference with the notion of 'containment'?

[18] J.Q. Ning, "CBSE Research at Andersen Consulting", Proceedings of the 1st International Workshop on CBSE, Kyoto, Japan, April 1998, http://www.sei.cmu.edu/cbs/icse98/papers/index.html.

Generalisation/specialisation (Fig. 3)

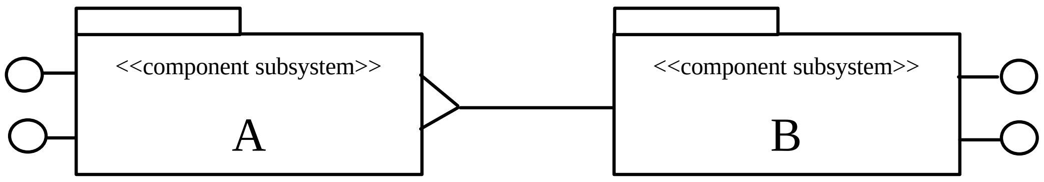

Fig. 3: component A generalises component B

If not allowed, why?
if allowed,

a. Are the interfaces of A a subset of those of B?
b. Do the interfaces of B extend those of A?
c. Does B depend on the interfaces of A?
d. Does the realisation of B inherit from that of A?

Associations (in general) (Fig. 4)

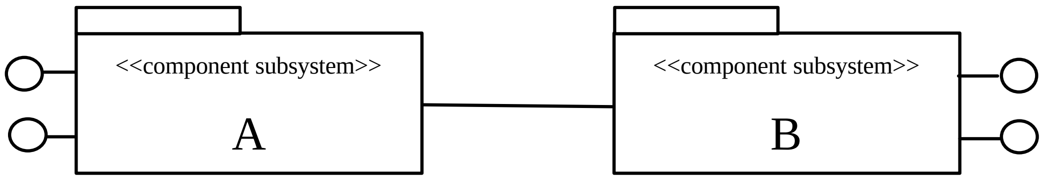

Fig. 4: Association between components A and B

If not allowed, why?
If allowed,

a. Any restrictions on the cardinalities (on either side a choice of 1..1, 0..1, 1..*, 0..*)?
b. Do they imply dependency upon each other’s interfaces?

Question 8: Should a COAD method support the transformation of an object model into a component model? What are the criteria for ‘componentisation'? Is the set of objects partitioned, or are overlaps allowed? What is the effect of relationships between objects (structural relationships, ‘uses’ relationships) that cross component boundaries? (e.g., do they imply relationships between components?, do they imply dependencies on external interfaces?)

Question 9: For a COAD method that models a domain 'from scratch' (i.e., not starting from an available object model), how to find component subsystems? How do concepts like use cases, events, actions fit into this process?

5. Selected literature

1. Software Concepts & Tools, special issue on componentware, vol. 19, no. 1, 1998.
2. IEEE Software, special issue on component-based software engineering, Sept/Oct. 1998.
3. Proceedings of the 1st International Workshop on CBSE, Kyoto, Japan, April 1998, http://www.sei.cmu.edu/icse98/papers/index.html.
4. Proceedings of the 2nd International Workshop on CBSE, Los Angeles, Calif., USA, May 1999, http://www.sei.cmu.edu/icse99/index.html.
5. K. Short, "Component Based Development and Object Modeling", White Paper, Version 1.0, Sterling Software, February 1997.
6. D.F. D'Souza and A.C. Wills, Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.
7. Rational Software, Object Oriented Analysis and Design, Student Manual, 1998, http://www.rational.com/.