# Components are from Mars

M.R.V. Chaudron[1] and E. de Jong[1, 2]

[1]Technische Universiteit Eindhoven, Dept. of Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.r.v.chaudron@tue.nl
[2]Hollandse Signaalapparaten B.V., P.O. Box 42, 7550 GD Hengelo, The Netherlands
edejong@signaal.nl

**Abstract.** We advocate an approach towards the characterisation of components where their qualifications are deduced systematically from a small set of elementary assumptions. Using the characteristics that we find, we discuss some implications for components of real-time and distributed systems. Also we touch upon implications for design-paradigms and some disputed issues about components.

## 1 Introduction

From different perspectives on software engineering, it is considered highly desirable to build flexible systems through the composition of components. However, no method of design exists that is tailored towards this component-oriented style of system development. Before such a method can emerge, we need a clear notion of what components should be. However, although the component-oriented approach can be dated back to the late 1960's (see [McI68]), recent publications list many different opinions about what components should be [Br98], [Sa97], [Sz98]. This abundance of definitions indicates that we do not yet understand what components and component-oriented software engineering are about.

The discussion on what components should be is complicated by the absence of an explicit statement of (and agreement on) the fundamental starting points. As a result, the motivations behind opinions are often unknown, implicit or unclear. Also, presuppositions are implicitly made that are unnecessarily limiting.

The goal of this paper is twofold: firstly, to make explicit the fundamental starting points of component-based engineering, and secondly, to systematically deduce characteristics of the ideal component.

## 2 Basic Component Model and Qualification

First we shall introduce a basic model and discuss its consequences for components in general. Next, we consider some implications for components for real-time and distributed systems.

**Basic component model**

In this section we introduce our basic model for reasoning about components. Our aim is to introduce concepts only when necessary. As a result, a lot of possible aspects of components are intentionally not present in our model.

The model we consider consists of the following:
- There are things called *components*.
- Components may be composed by some *composition mechanism*.

We use the following terminology:
- A configuration of a number of composed components is called a *composition*.
- Everything outside a component is called its *environment*.

A pitfall in reasoning about components is that we presuppose they have features that we are familiar with from programming methodology to such a degree that we cannot imagine that the issues addressed by these features can be approached in another way. Typically, many people endow components with features from the object-oriented paradigm. In order to prevent us from doing so, we will adhere to a strict regime for reasoning about components. We fit our reasoning in the form of a logical theory that has axioms and corollaries. We postulate our basic assumptions about components as axioms. From these axioms we aim to deduce corollaries that qualify components and their composition mechanism.

Next, we present our first axioms.

A1 A component is capable of performing a task in isolation; i.e. without being composed with other components. (1)

A2 Components may be developed independently from each other. (2)

A3 The purpose of composition is to enable cooperation between the constituent components. (3)

Axioms A1 and A2 are generally agreed upon. Already in [Pa72], axiom A2 appears explicitly and A1 is close in spirit to Parnas' observation ".. we are able to cut off the upper levels [of the system] and still have a usable and useful product." The intention of axiom A1 is more explicitly present in recent formulations such as "[a component is an] independent unit of deployment" [Sz98].

To build larger systems out of smaller ones, we want to combine the effects of components. In order to be able to do so, we need a composition mechanism (axiom A3). Note that axiom A3 does not imply that it is a component's purpose to cooperate. In fact, for the functioning of a component it should be immaterial whether it is cooperating with other components (cf. A1). It is the designer (composer) of a

composition who attributes meaning to the combined effect of the components. (Meaning [of a composition] is “in the eye of the composer.”)

Next, we present a first corollary.

C1 A component is capable of acquiring input from its environment and/or of presenting output to its environment. (4)

This corollary can be motivated in two ways. The first is that performing some task (axiom A1) would be futile without some means to observe its effect. The second can be inferred from A3: in order to achieve cooperation between components, there must be some mechanism that facilitates their interaction.

We proceed by deducing some more qualifications of components.

C2 A component should be independent from its environment. (5)

This corollary follows from axiom A1: In order for a component to fulfill its task in isolation, it should have no dependencies on this environment. Put more constructively, a design principle for components is to optimize their autonomy.

C3 The addition or removal of a component should not require modification of other components in the composition. (6)

Corollary C3 follows from C2. Suppose that the opposite of C3 was true; i.e. the addition (or removal) of a component does require modification of other components in the composition. Then, clearly, there is a dependency of the components that require modification on the one that is added to (or removed from) the composition.

Corollary C3 expresses the flexibility or openness generally required of component-based systems.

**Implications for distributed real-time systems**

From the preceding general observations we next shift attention to the design of components for real-time and distributed systems. The corollaries that we present follow straightforwardly from C2. To start with timeliness, C2 leads to the following corollary.

C4 Timeliness of output of a component should be independent from timeliness of input. (7)

Again this is a qualification towards the autonomy of components. One possible means to make the timeliness of output independent of timeliness of input is to build in a mechanism that enables a component to generate output when stimuli do not arrive as anticipated. Typically, such an output can be generated only at the cost of a decrease in the quality of the output.

The next corollary, C5, is the justification of a principle that is known in the area of parallel and distributed systems as *location transparency.* Clearly, C5 follows from corollary C2.

C5 The functioning of a component should be independent of its location in a composition. (8)

Corollary C5 is a constraint on the internals of a component (*internal location transparency).* The counterpart of C5, e*xternal location transparency (*corollary C6) is a qualification of the composition mechanism. Its justification is analogous to that of C3 (by contradiction of the opposite).

C6 The change of location of a component should not require modifications to other components in the composition. (9)

Next, we present our final corrolary of this paper.

C7 A component should be a unit of fault-containment. (10)

The justification of Corollary C7 is as follows: a component cannot assume that some input is normal and some other is faulty, since this implies a dependency on its environment. Hence, a component has to cater for all possible input.

Corollary C7 entails the following guideline for the design of components: components should shield their output from any anomalies[1] at their input.

## 3 On Disputed Issues in Component Design

In this section we will discuss some issues in the design of components based on the qualifications that we found in the preceding sections. When this has unexpected

[1] Actually, the term “anomaly” is indicative for an assumption about, and hence a dependency on, the environment.

implications we may refer to existing composition systems (e.g. pipe-and-filter [Ri80], or shared tuple-spaces [CG89], [FHA99]) to illustrate that there are systems that do not violate these implications.

**Do components have state?**

Let us assume that, in some composition, the task of a component is to store some state. The openness or flexibility corollary C3 asserts that the removal of a component should not require modifications to other components in the composition. This suggests that using a component to store data that is to be used by other components is a bad idea, since this storage component may be removed arbitrarily and the data it stored will no longer be available for other components in the system. In other words, a storage component induces dependencies on other components. This reasoning suggests that stacks and queues should not be considered good examples of components.

Although this is a surprising consequence, we see that neither the pipe-and-filter- nor the shared dataspace model require components that store data. In these cases the composition mechanism deals with the storage of data.

The fundamental issue seems to be that of openness versus encapsulation (in the style of abstract data types as encouraged by the object orientation paradigm). Giving priority to openness (as we do here) seems to prohibit encapsulation of storage.

However, a component is free to build up a "state" as long as the effect of this state cannot be observed by the environment. For example, a filter that performs a word-count on input text clearly computes the output by incrementing some local word-counter. However, this local state does not induce a dependency on other components.

**Are objects components?**

Components are often seen as the next logical step in the evolution of software engineering after objects. Be that as it may, this does not mean that components should be an extension of objects. It may turn out that some features of objects that were introduced to facilitate programming may not be suitable for the purpose of composition.

The following are examples of features of the object-oriented paradigm that seem to hinder composition:

- *The mechanism for cooperation:* The object orientation paradigm uses method invokation (based on message passing) as a mechanism for cooperation. This mechanism requires agreement between the invoking and the invoked object on the order in which methods are executed. Such an order is built into the definition of objects. As a result, addition or removal of an object requires modification to other objects in the system (methods may cease to exist or new methods may need to be introduced), contradicting corollary C3.

In the area of coordination models and languages [PA98], this style of interaction is called *endogenous.* In contrast, in *exogeneous* languages, the interaction between parties is specified outside (textually separate) from the computational code. An example of an exogenous composition language is the pipe-and-filter mechanism from Unix. The specification of the pattern of interaction outside of the components involved in it allows modification of the interaction pattern without requiring modifications to the components.

Also, with method invokation, the initiative for invoking a method may not reside with the object that the method is part of, but with some object in the environment. This is a violation of the independence of components (corollary C2).

- *Encapsulation of data*: One argument is given in the previous subsection as answer to the issue of components and state. Another is given by [HO93]. The essence of that argument is that in an evolving system, the future uses of data cannot be predicted; hence an object that encapsulates data cannot provide the methods for which a need may arise in the future.

The above, however, does not imply that object oriented programming should not be used for implementing components – only that this paradigm does not provide the right abstractions for designing component based systems.

## 4 Concluding Remarks

The fact that currently many different definitions for components are proposed, suggests that we do not yet fully understand the implications of the requirements for component based engineering. In this paper we pursued the implications of these requirements further than is often done. To this end, we presented a rigorous approach to the qualification of components that makes the fundamental assumptions explicit. In this way, we aim to incrementally develop a model for component-based engineering.

Our investigations suggest that object-orientation has some features that hamper the composability of software needed for component-based software development. Hence, we should investigate alternative composition mechanisms.

We welcome comments and additions to our framework.

## Acknowledgements

The authors would like to thank Tim Willemse for his critical comments.

## References


[Br98] Broy M., Deimel A., Henn J., Koskimies K., Plasil F., Pomberger G., Pree W., Szyperski C.: What characterizes a (software) component?, Software Concepts & Tools (vol. 19, no. 1), 1998.

[CG89] Carriero, N. and Gelernter, D., Linda in context, Communications of the ACM, vol 32(4), pp. 444-458, April 1989.

[FHA99] Freeman, E., Hupfer, S. and Arnold, K., JavaSpaces(TM) Principles, Patterns and Practice (The Jini Technology Series), Addison-Wesley, 1999.

[HO93] Harrison, W. and Osher, H., Subject-oriented Programming (a critique of pure objects), in: Proceedings of OOPSLA 1993, pp. 411-428.

[McI68] McIlroy, D., Mass Produced Software Components, in "Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968", P. Naur and B. Randell (eds), Scientific Affairs Division, NATO, Brussels, 1969, 138-155.

[Pa72] Parnas, D.L., On the Criteria to be used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12, Dec. 1972.

[Pa98] Papadopoulos, G.A. and Arbab, F., Coordination Models and Languages. In M. Zelkowitz, editor, Advances in Computers, The Engineering of Large Systems, volume 46. Academic Press, August 1998.

[Ri80] Ritchie, D.M., The Evolution of the Unix Time-sharing System, Proceedings of the Conference on Language Design and Programming Methodology, Sydney, 1979, Lecture Notes in Computer Science 79: Language Design and Programming Methodology, Springer-Verlag, 1980 (also at http://cm.bell-labs.com/cm/cs/who/dmr/hist.html).

[Sa97] Sametinger, J., Software Engineering with Reusable Components, Springer, 1997.

[SG96] Shaw, M. and Garlan, D., Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[Sz98] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.