Component vs. Component: Why We Need More Than One Definition

Bernhard Gröne, Andreas Knöpfel, Peter Tabeling
Hasso–Plattner–Institute for IT–Systems Engineering
E-mail: {groene,knoepfel,tabeling}@hpi.uni-potsdam.de

Abstract

This paper discusses the different meanings of the word component in various contexts of software engineering. To overcome this ambiguity we propose to distinguish between two fundamental meanings of the term which should be emphasized by using different names.

1. Introduction

Developing complex information processing systems is a complex task consuming many resources with regard to man–power, time and money. Much effort is spent in creating the software describing those systems. Object–orientation came with the promise to ease reuse, so classes that were developed for one system could also be used for different systems. If carefully designed, this may work, but often objects and classes—as defined at the level of programming languages—are units much too small for efficient reuse.

Hence, there was a growing demand for higher–level constructs which were called *components*. Following this thought, a new system is developed by mainly combining prebuilt components. The concept appeared promising and everybody had an intuitive understanding of it. Today, many software products and publications about software engineering and architecture promote the concept.

Therefore, Model Based Development (MBD) should support the component–based approach. Yet, there are different meanings of *component* in the context of software engineering that may cause severe misunderstandings and hamper systematic application.

2. The Problem

In its general meaning, *component* denotes a part of something that has been composed. Obviously, a more precise definition is needed in the context of software systems. The popular book “Component software” [10] defines three characteristic properties of a component: It is a unit of independent deployment, it is a unit of third–party composition and has no (externally) observable state. “It is required that the component cannot be distinguished from copies of its own.” Yet, the explanation is contradictory:

- In the definition, a component cannot have an observable state.
- As an example for a component a database server is introduced. Such a server, obviously, offers the service to store and provide data, i.e. to manage some state which is observable by queries.
- Addressing this contradiction, the author refers to the component as the *program* describing the database server. The server is called an instance or the database object, resp.

Following this discussion, components can only be (reusable and prebuilt) pieces of program code.

Architecture description languages (ADLs) are used to describe systems in terms of interacting system components. These components represent the building blocks of the conceptual compositional structure of the system. ROOM is an example for such an ADL [9]. Variants of this approach promote a system view with components and connectors [3].

In this view, components are obviously something different than code, as code alone shows no behavior. A piece of code is some part of a formal system description which is processed by a general purpose machine which becomes the system being described [4].

In other publications, a component can be almost anything. In [1], we read “Do the components consist of processes, programs, or both? [. . .] Is a software component an object? A library? A database? A commercial product? It can be any of these things and more.”

This ambiguity and fuzziness of the term *component* unnecessarily complicates the understanding of many valuable contributions about component–based approaches and hinders progress in this field. If a component is nothing else but a part of a composition, then the fundamental question arises: Which kind of composition?

3. Solution

3.1. Two views—two terms

We should (at least) distinguish two important views: The *system view* and the *software view*. Consequently, in the context of information processing system, at least two types of components should be distinguished:

- System components
- Software components

The essence of this distinction is the difference between a thing and the description of that thing, like between Africa and a book about Africa, or between an information processing system and the software describing this system. (When software is executed by a machine, that machine behaves as described by the program instructions, thus becoming the described system.) Both, the described thing and the description, are inherently different things.

3.2. System components

If we look at a system with its behavior and runtime structures, a component is an active part of the (abstract) system which exists at runtime; a component provides a defined functionality and communicates with other parts of the system. All parts of the system may have their own state.

This system view is typical for Architecture Description Languages (ADLs). Some approaches favor the distinction of different types of system components. For instance [2, 7] describe the (conceptual) structure of a system in terms of components (and connectors). Here, *component* describes a primarily information processing part of the system, while a *connector* has the task to connect components by communicating, filtering, buffering and even coordinating. In a similar way, FMC [5, 12] separates active (agents) and passive system components (channels and storages). Agents process and communicate information, while channels and storages are used to transport or store this information in the system.

3.3. Software components

Component–based development has a different view on components: Here, a component is a deployable software unit which is relevant at build–time (for example a library), or which may be loaded into memory at runtime and be processed by a processor or a virtual machine resp. In any case, it can be treated as a passive artifact [10].

3.4. Why the distinction is important

It is important, because software components and system components do not necessarily map to each other. The structure of the described entity may be very different to the structure of the description. In the context of software systems, this results from the different purposes of the different types of components: System components provide a certain functionality, defined interfaces and protocols for their communication partners. Software components should support deployment, maintenance, configuration and reuse.

In object–oriented software, a class corresponds to a software module, but the functional aspects of a system component may be dispersed on various classes. A detailed description of the various mappings between conceptual system components and objects can be found in [6] and [11]. Furthermore, in the component world more than one instance of a component (’multiple fork of one software component’) is possible. Some software components have no corresponding system component—they just enhance the programming language, as for example a library for mathematical calculations.

3.5. Related approaches

A similar conceptual distinction can be found in other publications. Yet, in our opinion, the criteria of distinction and the terms are less apparent.

With version 2, the Unified Modeling Language [8] distinguishes *components* and *artifacts*. An artifact is “the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system”, i.e. a piece of system description. Components seem to correspond to system components, yet the definition in the current UML2 specification leaves some questions: “Component: a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. [...] As such, a component serves as a type”, but also “a component is a self contained unit that encapsulates the state and behavior of a number of classifiers.” A classifier is a classification of instances, like a class, an interface. In UML, a classifier describes a set of objects. From that point of view, it would be a piece of description. Yet, descriptions encapsulate design decisions, not state or behavior.

Hofmeister, Nord and Soni [3] distinguish different kinds of components, as well: In context of the *conceptual view*, they identify *conceptual components* and *connectors*, while *source* and *deployment components* are introduced in the *code view*. Conceptual components and connectors correspond to system components, while source and deployment components are software components.

4. Conclusion

One single definition of the term *component* does not fit for different contexts which are still close enough to be confused with each other. For each context, i.e. for system and software (as a description), there should be a clear definition of *component* and a term that is concise enough to be understood without any additional definition. The approach of [3] is concise, but requires additional knowledge about the specific meaning of the different views. UML2 components and artifacts appear liable to be misunderstood. For that reason we favor the terms *system components* and *software components* as first–order categories.

Model–based development requires support of various aspects and views. One key to resolve the conflicts between them is a clear separation and definition of the different *component* terms.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] P. C. Clements. A survey of architecture description languages. In *IWSSD ’96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16. IEEE Computer Society, March 1996.

[3] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[4] M. Jackson. The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering*, pages 283–292, April 1995.

[5] F. Keller, P. Tabeling, R. Apfelbacher, B. Gröne, A. Knöpfel, R. Kugel, and O. Schmidt. Improving knowledge transfer at the architectural level: Concepts and notations. In *Proceedings of The 2002 International Conference on Software Engineering Research and Practice*, Juni 2002.

[6] W. Kleis. *Konzepte zur verständlichen Beschreibung objektorientierter Frameworks*. Shaker Verlag, 1999.

[7] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 16(1):70–93, 2000.

[8] OMG. Uml 2.0 superstructure specification, 2003. http://www.omg.org/.

[9] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 2 edition, 1994.

[10] C. Szyperski. *Component Software - Beyond object-oriented programming*. Addison-Wesley, 2 edition, 2002.

[11] P. Tabeling and B. Gröne. Mappings between object-oriented technology and architecture-based models. In *Proceedings of the International Conference on Software Engineering Research and Practice, SERP ’03*, pages 568–574, June 2003.

[12] S. Wendt and F. Keller. Fmc: An approach towards architecture-centric system development. In *Proceedings of 10th IEEE Symposium and Workshops on Engineering of Computer Based Systems, Huntsville Alabama USA (2003)*, pages 173–182, April 2003.