

Toward a unified terminology for component-based development

Stefan Van Baelen¹, David Urting¹, Werner Van Belle², Viviane Jonckers²,
Tom Holvoet¹, Yolande Berbers¹, Karel De Vlaminck¹

¹ K.U.Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Stefan.VanBaelen, David.Urting, Tom.Holvoet, Yolande.Berbers, Karel.DeVlaminck}@cs.kuleuven.ac.be

² Vrije Universiteit Brussel, PROG & SSEL Lab
Pleinlaan 2, B-1050 Brussels, Belgium
{Werner.Van.Belle, vejoncke}@vub.ac.be

Motivation

Component-oriented programming (COP) and component-based development (CBD) have become rather mature software development approaches in the last years, with both quite good conceptual and technological support. In spite of this rapid growth, the concepts used when talking about components are not always well-defined and can lead to misconception, misunderstanding and confusion. Although the general, abstract definition of a component, as defined at WCOP'96 [1]

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 parties.

is widely accepted, a lot of confusion can arise when characteristics of components are described. For instance, a number of people disagree on the fact whether a component should always be stateless[2,3], or whether it could also be stateful[4].

The reason why such discussions arise is that the term *component* is used for different artefacts during several phases of the development cycle, such as a design unit under construction, a commercial off-the-shelf (COTS) software module as well as a real runtime usage, a component deployment. As such, it is clear that a design unit or a COTS module is stateless¹, whereas a runtime usage can indeed be stateful.

At the start of our research project about component-based software engineering for real-time embedded software systems², we tried to clarify the terminology mismatch and provide a profound unified terminology for component-based development. The presented definition will be quite rigorous. We believe this is necessary to avoid misinterpretations and to ensure that everyone is using the same vocabulary.

In fact, we found three dimensions of component-related issues that needed some clarification:

- Component blueprint versus component instance
- Internal (has) versus external visible (gives) component characteristics

¹ The fact that a COTS component can have a version number or that a design component can be in a certain development stage is hereby not considered as stateful.

² The presented material is based on the results obtained from the European ITEA-DESS research project and the Belgian STWW-SEESCOA research project. We especially thank the people of C-LAB (Siemens/University of Paderborn) and DaimlerChrysler for their valuable feedback.

- Mandatory versus recommended versus optional component characteristics

After the detection and precise specification of these dimensions, they were applied on a number of possible component characteristics in order to obtain a clear categorization of such characteristics.

Component blueprint versus component instance

A common project description of a component, comparable with the one that was defined at WCOP'96, was agreed upon and was thought to be quite rigorous and unambiguous.

*A **component** is a documented, logically highly cohesive, lowly coupled software module that can be used as a unit of development, reuse, composition and adaptation. It is therefore an exchangeable architectural element of a software system that acts as a part within a larger whole. It provides and eventually requires dedicated functionality through contractually specified interfaces that can be used in a specific application environment within a specific component system, in order to accomplish higher-level goals.*

However, a lot of confusion arose soon due to the fact that the term component is used for both the component as an abstract *type*, that can be developed or bought, and the component as a runtime deployment. In order to define a clear terminology, we were forced to make a distinction between a component blueprint, as a description of a reusable software element, and a component instance, as a real runtime usage and therefore an instantiation of its blueprint.³ The term component is only used as a more general, encompassing concept, when both aspects are meant or when one does not want to make a distinction between them.

A **component blueprint** is a reusable documented entity that is used as a building block for software systems. It is used to perform a particular function in a specific application environment. Component blueprints are composed using their interfaces. These interfaces consist of provided interfaces and required interfaces. A provided interface describes how the functionality has to be accessed. A required interface describes what is needed to perform this functionality.

A **component instance** is an instantiated component blueprint. This instance behaves as described in the blueprint. More than one component instance can exist at the same time, all based on the same component blueprint. Every component instance needs an implicit or explicit component system to operate in. A component instance can have its own data space and possibly also its own control flow. As such, a component blueprint does not have a state, a component instance can have a state. It doesn't make sense to talk about the runtime properties of a component blueprint, since only its component instances can have runtime properties.

Internal (has) versus external visible (gives) component characteristics

³ This separation can be seen upon as analogous to the class versus object distinction in object-oriented programming.

A second point that had to be clarified are the characteristics a component can have. Based on the internal, white-box viewpoint of a component developer, a component should always have its full documentation associated with it, including e.g. its source code, internal design models and white-box test cases. When one takes an external, black-box viewpoint of the component user on the other hand, the component internals and its source code are not necessary to understand the functionality of the component.

In the case of component blueprints, the outside world (the component user) consists of the component composers. Component composers build applications using the component blueprints. To help them understand the component blueprints, a number of the component characteristics should be shown to the component composers. In the case of component instances, the outside world consists not of the component composer, but of a runtime instance that uses the component instance, like for example another component instance.

This distinction of has/gives-focus on the component viewpoint is one of the main differences between component development, as the development of new components or a system consisting of new components, and component-based development, as the development of a software system using existing components. When looking at the detailed characteristics of components, one now can specify whether a component **has** a specific characteristic and if so, whether this characteristic is **given** to the outside world.

Mandatory versus recommended versus optional component characteristics

The most difficult part of the component definition is to define the required characteristics of a component. Although a number of characteristics are useful or even advisable, they are not always mandatory in order to be accepted as a component. The necessary characteristics for a component are often a methodology-based or even a company-based policy. However, in order to obtain a rigorous definition, we tried to abstract the common component characteristics, and categorised them in 3 groups:

- Characteristics that are absolute mandatory for a component. Notice that, given the distinctions defined above, a component blueprint or a component instance can only be obliged to have a characteristic, or can even be obliged to expose the characteristic to its users.
- Characteristics that are recommended for a component. Although it is highly advisable to provide such characteristic, it is not in all cases necessary to have them in order to be a well-defined component.
- Characteristics that are optional for a component. Dependent of the problem domain of the component, its tasks and usages, or even the component system in which the component is used, a number of optional characteristics can be realized by a certain component. Although in general, more characteristics provide more flexibility, this does not always mean that a component should be built to incorporate as many characteristics as possible.

As such, we tried to categorise which characteristics a component must, should or could have.

Categorization of component characteristics

After the identification of a number of possible component characteristics, we categorized them according to the previous mentioned dimensions. Each characteristic was categorized as being part of the component blueprint or component instance, and for both the internal "*has*" and the external "*gives*" view, we try to define whether the characteristic should be mandatory, recommended or optional. The result of this categorization is the following table.

	Component Blueprint Characteristics	Has	Gives
1	Unique identification	mandatory	mandatory
2	Outside view: <ul style="list-style-type: none"> Provides view Requires view (when not self-contained) 	mandatory mandatory	mandatory mandatory
3	Boundary view: <ul style="list-style-type: none"> Provides interface Requires interface (when not self-contained) 	mandatory mandatory	mandatory mandatory
4	Boundary view: multiple interfaces	recommended	recommended
5	Boundary view levels: <ul style="list-style-type: none"> Syntactic level Semantic level Synchronisation level Quality of Service (QoS) level 	mandatory mandatory recommended recommended	mandatory mandatory recommended recommended
6	Inside view	mandatory	optional
7	Inside view: source code	mandatory	optional
8	Binary form	optional	optional
9	Testing, verification and validation information	mandatory	optional
10	Component management interface	optional	optional
11	Non-functional interface for component fine-tuning and adaptation	optional	optional
12	Negotiation policy for realising its contracts	optional	optional
	Component Instance Characteristics	Has	Gives
13	Unique identification	mandatory	recommended
14	Replaceable at runtime	optional	optional
15	Component Migration	optional	optional
16	Persistent	optional	optional
17	Introspection at runtime	optional	optional
18	Change of non-functional properties at runtime	optional	optional
19	Own control flow (active component instance)	optional	optional

Some clarification to the characteristics:

(1) A name that uniquely identifies a component blueprint. It can be used to specify an unambiguous reference to a component blueprint or as a key attribute for the component blueprint within a component catalogue. It should also be possible to have multiple implementation versions of a component blueprint. Therefore the identification of a component blueprint should consist of 2 distinguishable parts: an **identification name** and a **version number**. It is even possible that a certain version of a component blueprint supports more than one version of its interfaces. Not only the whole component, but also its individual interfaces should be named and versioned

(2, 3, 6) We adapted the terminology that was defined in the Catalysis method [5] to distinguish between the outside view, the interfaces (boundary view) and the inside view of a component.

(4) A component is free to offer more than one interface and it is free to require more than one interface. In fact, it is better to isolate distinct usage roles in to separate interfaces for each role.

(5) We adapted the 4 interface contract levels as defined in [6].

(7) Although components bought from third parties will not always be delivered with their source code, they inevitably must always have source code.

(8) Delivering a component in binary form only means that the component can only be used on platforms that do 'understand' this binary form.

(9) A further division can be made into black-box and white-box test cases. As such, only the black-box test cases can be given to the component users. Furthermore, formal proofs of correctness could also be used as a validation of the component. From a component user standpoint, such kind of information can be very valuable to verify and trust the correct working of a reusable component.

(10) A management interface allows the component composer to make specific functional-related choices for the component. Furthermore, it allows the component to interact with the underlying component system.

(11) A non-functional interface allows the component composer to make a flexible choice between the QoS options that a component supports.

(12) In some situations it is necessary that the user and the supplier negotiate with each other, e.g. to agree on the required quality of service. Negotiation between components is especially useful when components do not know about each other a priori.

(13) Every component instance is unique in the component system. Even when component instances are instantiated from the same component blueprint, they can be made distinguishable. This is indeed necessary when the component instance is able to capture state during its lifetime. Note that this also means that the unique identification of a component blueprint and of a component instance are not the same.

(14) Some component systems support dynamic plug-in components. As such, component instances can be removed, added or replaced at runtime. This feature can help to maintain, adapt, upgrade or fix a running system without stopping it.

(15) Some component instances could migrate from one processor or machine to another. This feature can be used to balance the load in a system. When the component instance captures some state, it is of course necessary to maintain that state of the component instance.

(16) In a running system, a component instance can have a state associated with it. When the component instance has to remain persistent, it is necessary to store this state at certain moments and recreate it at a later time. This can be needed to overcome system restarts, reboots, crashes or power failures.

(17) This allows the user to query a component instance about its services, and find dynamically at runtime the necessary services of that component instance. This can also be supported by the component system or the execution environment.

(18) A QoS choice can be a fixed choice on the level of the component blueprint (one choice for the whole component blueprint), a variable choice at creation time of the component instance, or a dynamic choice at runtime, changeable during the full lifetime of the component instance whenever a certain user asks it to.

(19) An active component is a component which independently initiates communication with other components, calling other component supplier instances to fulfil some tasks. A passive component must be triggered from outside in order to act upon other supplier component instances.

Conclusion

In order to clarify the terminology mismatch and provide a profound unified terminology for component-based development, we identified three dimensions in which the component characteristics can be categorized:

- Component blueprint versus component instance
- Internal (has) versus external visible (gives) component characteristics
- Mandatory versus recommended versus optional component characteristics

As a result, a number of component characteristics were classified belonging to the component blueprint or the component instance that, at its part must, should or could have these characteristics and/or expose them to its users.

References

- [1] C. Szyperski and C. Pfister, workshop on Component-Oriented Programming, Summary, M. Mühlhäuser (ed.) *Special Issues in Object Oriented Programming - ECOOP 96 Workshop Reader*, dpunkt Verlag, Heidelberg, Germany, 1997.
- [2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, Addison Wesley, Harlow, England, 1998
- [3] J. Sametinger, *Software Engineering with Reusable Components*, Springer, Berlin, Germany, 1997
- [4] P. Wegner, Towards component-based software technology, Technical Report CS-93-11, Brown University, 1993
- [5] D. F. D'Souza and A. C. Wills, *Objects, Components and Frameworks with UML: The Catalyst Approach*, Addison-Wesley, 1998
- [6] A. Beugnard, J-M. Jezequel, N. Plouzeau, and D. Watkins, Making Components Contract Aware, *IEEE Computer* 32(7), July 1999