

# Software Components

Author: Verdaguer Olivella, Joan

## Table of Contents

Abstract .....	3
Introduction.....	3
Software components.....	4
Component-based software development .....	5
Attributes of software components .....	7
White box and Black box classification .....	9
Costs.....	10
Services as components .....	12
Conclusions.....	13
References.....	14

## Abstract

The concept of building software from components is not a very new idea, but it seems that even though it can bring advantages in the software development, it is not being used that much among software developers. The purpose of this paper is to give a general overview on the software components and the component-based development software, explaining its characteristics, differences and advantages from the object-oriented paradigm and how can it help in reducing the amount of resources needed in software development. This paper also goes in deep in the component classification with the white-box and black-box models, also discussing its differences, advantages and disadvantages.

## Introduction

The term of component comes from far ago, when the industry began to use interchangeable parts to make its production easier and faster. The main idea there was that a product could be composed of different pieces, so that pieces could be produced independently and the product could be made out of assembling the pieces altogether afterwards. There were actually several reasons for doing that, and among them we find that therefore a company or a factory could then specialize itself into producing just one of the components the product would be assembled with, and distributing those components into all the factories or companies that required them. Also, a product could therefore be designed as a composition of components instead of having to design each piece, so that one component could be used in several products.

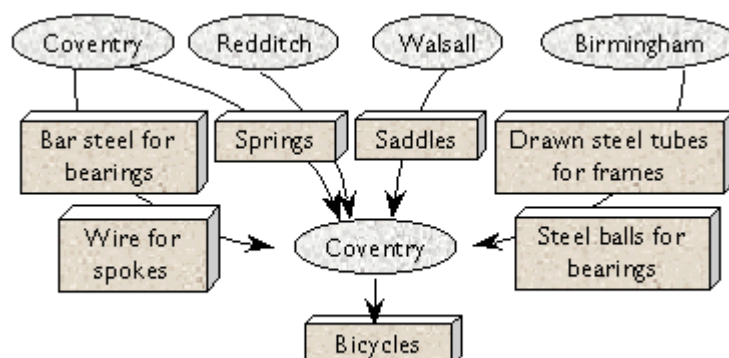


Figure 1, extracted from [10]

In Figure 1 we can see how a componentization of a bicycle can be made, producing different pieces in different cities and putting them altogether in another place.

As the concept of componentization had a great impact in the industry, that idea was also applied in the software production companies. Then, the idea of software components was to be able to make parts of software with a similar behavior than the components in the industry production had. This is, being able to develop a software component in some company and mixing these components somewhere else to make a software solution. That way, as in the industry components, the solution developers could let another company or department specialize in some kind components (e.g. some company could specialize in graphic components) and therefore develop it more effectively.

But obviously the componentization in software and the componentization for industry products have lots of differences since the software has some characteristics that the physical components don't have and vice-versa (e.g. software is easy to transport and copy), making the idea of componentization of these two things very different.

## Software components

Although everyone has in mind a similar idea of what a software component is, there is no definition of them which we could accept as correct or standard because the definition the people give always differs between them in small details. Even though, we could agree in some points that every software component should cover.

*“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”*,

*Definition extracted from [9]*

In the definition above, we can see how a software component is described as a “unit of composition”, meaning that it is meant for software developers to get it and compose it with

other components to effectively make a software solution. Another interesting part of this definition, and where lots of other definitions surely agree, is “with contractually specified interfaces”, id est, a software component should be viewed as a single element of software when developing a component.

Therefore, a software developer should only look at its behavior to use it in his/her solution. Thus, the software component should have a well-defined contract, indicating its behavior depending on its input and its context, to let the developers use the component without ambiguities. Basically, this aims for some kind of encapsulation. Finally, this definition also stands for saying that a software component should not be tied to only one software solution, it should rather be possible to get that component as an independent piece and use it in another solution somewhere else.

### **Component-based software development**

The development paradigm attached to the software componentization is the “Component-based software development”. Its idea is that software can be developed in a new way that is, instead of designing and implementing all parts of a solution, to design a solution based on mixing software components altogether. Id est, gluing components altogether, each of them representing some functionality the system must perform in order to meet the requirements, and just adding some code to make work altogether properly.

The main advantages one can acquire using this software development paradigm are the following:

- *Reduction of development time*: The time required in the development of a solution can be highly decreased if we just get some components that have been already developed and include them in our work instead of developing them from scratch. This can reduce a lot the time to market of our products.
- *Reusability of code*: If our organization has already developed some functionality that we need in some other solution, taking a little more of time to turn this functionality into a component might be useful to us in the new solutions that require it, thus we'll only have to develop it once.
- *Possible reduction of costs*: While developing a solution componentization can help us in reducing our costs if the resources that we have to use to acquire and put the

components in our solution are fewer than the ones we would need to design and implement the code corresponding to these functionalities by ourselves.

- *Reduction of maintenance work:* If a component have been implemented in several solutions and some vulnerability is found in that component, that vulnerability might affect all the solutions. However, we could simply update that component and implement the new version of it in all the solutions that are somehow exposed to that vulnerability. Thus, we might focus to reduce the vulnerabilities just in one component and that will improve the security of all solutions where that component is used.
- *Specialization:* Another great advantage that componentization brings, as mentioned above, is that some entity (either a company, organization or a department) can focus on developing certain kind of components, and this can be very useful when dealing with components that have some functionality that is very specialized. For example components regarding some artificial intelligence or some complex statistic calculations.

We must also mention that there are big differences between the *Component-based development paradigm* and the *Object-oriented development paradigm*, which is the most used paradigm in nowadays development. The *Object-oriented paradigm* has as a goal make a software system out of objects that can fairly represent the real ideas those are based on.

On the other hand, the *Component-based paradigm* wants to group the software in useful functionalities that make sense by themselves and that can be required by other software solutions. And of course actual development can be done by including both a part of component-based development and a part of another development paradigm (this might be, for instance, developing a software solution where only few functionalities are performed by components while the rest of the solution has been developed in the object-oriented way).

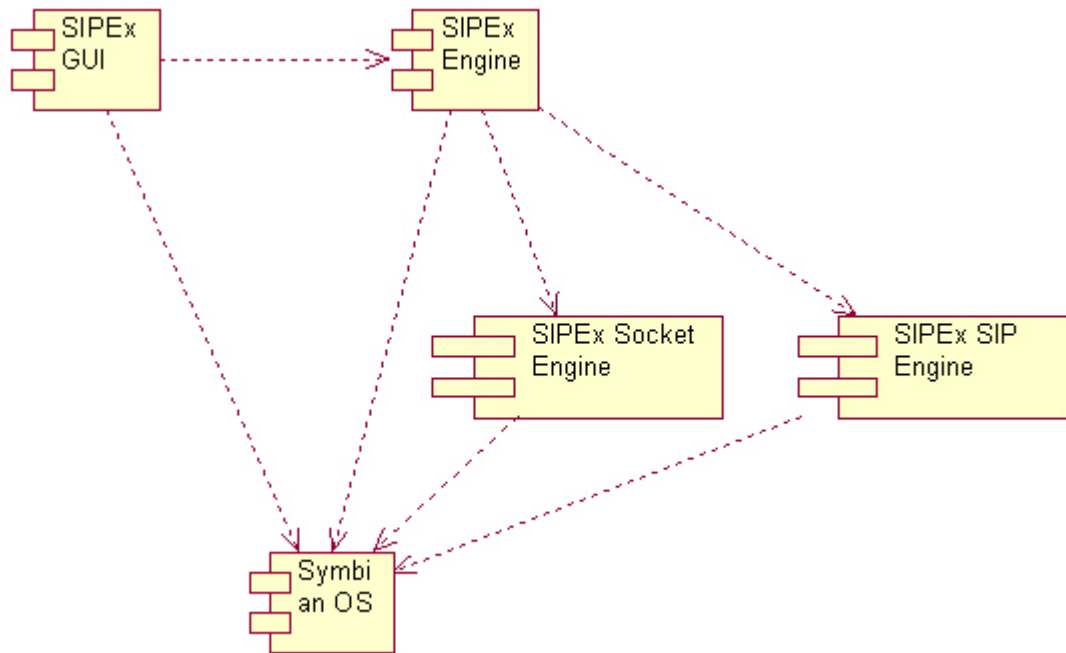


Figure 2, extracted from [11]

In figure 2 we can see a schematic diagram of a software application made out of components. The diagram has been made with the standardized UML. As it is a component-based diagram we can see that it doesn't go in deep into how does each component work nor the objects it is compounded of.

### Attributes of software components

When looking for a software component to use in the development of any of our solutions, there are some points that we should take into account before definitely choosing that component. This is because even if some component fits perfectly in the system we are developing, it might be possible that some of its characteristics can conflict with the way we want to develop the solution.

*Coupling:* Coupling means how dependant is one component upon other components or libraries. For instance a highly coupled component will require us to include other components in our solution, making us bear with the maintenance of those other components, etc. In the component-based development paradigm, just as in the object-oriented paradigm, developers always aim for the less coupling possible, even though sometimes that's not possible.

*Infrastructure:* The idea of infrastructure is the environment (mainly a set of system processes, variables and services) that one component needs to have available in order to be executed properly. For instance some part of the infrastructure of a component might be an interpreter (like Java, python, perl interpreters) or some specific operating system (in case the component is already compiled). That attribute is important when choosing a software component since we must know what kind of environment we will have available in the deployment of our solution. We must also take into account that it's easier to make out a solution with components that require the same infrastructure.

*Granularity:* The granularity of a component indicates how much functionality is in it. With this we can classify each component as a “fine” grained component, meaning that it has a small amount of functionality, or “coarse” grained component, meaning the opposite (obviously there can be gradients in this definition). The fact is that with fine-grained components our solutions will have more components with less functionality on each one, and with coarse-grained components they will be the other way around. The advantage of using fine-grained components is that if we want to modify the behavior of our solution we might just change one small component and let the others intact, while if the functionality we want to modify is in a coarse-grained component then we will either have to change or modify the whole component, even if the functionality of that component is much bigger than what we actually need to change.

*Licenses:* Another important attribute we need to take into account when acquiring external components are the licenses those components are attached to. If we purchase a component from the market that has a copyright it will be most likely that we will not be able to modify it, but we might be able to use it without many other restrictions. But many components can be found that have no copyright, and can be used freely. Many of them have some kind of open license like BSD, LGPL, etc. We have to be aware of those licenses and what are their restrictions when using them in our solutions. BSD license, for instance, allows us to get a component and use it in our solution and all we have to do is mention somewhere in our product that we have used that component, saying who the authors are and what's the license of that component. Another very common open license is GPL (standing for General Public License), in that case, if some of our components have this kind of license, we will be forced, not only to make acknowledgement of the original authors of it, but we will also have to publish a human-readable source code of our solution under the same terms of the component that had the GPL license. A variation of GPL is LPGL (standing for Lesser General Public

License), and looks much more suitable for software components rather than for standalone applications. It mainly has the same terms than GPL license, but a solution that uses a LGPL component doesn't necessarily have to be released under the same license (only should have to if there have been modifications on the code of the LGPL component).

## **White box and Black box classification**

A widely discussed classification in the software components field is the one of classifying the components as black-boxes or white-boxes.

White-box components are those kind of components that can be modified (i. e. its source code is know and changes on it are allowed) by the developers who are using this component and integrating it into one solution. Usually these kind of components have been made in the same company or organization where they are used, or have been acquired with license to make modifications on it (e.g. BSD, LGPL, etc.). The reason for this is that when we are dealing with a software solution we might find that there is a component already used in some other solution that doesn't fit the same requirements than in the new solution but with a small amount of modifications that component could fit perfectly in our solution. So basically we modify some part of that component to make it fit the requirements of our solution.

This approach to the components gives us some advantages and some disadvantages. The main advantage is, as we have already seen, that we can easily reuse other components by just introducing slight modifications in them (thus, the reuse rate might increase), while otherwise we could spend quite an amount of time and resources looking for a proper component for our solution, or even don't find any one that really fits our needs or that can be useful to us.

On the other hand we have the black-box components. As opposite to the white-box components these ones are not meant to be modified when they are used in developing some solution. So the only way to modify its behavior is by modifying the parameters it is called with. The most common reason for these kind of components is that they have been acquired from the market with a license that doesn't allow that, but sometimes, one organization or company might decide to keep some component as a black-box component even though the can effectively access the source code and modify it because of the advantages it might take.

The main advantages of the black-box componentization approach reside in that all the code is the same in all solutions, therefore if there is an update or modification in some part of

the code to solve some vulnerability or make some improvement to the component, this modification can be easily applied to all the solutions where this component is present without much effort. Thus, this is an advantage towards the white-box componentization model since if we want to modify some part of a component there, we will have to look at each solution that has that component, check its modifications and then apply the changes in that component without altering the modified behavior that we gave to it. The main disadvantage of the black-box components respect to the white-box components is obviously the lack of personalization a white-box component can offer. For this, if we have to use a black-box component for our needs but it doesn't exactly fit in our requirements we will then have to do some kind of workaround in the code of our solution to adapt the solution to the black-box component we want to use.

A lot of factors may influence in the decision of whether choosing a black-box or a white-box component for our solution. First, a company should look at the in-house components that have already been developed and if any of those suits the requirements then choose between adapting it to the new solution (this is, turning it into a white-box component) or keeping it as it is and just include it in the solution without further modifications. This decision might be motivated for the amount of changes needed to include the component into the solution, if it is expected to have lots of further modifications, etc. If the company doesn't have any suitable component already, then it should look for another component in the market. If we acquire a component from the market, it is most likely that we will have to follow the restrictions of the seller, which in many case may not give us the source code or the right to modify it.

The white-box and black-box componentization approaches are widely accepted and very realistic, yet some references only consider a software component as a black-box element. But this might be because we use software components as black-boxes when we are in the design phase (id est, in the design phase we never represent the inside of a component even if it is modeled as a white-box one, we just specify the inputs and outputs of it), but we can still model a component as a black-box when designing a solution and then use a modification of another component to make it fit the design (it est, use a white-box component).

## Costs

The costs of using one software component in one of our solutions can be clearly separated in two parts. First the *Acquisition cost*, which stands for all the resources we will have to

spend to get a software component. Thus, this cost depends on two things, first the amount of resources we have to dedicate to look for the component that best fits in our solution. This might include to search for a component, contact companies specialized in some kind of component development, etc, check the contract of the component specification and make sure that all the attributes of that component fit properly in our requirements.

Secondly, we have the *Customization cost*, which is the amount of resources we'll have to dedicate to adapt our solution to that component (in case of a white-box component this does also include the resources used to modify the component).

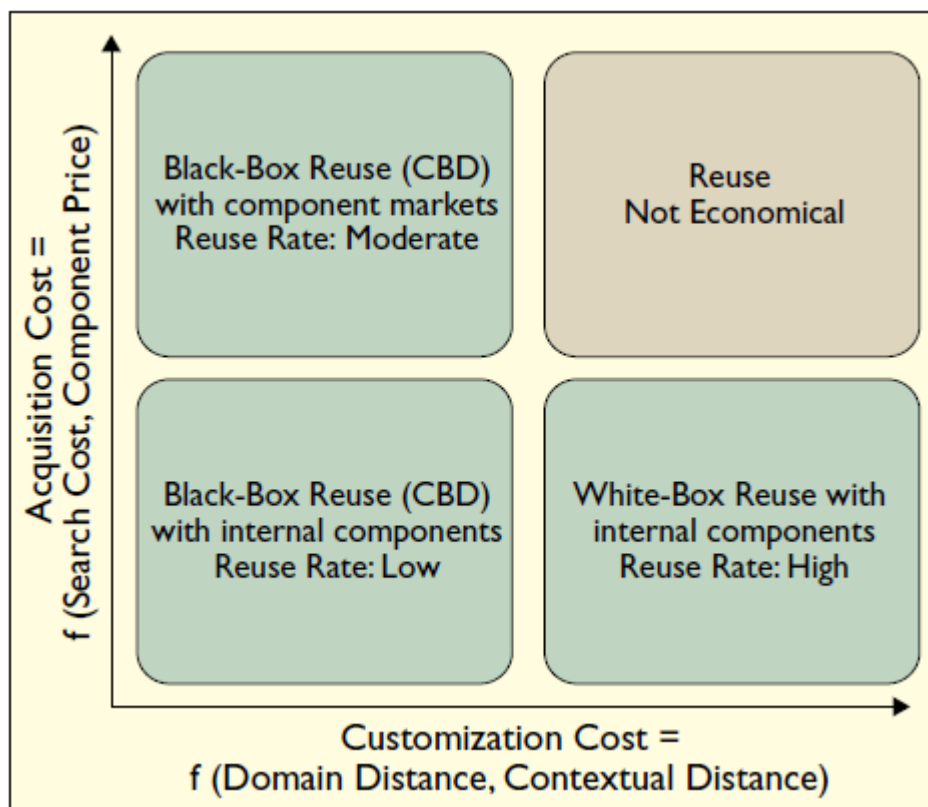


Figure 3, extracted from [2]

In picture 3 we can see a relation of costs of a solution (divided into acquisition costs and customization costs) development and its impact on the black-box and white-box models. We can see how a black-box component reuse is done when there is no much difference between the functionality that the component offers and the functionality that we are looking for in our solution. In that case, the acquisition cost will grow if the component comes from outside of the company or organization (since we'll have both to purchase and search and analyze deeper the candidate components we find). If the customization costs are high (id est, the components that we can find have a functionality that differs from the one that we need for

our solution) we'll choose then either a white-box component so we are able to modify it accordingly to our needs or have to develop the code by ourselves (this might mean use another development paradigm in that part of the solution or create our own component which we will be able to reuse in some future).

## **Services as components**

In the last few years, with the improvements of the internet connections, a new way of software usage has been arising. This stands for asking a remote process to make some computation and retrieve the results through the network. This kind of are called the services. Services are basically functionalities that can be used by a remote system. Thus, a software can be running somewhere and use a service that is located in another place on the internet (or intranet). This way, the application that is using the service doesn't need to know anything but the necessary inputs and outputs of the service, and the service will make all the computation by itself. The application that uses the service just needs to know the specification of the function it offers and have some code to let it perform remote invocations.

The idea of using a service as a component is, basically, that we can use a remote service instead of including a component with that functionality in our solution. This approach can bring lots of advantages.

Firstly, the fact that the code that performs the operations is not in our own system, makes it require less resources when it's being run (processor time, memory, etc). The remote location of the code also makes it easier to maintain, since any updates or modifications applied to it should only be developed once and applied once (in the remote location), and they will be reflected on the outputs that it gives to all the solutions that use it without having to change anything there.

Another factor to take into account is that most of the times services need to retrieve information from some database, that way it's also easier to put the database close to the service and only grant access to it instead of letting different solutions access the database or replicate it. This model also provides some advantages to the service provider, for example, it won't have to expose anything about how the back-end works.

Nowadays there are already enough technologies and standards to create and use services, so the developers don't really have to care about the details of the implementations or

protocols. The most used kind of services are the web-services (mainly SOAP and REST protocols), which are standards that also work on other standard protocols (HTTP and XML). There are also some remote invocation methods which work on their own standards but are quite used as well, such as .NET RMI or Java Remote Method invocation.

On the other hand we should mention that the use of services can have some disadvantages, for example the availability. Sometimes services are not available just because there has been some incident on the net, or the remote server has crashed, and most of the times the application user can't do anything to solve it, so if the application relies on that service it might not be able to perform its normal operations anymore. Another major disadvantage is the latency. For those applications that require fast operations (for example, operations related to graphics), the latency of the transaction the service will need might be too high.

## **Conclusions**

When developing new applications, we should seriously take into account the idea of using components in our development (either as a small part of our solution or the whole), since it can help in many ways such as reducing the time or the costs of the development itself.

We should also have in consideration the characteristics of the components we choose, and be able to make the right decision between using a component as a white-box or a black-box component, having in mind the needs of our solution and the advantages and problems these models can offer to us. In this decision we might also include the fact that we should try to reduce the costs of adopting a software component by looking for an adequate trade-off between the acquisition cost and the customization costs (this is, for instance, not trying to acquire a too expensive component if we can spend fewer resources in using a white-box one in our solution).

The last important point is to have in mind that services can help us in the componentization of our solutions and bring some advantages we cannot get with the "regular" componentization of our solutions. As the use of services is relatively new, it might be possible that it grows in the next few years, becoming a major trend.

## References

- [1] Messerschmitt, David G.(2007). Rethinking components: From hardware and software to systems. Multi-Campus: Retrieved from: <http://escholarship.org/uc/item/65t2v4cr> [last accessed November, 2009]
- [2] T. Ravichandran and Marcus A. Rothenberger (2003), "*Software reuse strategies and component markets*", Communications of the ACM
- [3] Clemens Szyperski and David G. Messerschmitt, "The Flexible Factory", Software Development, december 2003
- [4] *Michael Stal*, "Web Services: Beyond Component-Based Computing", Communications of the ACM October 2002/Vol. 45, No. 10
- [5] Kung-Kiu Lau and Zheng Wang, "Software Component Models", October 2007, IEEE Transactions on software engineering, VOL. 33, NO. 10
- [6] Kung-Kiu Lau and Zheng Wang, "A Taxonomy of Software Component Models", School of Computer Science, the University of Manchester
- [7] Parastoo Mohagheghi & Reidar Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies", Springer Science + Business Media, LLC 2007
- [8] "An Overview of the Koala Component Model", Distributed Systems Research Group, Charles University Prague, Faculty of Mathematics and Physics
- [9] C. Szyperski, D. Gruntz, S. Murer. "*Component Software: Beyond Object-Oriented Programming*". Addison - Wesley, 2nd edition, 2002.
- [10] FAQ section about CBD on the page of Veryard projects on users.globalnet, <http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm> [last accessed November, 25th 2009]
- [11] Carbide help center (used only for image extraction) [http://carbidehelp.nokia.com/help/index.jsp?topic=/S60\\_5th\\_Edition\\_Cpp\\_Developers\\_Library/GUID-441D327D-D737-42A2-BCEA-FE89FBCA2F35/SIPEXample/doc/index.html](http://carbidehelp.nokia.com/help/index.jsp?topic=/S60_5th_Edition_Cpp_Developers_Library/GUID-441D327D-D737-42A2-BCEA-FE89FBCA2F35/SIPEXample/doc/index.html) [last accessed November, 25th 2009]