Software Components — difficulties with acquisition

Benneth CHRISTIANSSON
Karlstad University
Division for Information Technology
Benneth.Christiansson@kau.se

Abstract. Many industry observers have characterized the problems associated with software development as a crisis. We have all seen and read about some spectacular software failures that have occurred over the past decade. But we believe that it is more accurate to describe the problem as a chronic affliction rather than a crisis. A crisis is a turning point where things can't get any worse, only better. A chronic affliction indicates rather that the problems are here to stay and maybe something we have to learn to live with and accept. This affliction is not only regarding software failure but also regarding software development, and the growing amount of already built software that demands maintenance. One possible solution to this chronic affliction may be in the use of software components. This implies developing software systems by joining a number of essentially standardized software components. These components can be developed in house or by a third party like COTS-components (Components Off The Shelf). In this paper we focus on the process of acquiring software components to be used as parts in software systems. This is a nontrivial and very essential issue regarding software component usage. The outline for this paper starts with a definition of the term "software component", this will establish a foundation for the paper and also indicates some of the problems with acquisition. The following chapter describes a process for developing systems consisting of software components, this process identifies an acquisition phase and this phase is further elaborated in the next section of the paper. Some problems and issues regarding acquisition are raised and focused. The paper ends with a summary and a discussion regarding future research in this area.

1. Introduction

Software systems constitute an essential part of most companies business backbone or infrastructure, and becomes increasingly more complex. In today's industry, these enterprises have to continuously adjust and improve their business practices to maintain a competitive edge [1][2]. Such changes to business practices often raise requirements for change in their software systems and the need for new systems. It is in this context that being able to assemble or adapt software systems with reusable components proves useful. Experience has shown that even with advanced technological support, in general, it is not an easy task to assemble software components into systems [3][4]. A major issue of concern is the mismatches of the components in the context of an assembled system, especially when the mismatches are not easily identifiable. The hard-to-identify mismatches are largely due to the fact that the functionality of the components are not clearly described or understood [5]. Most commercially available software components are delivered in its binary form. We have to rely on the components' interface description to understand their functionality is. Even with the components' development documentation available, people would certainly prefer or can only afford to explore their interface descriptions rather than digesting their development details. Furthermore, interface descriptions in natural languages do not provide the level of precision required for component understanding, and therefore have resulted in the above mentioned mismatches. Although the motivation for widespread use of COTS products is cost savings, there are many more unknowns that must be addressed from a business perspective. For instance, the unforeseen expense needed to keep appropriate "wrappings" on COTS products throughout the entire maintenance phase [6]. How should a program manager react if a commercial approach results in a higher life-cycle cost? What business case should be made in that circumstance? [7]. In this paper we have chosen to focus on the lack of ways to describe and identify component behavior, and more specifically only regarding the acquisition of components. This is a nontrivial and very essential issue regarding software component usage.

2. The software component

The term software component isn't easy to define, it does not have a clear-cut definition in the software development community, but the meaning fluctuates. This paper does not focus on the issue of defining the term software component even though this is something needed and hopefully soon to be done. Instead we support the definition that Christiansson [8] makes. This definition is based on a survey and symbiosis of several more or less well-established definitions. We start our discussion concerning the meaning of the term with two quotations:

> *"A component is a unit of software of precise purpose sold to the application developer community…with the primary benefit of eliminating a majority of the programming the buyer most perform to develop one or more function points…"* [9]

> *"A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort"* [10]

These definitions, we believe, illustrates several of the more basic criteria such as a software components binary shape and its ability to connect without reconstruction to other software components. However these definitions does not include issues, such as identifying, maintaining and refining software components. Therefore we suggest a more mature definition of the concept software component. In this paper we will use a definition made by Christiansson [8], this quotation is translated from Swedish.

> *" A software component:*
> *is independent and reusable;*
> *provides a defined functionality using a specific interface;*
> *can affect/be affected by other software components;*
> *should have a specification (in which the software component is described on a high level of abstraction);*
> *can have multiple implementations, meaning that the same component can be implemented in several programming languages; and*
> *can have several executable (binary) forms, i.e. the same component can be executed in different operating systems."*

The fact that a component is independent and reusable shows that a component can be used without other components present, the services provided by the component should be accessible without any external help except from the software glue and necessary run-time environment. A component can affect and be affected by other software components. This means that two components can "work together" and "as a whole" create a greater service than used separately. In figure 1 we illustrate a software component with a context.

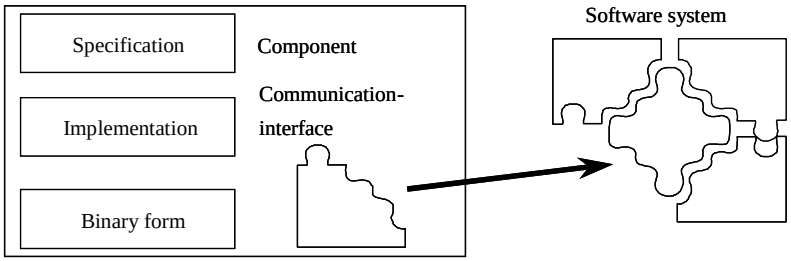

Figure 1. A software component with a context

The need for a documented specification for a software component is obvious if one consider the process of acquiring a component. How can one find a software component if one doesn't have something to look for. This is maybe the one single factor that can decrease the gap between the formal- and informal-part when developing a software system. If there are documented specifications, these can be described in such ways that they are useful when dealing with the development of the informal part of a software system and then the component as such is directly applicable in the formal part [8]. This notion of documented specifications can be elaborated to incorporate the need for standardization concerning specifying software components. Vaughn [11] implies that:

> *"A standard approach to building and using components must be set and universally practiced if the software engineering community is to reap the benefits…of reusable software components…"*

With this quotation we want to stress the fact that standardization of software component specifications will be one of the next issues for solving the fundamental problem with software development [12].

3. The composition of component-based software systems

A component-based software system is a more complex phenomenon than a traditional software monolith-based software system. A component-based software system can be regarded as consisting of sections on three different "levels" [7]. The innermost level is the component-architecture i.e. the components themselves and the necessary glue-code to enable their collaboration. The intermediate level is the application architecture, i.e. the grouping of cooperating components into software applications. The outer level is the software system architecture i.e. the software systems which the different applications support or of which they consist. These "levels" are illustrated by Fig 2.

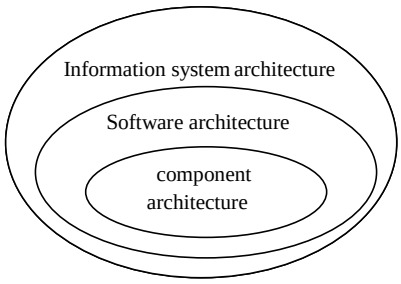

Fig. 2 The composition of component-based software systems.

Each of the levels should be represented by a separate architecture. When developing software systems using software components one have to be aware of these different architectures. A specific software component have to 'fit' the architecture chosen for the system [13]. This is a condition both regarding acquisition as well as developing components from scratch. The different architectures define how a specific software component has to be designed and constructed.

Component-based development will have to take into account all of the issues described in the section above. Of course the development processes will be affected by the architecture shown in the section above. Component architecture is very important to take into account as a component architecture may apply to a single application or to a wider context. This wider context can be a set of applications serving a particular business process [14].

The fact that components can, and should, be viewed from different perspectives is important for the quality of the component, and in the long run for the system which is composed by the components. A software system can be composed by components from different producers and different consumers can use the components in different combinations to fulfil the demands on the consumers' specific software systems. This also implies that the same component may have different lifecycles dependent on from which perspective you look at the component.

4. Component-based Development the phases

This section is based on the development process described in [7] You may find a detailed description and motivation of this process there.

4.1 Requirements analysis and definition

The analysis phase involves identifying and making understandable demands and requirements that should be met by the information system. In this phase the systems boundaries should be defined and clearly specified. Using a component-based approach, analysis will also include specifying components. The components will collaborate to provide the functionality defined as the system. To be able to do this there is a need to define the domain or system architecture that will enable component collaboration. Analysis is a task with three dimensions in CBD. The first dimension being capturing of systems requirements and defining system boundaries. The second dimension is defining the component architecture to enable component collaboration and the third dimension; defining component requirements to enable acquisition or development of the required components.

4.2 Component acquisition

In the following quotation Vigder and Dean [6] define the acquisition phase where the acronym COTS stands for Components Off The Shelf:

> *"A survey of COTS components available in the marketplace must be performed, and criteria established for selecting the appropriate components. The criteria range across a broad spectrum including run-time characteristics, documentation, vendor support, etc."*

This quotation shows that it is advisable to look over existing components on the market. To carry through this search and make identification possible, the component needs to be specified preferably in a standardized manner.

If the search for a specific component results in an component that only partly fulfils the specification on which the acquisition is based, there are two alternatives: either the component is adapted to its particular specification, or the specification is adapted to the component. The latter alternative may be considered radical, but there are advocates who will point out the advantages of this approach [5].

The acquisition phase, should start by examining the content in the internal component repository. If the component can't be found there, an external search is carried out, with component manufacturers. If the search does not result in a satisfactory way, the component requirements can be used to initiate a component development, which is developed either by refining existing components, or in a completely new development, see Figure 3. In this figure the triangles represent activities, the rhombi indicate choice and the arrows show the flow of work.

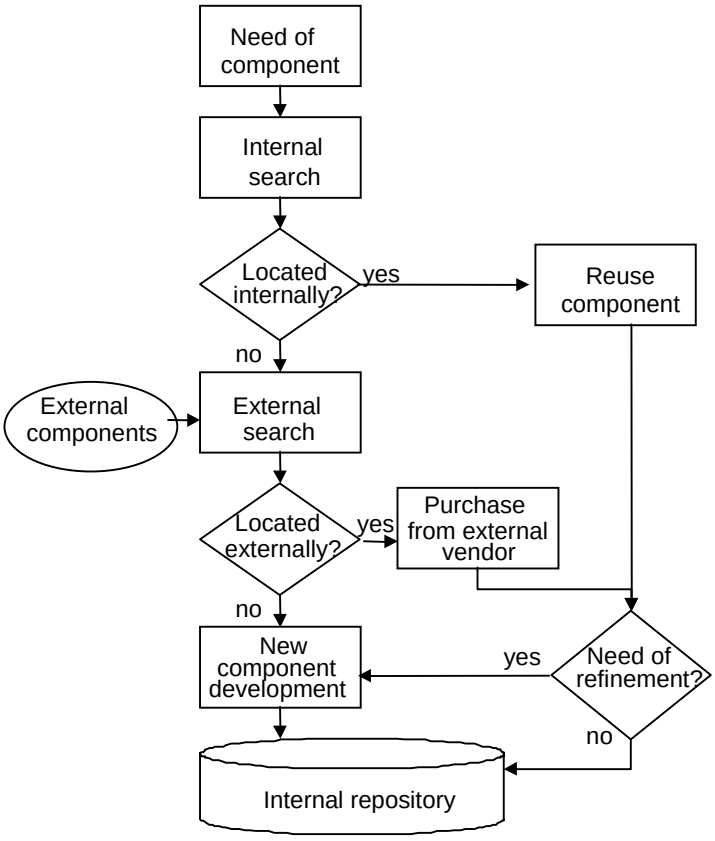

Figure 3. A flow chart showing the acquisition process [8].

If the acquisition phase has resulted in the development of a component, whether from new production or by refinement, it needs to be designed. This design should harmonize with traditional software design. However, some differences should be identified:

- In the design phase there should be a clearly expressed specification of the component [8];
- The design of components may be executed independently and parallel with the configuration of other components;
- The design should be accomplished so that reuse can be achieved;
- When components are designed, these must be adapted to a given component communication standard; and
- To be able to reuse a component it has to be designed in a more general way than a solution tailored for a unique situation. The component has a purpose of being reused which requires adaptability, this added adaptability will increase the size and complexity of the component.

However, to establish whether a component is unique or not, the design phase should initially be replaced by an acquisition phase for the components that were identified in the analysis phase. This is where the importance of using a standardized mode of expression for component specification is evident. If a standard is adopted, components can be selected in the acquisition phase that, partly or completely, meets the requested specification.

4.3 Implementation

In the implementation phase, the design will be transformed into software. In a traditional sense, this is done by designing or purchasing the required software from a vendor. With component-based development, this means that the implementation will proceed from design to assembly. Instead of creating software with traditional software development tools, finished software components are assembled to make up the system that was visualized in the design phase. This, according to some proponents of a component-based development, will lead to a greatly reduced workload, the notion of pluggable components are often used here. A pluggable component is a component that can connect to other components using a mutual communication standard. Programmers no longer need to design complete systems from scratch, but can use existing components for the assembly.
It should be mentioned, however, that some components will still need to be designed – those that are business critical or unique to a specific situation. Some components will require refinement to fit into a given solution.

4.4 Implementation testing

Another important distinction, compared to traditional software development, is the need for a comprehensive initial testing phase before using a component. When a component is acquired, it should be tested before it is integrated. A new component should be tested to check and verify that it performs the functions it should according to specifications. Perhaps it is even necessary to perform tests in order to understand what a component will do and how it should be implemented depending on the degree of documentation that comes with the component. Tests may also be conducted to achieve a better understanding

and knowledge of a given component. When a component has been acquired, it should be checked for performance fulfillment, to make sure it performs the task it was designed to do, and to ascertain that it is adequately reliable. Insufficient reliability could mean that all, or part of, the software system stops functioning or produces improper results.

4.5 Integration

Integration means that the implemented and acquired components are put together into the software system defined in the analysis phase. According to advocates of component-based systems development, no great resources are needed, as all it takes is a plug-in process, obviously, on the condition that you have the necessary basic systems architecture and use the correct communication standard for component collaboration. An important aspect of the integration phase is that it is not possible to discover all the effects of using a component until the integration is done. This means that the integration phase too should to a large extent involve testing, although here the tests will be based on the effects that the components have on each other.

4.6 Integration and system testing

The integration and system-testing phase consists of two major activities. Firstly the already mentioned integration tests to evaluate and examine component interoperability. The second, when the integrated software system is subject to a series of tests, to identify and eliminate defects and unwanted side-effects in the system, and to verify and check the quality of the system. A test phase does not automatically lead to all defects and shortcomings being eliminated, but means that a certain level of reliability is achieved.
The test phase should be tied to a given component rather than to a given software system. Also, when a component is integrated with other components, the integration should be tested [5].This will ensure that the acquired component does not contain any defects that will affect the functionality and reliability of other components.

In order to perform a test, you have to know not only what to test; how to perform the test; but also what the system is expected to accomplish. In traditional systems development, this implies that the results from the analysis and design phases are used to make up the required test cases. A problem in achieving necessary test cases is that the analysis results must be translated into concrete tests during the implementation phase. With component-based development this can be made easier as each component should have a specification describing what it is expected to perform, from which test cases can be designed.

5. Difficulties with acquisition

In this section we will focus on acquisition of already existing components. The acquisition can be internal (in house components) or external (purchased in a marketplace). The component acquisition is a nontrivial task. As described in previous section we identify a need to structure and preferably standardize the way we describe components to enable acquisition. Data, functional, and behavioral models (represented in a variety of different

notations) can be created to describe what a particular component or application must accomplish. Written specifications can then be used to describe these components or applications. A complete description of requirements is the result. Ideally, the requirements in the form of specifications, are analyzed to determine those elements of the model that point to existing reusable components. The problem is extracting information from the requirements in a form that can lead to 'specification matching' i.e. The possibility to identify existing components from the documented requirements. Components should be defined and stored in three different states specifications, implementations and binary executables. Ideal these components are an engineered description of a product from previous applications. The specifications we suggest should be stored in the form of reuse-suggestions, which should contain directions for retrieving reusable components on the basis of their description and for composing and tailoring them after retrieval. Bellinzona et. al [15] describes one very formal but possible way of doing this. A reusable software component can be described in many ways, one formal way encompasses what Tracz [16] has called the 3C Model-concept, content, and context. The concept of a software component is a description of what the component does [17] The interface to the component is fully described and the semantics represented within the context of pre- and post-conditions-is identified. The concept should communicate the intent of the component. The content of a component describes how the concept is realized. In essence, the content is information that is hidden from casual users and need be known only to those who intend to modify the component. The question we raise is how this information should be expressed.

Today there are three major areas for research regarding component specification they are: library and information science methods, artificial intelligence methods, and hypertext systems. The vast majority of work done to date suggests the use of library science methods for component classification. We do not prefer one of these strategies over another but instead we argue that the design of this description is the major difficulty concerning component acquisition. Christiansson [7] describes this as a need for component specifications (see previous section in this paper). All three of these methods rest on the foundation of rigorous and formal ways of defining component behavior and design. We believe that this is one important aspect of component specification but not always a very pragmatic and useful way of specifying. We argue for the use of Langefors [18] formal and informal way of defining information systems. We need two different or at least two parts of a specification one that describes the component in a formal way (this is where the vast majority of research is being done today) and one specification that describes the component in an informal way. This more informal specification, we believe, can be used for defining demands and as well acquiring and distribution purposes. For this more informal specification to be useful in the global marketplace we need to standardize the way we express this informal specification. We believe that in the same way as for instance CORBA and COM supplies standards for component construction we need standards for component specification both concerning the formal and informal part of the specification.

In today's software industry, automated tools are used to browse a repository in an attempt to match the requirement noted in the current specification with those described for existing reusable components. Characterization of functions and keywords are used to help find potentially reusable components. If specification matching yields components that fit the needs of the current application, the designer can extract these components from a reuse library (repository) and use them in the design of new systems. If components cannot be found, they may be acquired through a third party, or in house development. To be able to acquire components from a third party one need to be able to express requirements in a way that enables a global search through the marketplace for components. To be able to perform this search the component needs to be described in such a way that a global search is possible. We argue that function characterization and keywords are weak tools for this. We

need more mature ways of describing and searching for components. We argue for the need of specific languages and notations preferably one global standardized notation (such as for instance UML).

6. Summary and future research

A software component has a set of characteristics:
" *A software component:*

is independent and reusable;
provides a defined functionality using a specific interface;
can affect/be affected by other software components;
should have a specification (in which the software component is described on a high level of abstraction);
can have multiple implementations, meaning that the same component can be implemented in several programming languages; and
can have several executable (binary) forms, i.e. the same component can be executed in different operating systems."

The composition of a component-based software system can be described as consisting of three different levels or architectures .The innermost level is the component-architecture i.e. the components themselves and the necessary glue-code to enable their collaboration. The intermediate level is the application architecture, i.e. the grouping of cooperating components into software applications. The outer level is the software system architecture i.e. the software systems which the different applications support or of which they consist. The specific architecture will affect the criteria used for acquisition. One have to acquire components that fit the architecture they are intended to be used in.

The component acquisition is a nontrivial task. As described in previous section we identify a need to structure and preferably standardize the way we describe components to enable acquisition. Data, functional, and behavioral models (represented in a variety of different notations) can be created to describe what a particular application must accomplish. Written specifications are then used to describe these models. The design of this description is a major difficulty concerning component acquisition. The methods proposed can be categorized into three major areas: library and information science methods, artificial intelligence methods, and hypertext systems.

We argue for the use of Langefors (1995) formal and informal way of defining information systems. We need two different or at least two parts of a specification one that describes the component in a formal way and one specification that describes the component in an informal way. This more informal specification, we believe, can be used for defining demands and as well acquiring and distribution purposes. For this more informal specification to be useful in the global marketplace we need to standardize the way we express this informal specification. One area for future research is creating languages/notations for describing informal component specifications. In this area we intend to do our future research.

References

[1] Casanve C. (1995) *Business-Object Architectures and Standards*. Data Access Coorporation, Miami, USA.

[2] Sutherland J. (1996) *The Object Technology architecture: Business Objects for Corporate Information Systems.* OOPSLA'95 Workshop on Business Object Design and Implementation, Springer-Verlag, Berlin.

[3] Veryard R. (1998). *Making CBD effective in your organization.* (september) http://www.scipio.org.).

[4] Schach S. R. (1997) *Software Engineering with Java*. McGraw-Hill Companies Inc. New York USA.

[5] Vigder M. R, Gentleman W. M. & Dean J. (1996). *COTS Software Integration: State of the art.* National Research Council of Canada., Toronto, Canada.

[6] Vigder M. R. & Dean J. C. (1998). *Managing Long-Lived COTS Based Systems*. National Research Council of Canada, Toronto, Canada.

[7] Christiansson (2000) *Component-Based Systems Development – according to theory and practice.* (Licentiate Thesis, presented the 15:th of December at Karlstad University), Department of Computer and Information Science, Linköping University. (in Swedish).

[8] Christiansson, B. (2001) *Component-Based Systems development*. In: Nilsson A.G. & Pettersson, J.S. (eds.).On Methods for Systems Development in Professional Organisations, Studenlitteratur, Lund

[9] Steel J. (1996). *Component Technology Part I An IDC White Paper*. International Data Corporation, UK, London.

[10] Microsoft. (1996) *The Microsoft Object Technology Strategy: Component Software.* www.microsoft.com.

[11] Vaughn T. (1990) *Issues in Standards for Reusable Software Components*. (november, 1996) http://ruff.cs.umbc.edu.

[12] Christiansson B. (1999) *Komponentbaserad systemutveckling – genväg eller senväg?*, in Nilsson, A. G. & Pettersson, J. S. (eds.) *Om metoder för systemutveckling i professionella organisationer - Karlstadskolans syn på informatikens roll i samhället*, Studentlitteratur, Lund, (in Swedish).

[13] Szyperski C. (1997). Component Software Beyond Object-Oriented Programming. Addison Wesley Longman, Inc., California, Menlo Park, USA.

[14] Cheesman J. Daniels J. (2001) UML Components, Addison Wesley Longman, Inc., California, Menlo Park, USA.

[15] Bellinzona R., Gugini M. G. & Pernici B. (1995) *Reusing Specifications in OO Applications.* IEEE Software, March pp. 65-75.

[16] Tracz W. (1990). *Where does reuse start?* Proc Realities of Reuse Workshop, Syracuse University CASE Center, jan.

[17] Whittle B. (1995). Models and Languages for Component Description and Reuse. ACM Software Engineering Notes vol. 20, no. 2 April, pp. 21-22.

[18] Langefors B. (1995) *Essays on Infology*. Department of Information Systems, Studentlitteratur, Lund.