On Software Components Characterization and Specification

Abbas HeydarNoori, Farhad Mavaddat

Department of Computer Science
University of Waterloo, Canada
{aheydarnoori, fmavaddat}@cs.uwaterloo.ca

Abstract

Because of the increasing complexity of software applications, it is becoming more and more complicated to develop them from scratch. Developing software applications by using existing software components is a solution to this problem. But, before everything, we will introduce software components as a solution to software crisis and software engineering problems. By software crisis we mean the lack of productivity and performance in the software development projects. Using existing software components in the development of new software applications has some advantages and disadvantages. In this paper, we will talk about some of these issues. Another problem relates to the definition of software components itself. Unfortunately, there is no standard and specific definition of software components. In this paper, we try to define a comprehensive set of attributes for characterizing software components in a repository of software component products. These attributes are classified into six major categories: Source, Specification, Packaging, Modification, Requirements, and Quality attributes.

Key Words
Software components, Reusability, Software engineering, Software crisis

1. Introduction

In 1968, NATO organized a conference to discuss the issues in software development projects in large corporations. The problem was that software projects were often underestimated in time and cost and about 80% of them were never completed. So, the phrase "software crisis" came into existence. To overcome software crisis, "software engineering" was introduced. The aim of software engineering is using the engineering approaches in the development of software applications. But, there are still some problems in the software engineering. For example, issues in efficiency, usability, dependability and maintainability of software projects. To solve these problems, many different solutions have been proposed (e.g. outsourcing, CASE tools). Unfortunately, each of theses solutions has some advantages and disadvantages and none of them can definitely solve all of the software engineering problems. But, one of the most prominent proposals is the usage of existing software components in the development of new software applications.

Because, many software applications contain similar or even identical components, there is no need to redevelop them from scratch. Though, component-based software development (CBSD) has this potential to solve many of the software engineering problems, it is still in its infancy and a lot of work should be done. Before everything, this important question should be answered: "What is a software component?". Unfortunately, there are a lot of different answers to this important question and there is not any standard and specific answer to this question yet. Another important question is "Which characterization attributes should be used to describe a software component?". There is not any specific answer to this question either. In this paper, a set of attributes or properties for software components will be proposed which can be used to characterize them in a repository of software component products. These attributes are classified into six major categories: Source, Specification, Packaging, Modification, Requirements, and Quality attributes. You may

find different sets of component attributes in different literatures [1, 2, 3, 4]. But, in our work, we tried to define a comprehensive set of attributes which encompasses all of those sets.

This paper is organized as follows. In section 2, software crisis and software engineering are introduced. Then, in section 3, some solutions to software engineering problems are discussed and component-based software development is introduced as one of these solutions. In section 4, this question will be answered: "What is a software component?". Section 5 talks about pros and cons of using software components in the development of new applications. In section 6, we will define a set of attributes for characterizing a software component in a repository of software component products and section 7, concludes.

2. Software Engineering Problems: Software Crisis

The phrase "*software crisis*" was declared first at the first NATO conference on software engineering in 1968 [5]. A survey done by Standish Group in 1994 [6], showed that 31% of software projects were aborted prior to completion. In this study, 350 companies with over 8000 software projects were analyzed. This survey showed that in large development companies, 9% of all projects and in small development companies, 16% of all projects were completed within project budget and time limits. Another survey by IEEE software in 1995 [6], showed that 30% of all software projects are cancelled, 50% of all projects are 150% over budget and only 60% of functionality is in the final product.

The phrase "software crisis" shows the lack of productivity and performance in the software projects. It also, shows that software developers are not capable of satisfying the needs of their customers and users [7]. In other words, software crisis is expressed by delays and failures in software projects tasks that result in low quality software, unpredictable costs and times, and unreached goals. The symptoms of software crisis are:

1. Unacceptable quality of software
2. Not completion of software project within the estimated time and/or budget
3. Failure in software development project management
4. Abortion of software projects before completion

There are a lot of reasons for software crisis. Here are some of them [6]:

1. No similar systems ever built before
2. Requirements are not well understood
3. Requirements change during the software life cycle
4. Software is very flexible

To overcome these problems, "*software engineering*" came into being. The phrase "software engineering" was first introduced by one of the study groups of NATO in 1967 [6]. The intent of software engineering is using the engineering principles and methods in the development of software projects. In other words, software engineering is an attempt to base the software development on an engineering approach with well defined inputs, well defined outputs and well defined methods [6]. The goal of software engineering is to produce systems that are correct, efficient, reliable, useable, maintainable, which satisfy their specification.

There are still issues in the software engineering. Here are some of them:

1. How to ensure the correctness, quality and maintainability of software systems?
2. How to meet the estimated time and budget for the software project development?
3. How to divide large systems into smaller manageable subsystems?
4. How to ensure productivity of the software project development?
5. How to answer the changing requirements during the software development?

In the next section, we will talk about the proposed solutions to these problems.

3. Solutions to Software Engineering Problems

As mentioned before, there are still some issues in the software engineering and a lot of research is being done to solve these problems. Brooks in his classical paper on software crisis [8], mentioned some ways which can help us to get rid of these issues: high level languages, expert systems, software environments, incremental development, requirements refinement, prototyping, reuse and great designers [9]. In different literatures, some other solutions have been proposed too: outsourcing, open source, visual environments,

CASE tools, software component repositories, object-oriented environments and so on. But none of these solutions can solve the software engineering problems definitely and there are some pros and cons with each of them. But, one of the most promising proposals is reusing existing software components in the development of new applications or component-based software development (CBSD).

Making a system out of components and integrating these components together in order to develop a new system is a common approach used in hardware design [10]. The success of this methodology in hardware design, inspired software experts to use this idea in software design, because many software systems encompass many similar or even identical components and there is no need to redevelop them from scratch. Also, because of the increasing complexity of software systems, it is becoming more and more difficult to redevelop software components from scratch.

In CBSD, the following steps are followed:

1. Identification of reusable components
2. Retrieval of reusable components from a set of available components
3. Adapting the retrieved components to specific needs
4. Integrating the adapted components into the current application

In the following sections, we will talk more about software components and their characteristics.

4. Software Components: What are they?

In this section we want to answer to this important question: “What is a software component?”. Up to now, there is no standard and specific definition of software components. But, the way in which a software component is defined is a key factor in many of the CBSD activities [11].

Here are some different definitions of software components:

- Jose M. Troya and Antonio Vallecillo [12] believe that, components can be seen as encapsulation of programs. The “capsule” abstracts the program functionality, offers a common interface to the program services, hides their implementation and allows the composition and coordination of components.

- Alan W. Brown and Keith Short [11], characterize a component as an independently deliverable set of reusable services.

- H. John Reekie and Edward A. Lee [13] defined a component as a piece of software that can be plugged into some other piece of software. In their definition, a software component has clearly defined characteristics and clearly defined behaviour in the domain of interest. Also, they told that, the context of production and consumption maybe quite different.

- Sherif Yacoub and ET. al. [4] defined a component as an independent replaceable part of the application that provides a clear distinct function. In their definition, a component is a unit of composition, with predefined dependencies on other components.

- Clemens Szyperski in his book [14] introduced a component as a unit of composition with contractually specified interfaces and explicit context dependencies only. Also, he told that a software component can be deployed independently and is subject to composition by third parties.

- Wojtek Kozaczynski [15] provided the following definition: A component is a non-trivial, nearly independent and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Though these are different definitions of software components, there is a common idea behind all of them. In all of these definitions, a software component has the following features:

- It is a unit of software implementation that can be reused in different applications.
- It has one or more predefined interfaces.
- The inside of the component is hidden from the outside of it.
- It does a specific function.

5. Software Components: Pros and Cons

In this section, we will talk about pros and cons of using software components in the software development process.

One of the main advantages of using software components is the addition of "divide and conquer" or modularity to the software development process. By using software components, a big problem can be divided into smaller ones and then, for each subproblem, it can be checked that whether there exists a software component for solving that subproblem or not. If yes, it can be used in the development process. By this approach, higher quality software systems can be developed within lower time, cost and effort. So, it increases efficiency and productivity.

In addition, by using software components, it is possible to have more reliable software systems. Because these components have been reused several times in different situations and they were tested under a larger variety of conditions [16], fewer errors would occur and maintenance costs would be reduced [17]. Also, by using qualified components, debugging of the software would be much easier.

If the interface of a component does not change, a component implementation can be simply substituted by a newer version of that component. So, a software application which is built by software components is more maintainable. Also, it is possible to add new functionality to the application over time by adding new components to the existing application in order to answer changing requirements [14].

In CBSD, software developers should develop components which can be used in different conditions. So, they should follow a predefined standard. Thus, another profit of software components is the standardization of software development process.

As we know, in CBSD we have this opportunity to use software components which are developed by third parties (COTS[1]). At this situation, developers can focus more on the business problems by taking advantage of using existing market proven, vendor supported components [18].

Although there are lots of advantages in using software components, there are also some disadvantages and problems too. One of the main difficulties in CBSD is the difficulty to find suitable components. This raises risks such as using a software component which does not sufficiently satisfy the requirements regarding reliability, suitability, correctness and interoperability [19].

Another difficulty relates to the composition and packaging of components. The selected components might be compositional mismatch and it is impossible to interconnect them successfully [20].

If COTS components are used, some other issues may arise too. Most of the times, the COTS component source code is not available to the developers and so, they do not have enough control over the component evolution. If that component does not satisfy their needs, they should ask the vendor of that component to make the desired changes. This is often impossible. Because the level of support that is available from COTS vendors varies significantly, so the reliance on vendors may make the adjustments to the component much slower.

Another problem in using COTS products is the hidden dependencies amongst software components which software developers are unaware of them [21].

Another problem relates to the project budget. Some COTS components may be too expensive for that project.

6. Software Components: Characteristics

In this section, a characterization framework for software components will be proposed. Suppose that we want to have a repository of software component products. Then, whenever a new application is being developed by using existing components, simply this repository can be searched through and suitable components can be found. Before using a component in an application, the following two questions should be answered: (1) Is that component the one we need? (2) Is the quality of that component high enough for our system?. Because the key success factor of any CBSD project is the availability of carefully validated software components, answering to these questions is very important.

[1] Commercial Off-The-Shelf

For answering these questions, before everything, a set of attributes or properties for software components should be defined which can characterize them from different aspects. After careful consideration of existing literatures, we proposed some attributes for characterizing components. These attributes are classified into six major categories: Source, Specification, Packaging, Modification, Requirements, and Quality attributes. In the following sections, these categories are described.

6.1. Source

This category relates to the source or origin of the component. This category includes the following attributes:

Origin of the component: Where was this component developed? Some different values for this attribute could be: internally developed, externally developed, and existing but externally developed.

Availability: If a component is selected to be used in an application, whether it is available or not? To obtain that component, what tasks should be done?. As an example, if that component should be bought, whether it is available in the market or not?

Price of the component: If the component has to be bought, how much money should be paid? If the component is being bought, different license types may be acquired.

Component age: How old is this component? When a component is completely new, then it is more possible to have some undetected errors. So, the risk of using that component would be higher. After being used for several times, more errors would be detected and be solved and so, the reuse risk decreases [3, 4].

Product support: A list of the facilities provided by the component vendor to the component client.

Change Frequency: On the average, how many versions of this component have been released per year? Whether that component changes frequently or not?

6.2. Specification

This category includes attributes which describe the component from different views. This category encompasses the following attributes:

Purpose: What is the functionality or purpose of this component? What does this component do? Which problems can be solved by this component? [4] By this property, an informal description of the functionality of that component is provided. The component functionality can be classified into *horizontal* or *vertical* functionality [2]. In the vertical functionality, that component is specific to a specific domain (e.g. accounting systems), but in the horizontal functionality, that component is not specific to a specific domain and can be used in different domains (e.g. web-based systems).

Domain: This component is specific to which domain? For example, e-commerce, real-time systems, operating systems, and so on.

Related Components: Other known components which can solve the same or similar problems [4].

Documentation: In order to effectively use a component in an application, one of the most important characteristics of that component is its documentation. Documents of a component provide much useful information about that component: how to use that component (e.g. user manual), how to configure that component, etc. Documentation could have different forms: online, paper-based, interactive, etc. [17].

Also, one of the important parts of documentation is the analysis, design, and implementation documents of that component. If we had this authority to make changes to that component, then these documents are very important.

Standard Conformance: A list of standards which this component is compatible with them (e.g. COM, CORBA, DCOM, etc.) [3]. Because components in an application should interact with each other, it is necessary to all of them to be compatible with the same standard.

Technology: This property specifies the technology that this component is based on. For example, which programming language, operating system, or compiler was used in the development of this component? Anyway, this information can be part of the component documentation.

6.3. Packaging

As mentioned before, for developing new applications by using existing components, we need to package or interconnect these components together. This category talks about this issue and includes the following attributes:

Interface: As told before, the aim of the CBSD is to use existing components to build new applications. In such applications, each component provides some services to other components and requires some services from other ones. So, the interconnection and relationship between components must be clearly defined. In other words, interfaces are the mechanisms by which information is passed between two communicating components [22].

In [22], component interfaces are classified into direct and indirect interfaces. In direct interfaces, components communicate with each other, directly. In this method, a component directly invokes services from other components. But, in indirect interfaces, components invoke services from other components via a middleware.

In fact, an interface can be seen as a contract between a component which uses the services of another component (client), and a component which provides some services to other components (provider) [22].

One of the interesting points of interfaces is that they hide the details of implementation from the clients of that component. So, a component is treated as a black box. An interface, determines how a client component should interact with a provider component. But, it does not tell anything about how those services are implemented. Another interesting point is that a single component may have multiple interfaces. So, clients of a component may make use of one or more of these interfaces, as appropriate [11].

Interoperability: This property shows that how this component interacts with other components in the same application [4]. This characteristic determines how this component provides some services for other components or get some services from them (e.g. in a client/server manner). This property is much related to the interface property of components.

Another interesting point is that some components may play a *passive* or *active* role in an application [4]. In the passive role, this component is affected or invoked by other components. But in the active role, this component can affect other components too.

6.4. Modification

After selecting the desired components, maybe it is necessary to do some modifications on them to be able to use them in the desired application. This category includes the following attributes:

Possible modification: Whether it is possible to make desired changes on the component? How much this component is modifiable? What changes can be made to this component? How much configurable is the interface of this component? How much extensible is this component?

Required modification: What changes should be made to the desired components to make those components suitable for using in this application?

Accessibility to source code: Whether the source code of the component is accessible or not? If it is accessible, making change to the source code is authorized or not?

6.5. Requirements

In order for a component to work properly, its requirements should be noticed. For example, a component may require a special kind of hardware to do its intended function. Also, during the selection process of components, we should notice their requirements too. For example, in a system with memory constraints, a component which needs a lot of memory is not suitable. This category encompasses the following attributes:

Hardware requirements: A list of the hardware elements required by this component. For example, CPU speeds, memory, disk space, and so on.

Software Requirements: Similar to the hardware requirements, usage of a component needs a software platform too. For example, operating system, DBMS, networking software, requirements from other components, and so on. Also, for developing applications by that component, some special software tools and environments might be needed too.

Run-time requirements: In addition to hardware and software requirements, components may need some run-time requirements either, for example, CPU and memory usage at runtime.

6.6. Quality

As mentioned before, a key success factor in the development of new systems from existing components is the usage of well qualified software components. This category talks about different quality attributes. One of the problems with quality attributes is that different people may have different interpretations of quality and so,

there is still no agreement over the assessment of these attributes. This category includes the following attributes:

Performance: This property shows the performance of a component. Different users in different domains may have different definitions of this property. For example, in a real-time system, the response time of that component may show the performance of it. In another system, the performance can be defined as the number of users who can use that component without any decrease in its response time [3]. So, before entering the software component specifications into the repository, the interpretation of this attribute should be determined.

Security and safety: How much secure is this component? Whether every unauthorized input and output is prohibited? Can this component be used in secure systems?

Reliability and fault tolerance: The reliability of software is defined as the probability of failure-free operation of that software in a specified environment for a specified time [23]. This probability can be calculated by using historical data about that component.

Correctness and accuracy: By definition, correctness is the degree to which the component performs its required function correctly [23]. For correctness verification, some software tests should be done. This property has to show what tests have been done on this component.

User satisfaction: A criteria for selecting a component from a set of components is checking the satisfaction of the users of that component who have used it before, using their experiences, checking whether they have had any problem with that component or not, and what their problems were. In other words, what percentage of users is satisfied with that component?

Learnability: For an average user, how much time is needed to learn how to use this component? Whether user training is hard? As a measure, how many person-hours are needed for teaching the usage of this component?

Portability and Adaptability: Portability means the required effort for transferring the software component from one hardware and/or software platform to another. In other words, it describes the degree to which a component is unconstrained by the choice of execution platforms [17]. So, this property shows that how this component can be ported from an underlying platform to another one in order to accommodate to different interfaces, protocols, hardware, software and so on. Adaptability means whether the component can be adapted with minimal effort and impact on the component content [17].

Replaceability: Whether it is possible to replace this component with another one without any changes in other components and interfaces?

7. Conclusion

In this paper, first we talked about software crisis. We introduced software crisis as delays and failures in software projects that lead to low quality software, unestimated times and costs and unreached goals. Then, we introduced software engineering as a solution to software crisis. However, the problems exist yet and software engineering itself has some problems too. Some solutions have been proposed to these problems and component-based software development (CBSD) has been introduced as one of these solutions. Then, we tried to define software components and their characteristics. We introduced a set of attributes for characterizing software components in a repository of software component products and then, we classified them into six major categories: Source, Specification, Packaging, Modification, Requirements, and Quality attributes. Also, we talked about pros and cons of reusing existing components in the development of new software applications.

References

[1] C. Abts, B. W. Boehm, E. B. Clark, *COCOTS: a COTS software integration cost model - model overview and preliminary data findings*, In Proceedings of the ESCOM-SCOPE 2000, April 2000, Munich, Germany, Shaker Publications, pp. 325-333

[2] M. Morisio, and M. Torchiano, *Definition and Classification of COTS: a proposal*, In Proceedings of the 1st International Conference on COTS Based Software Systems (ICCBBS) (LNCS 2255), February 2002, Orlando, FLorida, USA, pp. 165-175

[3] M. Torchiano, L. Jaccheri, C. F. Sørensen, and A.I.Wang, *COTS Products Characterization*, In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), July 15-19, 2002, Ischia, Italy, pp. 335-338

[4] S. Yacoub, H. Ammar, and A. Mili, *Characterizing a Software Component*, In Proceedings of the 2nd International Workshop on Component-Based Software Engineering, in conjunction with IEEE/ACM 21st International Conference on Software Engineering (ICSE99), May 1999, Los Angeles, CA, USA

[5] W. Wayt Gibbs, *Software's Chronic Crisis*, Scientific American, September 1994, pp. 86–95

[6] S. Leue, *Software Engineering: Introduction and History*, http://tele.informatik.uni-freiburg.de/~leue/IU/it460.part1.pdf, last visited: November 5, 2003

[7] E. A. Karlsson, S. Sorumgard and E. Tryggeseth, *Classification of Object-Oriented Components for Reuse*, In Proceeding of the 7th International Conference TOOLS Europe'92, 1992, Dortmund, Germany

[8] Jr. Frederick P. Brooks, *No silver bullet: essence and accidents of software engineering*, IEEE Computer, 20 (4), 1987, pp. 10-19

[9] G. Butler, *Quality and reuse in industrial software engineering*, In Proceedings of Asia-Pacific Software Engineering Conference and International Computer Science Conference, December 1997, Hong Kong, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 3-12

[10] Z. Fan, *Implementing Models of Computation using uC++*, M.Sc. Thesis, University of Waterloo, Canada, March 2003

[11] A.W. Brown and K. Short, *On Components and Objects: The Foundations of Component-Based Development*, In Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97), June 1997, Pittsburgh, PA, USA, IEEE Computer Society Press, pp. 112-121

[12] J. M. Troya and A. Vallecillo, *On the Addition of Properties to Components*, In Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP'97), June 1997, Jyvskyl, Finland, pp. 95-103

[13] H. J. Reekie and E. A. Lee, *Lightweight Component Models for Embedded Systems*, Technical report, Electronics Research Laboratory, University of California at Berkeley, UCB ERL M02/30, October 2002

[14] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 1999

[15] W. Kozaczynski, *Composite Nature of Component*, In Proceedings of the 1999 International Workshop on Component-based Software Engineering, May 1999, pp. 73-77

[16] M. Goulão and F. B. e Abreu, *Towards a Components Quality Model*, Work in Progress Session of the 28th Euromicro Conference - Euromicro 2002, Dortmund, Alemanha, 2002

[17] S. Yacoub, A. Mili, C. Kaveri, and M. Dehlin, *A Hierarchy of COTS*

Certification Criteria, In Proceedings of the 1st Software Product Line Conference (SPLC1), August 2000, Denver, Colorado, USA

[18] A. M. Aitken, *Components and Component-Based Software Development*, In Proceedings of the 3rd Western Australian Workshop on Information Systems Research (WAWISR), November 2000, Edith Cowan University, Perth, WA, USA

[19] M. Ochs, D. Pfahl, G. Chrobok-Diening, B. Nothhelfer-Kolb, *A COTS Acquisition Process: Definition and Application Experience*, In Proceedings of the 11th ESCOM Conference, 2000, Shaker, Maastricht, pp. 335-343

[20] R. Wuyts and S. Ducasse, *Composition Languages for Black-Box Components*, In Proceedings of the 1st OOPSLA Workshop on Language Mechanisms for Programming Software Components, October 2001, Tampa Bay, Florida, USA

[21] A. Orso, M. J. Harrold, and D. S. Rosenblum, *Component Metadata for Software Engineering Tasks*, In Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000) (LNCS 1999), November 2000, Davis, CA, USA, pp. 126-140

[22] S. Yacoub, H. Ammar, and A. Mili, *A Model for Classifying Component Interfaces*, In Proceedings of the 2nd International Workshop on Component-Based Software Engineering in conjunction with the 21st International Conference on Software Engineering (ICSE99), May 1999, Los Angeles, CA, USA

[23] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001