Few Important Considerations For Deriving Interface Complexity Metric For Component-Based Systems

Nasib S. Gill
Department of Computer Science & Applications,
Maharshi Dayanand University, Rohtak – 124 001
Haryana (India)
Email: nsgill2000@yahoo.com, nsgill_2k4@yahoo.com

P.S. Grover
Department of Computer Science
University of Delhi,
Delhi, India
Email: groverps@hotmail.com

Abstract

Component-based software engineering (CBSE) represents an exciting and promising paradigm for software development. Software components are one of the key issues in CBSE. The software development community is continuously seeking new methods for improving software quality and enhancing development productivity. There is an increasing need for component-based metrics to help manage and foster quality in component-based software development. The traditional software product and process metrics are neither suitable nor sufficient in measuring the complexity of software components, which ultimately is necessary for quality and productivity improvement within organisations adopting CBSE. In this paper, we propose an interface complexity metric (ICM) aimed at measuring the complexity of a software component based on the interface characterisation model of a software component that mainly include such as interface signature, interface constraints, interface packaging and configurations. Based on the value of this metric, the complexity of the software component could be managed within reasonable complexity limits. In this way, the software components could be kept simple which in turn help in enhancing the quality and productivity.

Keywords: Software components, interface characterisation, interface complexity, interface complexity metric.

1. Introduction

Component-based software engineering (CBSE) has recently attracted tremendous attention from both the software industry and the research community. It has been widely recognised that more and more software systems are being built by assembling existing and new components. A lot of research efforts have been devoted to analysis and design methods for component-based software. Although there are many published articles addressing in building component-based programs, hardly very few of them address about component-based measurement [1, 3].

Component-based software development is the process of assembling software components in an application such that they interact to satisfy a predefined functionality [10-13]. Each component will provide and require pre-specified services from other components, thus, the notion of component interfaces becomes an important issue of concern. A key to the success of CBSE is its ability to use software components that are often developed by and purchased from third parties. As such, the software components should be adequately specified or packaged through their interfaces to facilitate proper usage [4-9, 14].

The main objective of this paper is to propose an interface complexity metric of a software component that is capable of measuring the interface complexity of a software component for which interface characterisation of software component is a pre-condition.

The present paper is split mainly in four sections. First section of this paper emphasises on the necessity of interface characterisation of a software component. Second section of this paper elaborates the interface characterisation of a software component that is necessary for its proper usage in the component-based software development. The types of complexities contributed by the different interface paradigms of a software component are discussed in section 3. Interface complexity metric is proposed in section 4 of this paper. Conclusions and future directions are discussed in the last section of this paper.

2. Necessity of Interface Characterization of Software Component

Before we detail out the interface characterization of software components, let us first discuss how the interface characterization would help. Proper characterization of software components will be helpful in many ways [2]. Few of the expected benefits of characterization of software component are:

Better Usage

- Understanding the characteristics and properties of a specific component means better usage and ensures usage of the correct component in particular application development.

Better Understanding of Architecture Problems

- Identifying characteristics of components helps in solving architecture mismatch problems such as the nature of the component and its control mode.

Better Retrieval From Component Libraries

- Characterizing components could facilitate selection, acquiring, and acquaintance of components.

Better Cataloguing

- Understanding the characteristics of a component plays a major role in documenting, cataloging, and classifying the wide literature of available components.

3. Interface Characterization of Components

Proper characterization of software components is essential to their effective management and use in the context of component-based software engineering (CBSE). Characterization of components through comprehensive interface definition is a step towards such systematic approaches and their enabling technologies [2].

A model for comprehensive component interface characterization was proposed in [2] to provide a basis for the development, management and use of components.

The interface characterization of a software component is described as under:

Signature

At the bottom level, there is the *signature* of the component, which forms the basis for the component's interaction with the outside world and includes all the necessary mechanisms for such interaction (i.e., properties, operations and events).

Constraints

The next level up is the *constraints* on the component signature that must be satisfied for the proper use of the component.

The component signature and its constraints define the overall capability of the component.

Packaging

The third level concerns the *packaging* of the interface signature according to the component's roles in given scenarios of use, so that the component interface has different *configurations* depending on the use contexts.

Non-functional Properties

The fourth aspect of component interface is about the characterization of the component in terms of their *non-functional properties* [3, 9]. The non-functional properties occupy a special place in this component interface structure, and may interact with the interface signature and configurations.

4. Interface Complexity of Software Component

Taking into consideration the overall interface characterization of a software component, we may identify the interface paradigms that mainly contribute towards the overall interface complexity of a software component.

Important interface paradigms of a software component mainly include the following:

4.1 Interface Signature

Interface signature characterizes the functionality of the component. The component interface signature forms the basis of all other aspects of the component interface [2].

The interface signature of a component is comprised of *propertie*s, *operations* and *event*s. The overall complexity of interface signature of a software component is mainly due to operations and events while keeping in view the properties of software component.

For better understanding of interface signature, all its constituents are discussed in detail below.

Properties

A software component may have a number of properties externally observable. These properties form an essential part of the component interface, i.e., the *observable structural elements* of the component.

The users including people and other software components may observe and even change the property values, to understand and influence the component's behavior.

A common use of component properties is to customize and configure the component at the time of use. It should be noted that certain component properties can only be observed, but not changed.

Operations

Another aspect of a component signature is the operations, with which the component interacts with the outside world.

The operations capture the dynamic behavioral capability of the component, and represent the service/functionality that the component provides.

Events

Besides proactive control, another form of control used to realize system behavior is reactive control. It is often the case that certain aspects of a system are better captured through proactive control via operations, while other aspects of the system are better captured in the form of reactive control via events.

To facilitate reactive control, a component may generate events from time to time, and there may be none or many responses to an event from other components in the system.

4.2 Interface Constraints

In addition to the constraints imposed by their associated types, the properties and operations of a component interface may be subject to further constraints regarding their *us*e. There shall be a definite complexity contributed by various interface constraints. These interface constraints are discussed in detail below.

In general, there are two types of such constraints:

- *Those on individual elements and*
- *Those concerning the relationships among the elements.*

Examples of the *first type* are the definition of the operation semantics (say, in terms of pre-/post-conditions) and further range constraints on properties.

There are a variety of constraints of the *second type*. For example, different properties may be inter-related in terms of their value settings.

An operation can only be invoked when a specific property value is in a given range. One operation has to be immediately invoked after another operation's invocation. The explicit specifications of these constraints are important.

First of all, they form part of the defining characteristics of the component. They make more precise the capability of the component. Furthermore, it is essential for the component user to understand these constraints, so that it can be used properly. Without such constraints, the proper understanding and use of the component will be much harder. The interface definition of a component may well be the only available source of information about the component.

4.3 Interface packaging and configurations

The signature and constraints of a component define the overall capability of the component. For the component to be used, certain packaging is required [2]. It involves two aspects:

- *The component plays different roles in a given context, and*
- *The component may be used in different types of context.*

In a particular use scenario, a component usually interacts with a number of other components, and plays specific roles relative to them. The interactions between the component concerned and these other components may differ depending on the components and their related perspectives.

When interacting with a particular type of component from a specific perceptive, for example, only certain properties are visible, only some operations are applicable, and some further constraints on properties and operations may apply.

The scenarios provide the contexts of use for component. A component may be used in different scenarios and has different role partitions in these scenarios. Therefore, a component may have different sets of interaction protocols, with each set for a scenario in which the component is to be used. This suggests that a component may have different interface configurations. In principle, an interface configuration should be defined in terms of both the component and the use scenario, and it relates the component to the use context.

When a component is designed, the designer usually has one or more use scenarios in mind. Therefore, a few packaging configurations may be defined for the component interface.

When a new use scenario is discovered, a new packaging configuration may be defined.

The importance of interface packaging can not be over emphasized. It serves to relate the component to a context of use. In fact, much of the requirements for the component is derived from the use scenarios. The roles that a component plays in a use context are vital to the architectural design of the enclosing system. It provides the basis for defining the interactions between the components of the system and realizing the system functionality. It enables the relative independent development of the system components with clearly defined interfaces as well as requirements [2].

4.4 Non-functional properties

Another aspect of a component is its non-functional properties, such as security, performance and reliability. In the context of building systems from existing components, the characterization of the components' illities and their impact on the enclosing systems are particularly important because the components are usually provided as black boxes [2, 4, 9].

5. Necessary Elements For Interface Complexity Metric

Measurement of interface complexity of software component should be based mainly on the attributes characterizing interface(s) of a software component. Thus, the interface complexity metric of a software component should involve the measurement of all interface aspects discussed in above sections of this paper that mainly define the overall capability of a software component.

Thus, the interface complexity metric of a software component, S, should include complexity element related to interface signature, interface constraints and interface configurations of a software component that define the overall capability of the component, such that:

$$\text{CICM}(S) = a\, C_s + b\, C_c + c\, C_g$$

where

C_s is the complexity contributed by interface signature, C_c is the complexity contributed by interface constraints, and C_g is the complexity contributed by interface configurations of the software component.

a, b, and c are the respective coefficients for C_s, C_c and C_g, and are dependent on the nature of software component and the nature of its interfaces.

The signature complexity, C_s, of a software component is mainly contributed by the events and operations supported by software component. Thus, the signature complexity could be arrived as a function, f_s, of events and operations of a software component:

$$C_s = f_s (E, O)$$

Where E and O represents the total number of events and operations of a software component respectively. Thus, the signature complexity

$$C_s = a_1 O + b_1 E$$

a_1 and b_1 are the coefficients for operations and events respectively, which could be determined on the basis of the type of the software component, the nature of the application area and the properties of the software component.

The constraint set C of a software component is

$$C = \{c_1, c_2, c_3, \ldots\ldots.., c_k\}$$

where $c_1, c_2, \ldots., c_k$ are the k different interface constraints related to the component signature to ensure their proper usage. The constraint complexity may be defined as:

$$C_c = \sum \alpha_i(c_i)$$

Where $\alpha_i(c_i)$ represent the total number of interface constraints of c_i type depending upon the constraint's complexity.

The configuration set F of a software component is

$$F = \{f_1, f_2, f_3, \ldots\ldots.., f_m\}$$

where $f_1, f_2, \ldots., f_n$ are n different interface configurations in accordance to the component's roles in given contexts of use. Similarly, the interface configuration complexity may be defined as:

$$C_g = \sum \beta_j(f_j)$$

Where $\beta_j(f_j)$ represent the total number of interface configurations f_j type depending upon the configuration's complexity.

Thus, the overall interface complexity of a software component, S, may be summed up as:

$$\begin{aligned} ICM(S) &= a\, C_s + b\, C_c + c\, C_g \\ &= a\,(a_1 O + b_1 E) + b \sum \alpha_i(c_i) + c \sum \beta_j(f_j) \end{aligned}$$

The proposed metric seems to be logical and fits the intuitive understanding but is not the sole criteria for deciding the complexity size of the software component on the basis of the computed value of this metric. More work towards the validation of this metric is suggested as future directions by taking into consideration several software components from the software organisations adopting CBSE.

7. Conclusions and Future Directions

In this paper, we have proposed an interface complexity metric for a software component based on which we can control the complexity of a software component by keeping it simple. The complexity of a software component is kept within reasonable limits while fulfilling the desired functionality. This is decided on the basis of the value of the proposed metric. The proposed metric appears to be logical and fits the intuitive understanding but is not the sole criteria for deciding the complexity size of the software component on the basis of the computed value of this metric.

More work towards the validation of this metric is suggested as future directions by taking into consideration several software components from the software organisations adopting CBSE.

References

[1] Gill N.S., Grover P.S. (2003): Component-based Measurement: Few Useful Guidelines. ACM SIGSOFT SEN Volume 28 No. 6, pp. 30.

[2] Han, Jun. (1998): Characterisation of Components. Proceedings of International Workshop on Component-Based Software Engineering.

[3] Brereton, B., Budgen, D. (2000): Component-Based Systems: A Classification of Issues. IEEE Computer, pp. 54-62.

[4] Gill N.S., deCesare Sergio, Lycett Mark (2002): Measurement of Component-based Software: Some Important Issues. UKAIS 2002, pp. 373-381.

[5] Chidamber, Shyam R., Kemerer, Chris F. (1994): A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), pp. 476-493.

[6] D' Souza, Desmond F., Wills, Alan C. (1999): Objects, Components, and Frameworks with UML. Addison-Wesley. United States of America. First Edition.

[7] Etzkorn, L.H., Hughes Jr., W.E., Davis, C.G. (2001): Automated Reusability Quality Analysis of OO Legacy Software. Information and Software Technology. 43, pp. 295-308.

[8] Fenton, Norman E., Neil Martin (1999): Software Metrics: Success, Failures and New Directions. The Journal of Systems and Software. 47, pp. 149-157.

[9] Gill N.S. (2002): Software Engineering. Khanna Book Publishing Company(P) Ltd., New Delhi.

[10] Hevner, A.R. (1997). Phase Containment Metrics for Software Quality Improvement. Information and Software Technology. 39, pp. 867-877.

[11] Gill N.S. (2003): Reusability Issues in Component-Based Development. ACM SIGSOFT SEN Volume 28 No. 6, pp. 30. [9] Kamiya, T., Kusumoto S., Inoue K., Mohri Y. (1999): Empirical Evaluation of Reuse Sensitiveness of Complexity Metrics. Information and Software Technology. 41, pp. 297-305.

[12] Sedigh-Ali, S., Ghafoor, A., Paul, Raymond A.(2001): Metrics-Guided Quality Management for Component-Based Software Systems. Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01).http://dlib2.computer.org/conferen/compsac/1372/pdf/13720303.pdf.

[13] Sedigh-Ali, S., Ghafoor, A., Paul, Raymond A. May (2001):. Software Engineering Metrics for COTS-Based Systems. IEEE Computer, pp. 44-50.

[14] Brown, Alan W., Wallnau, Kurt C. (1998): The Current State of CBSE. IEEE Software, pp. 37-46.